Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master's Thesis

Unified String Dictionary in DuckDB

Author: Omid Afroozeh (2802676)

1st supervisor:	Prof. Dr. Peter Boncz
daily supervisor:	Paul Groß
2nd reader:	Dr. Pedro Holanda

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

August 14, 2025

I dedicate this thesis to my family, Mom, Dad, Ali, Amir, and Azim, for their constant support throughout the past two years.

Abstract

Strings are the most common data type stored in real-world datasets, accounting for up to 50% of all columns. Moreover, they are computationally expensive to process due to their variable length and potentially large size. Compared to fixed-size numerical data types, common operations such as hashing, checking for equality, and copying are significantly more costly for strings. To address this challenge in DuckDB, an analytical, high-performance database system, we propose to implement the Unified String Dictionary (USD), a dictionary-like data structure, which is constructed at runtime for each query. Inspired by the Unique Strings Self-align Region (USSR), USD enables column-wide compressed execution by unifying strings originating from different sources, such as per-block dictionaries in storage. This leads to faster string processing, particularly in materializing operators such as hash join and hash aggregate. USD integration with DuckDB involves a dedicated operator, a custom optimizer rule, and modifications to the execution engine. Our evaluation demonstrates performance improvements of 1.3× to 7× on string-heavy workloads, especially for low-domain cardinality columns. However, these benefits diminish as cardinality increases, possibly leading to performance regression. To prevent this, we enforce defensive constraints that disable USD when high-cardinality columns are detected.

Acknowledgements

First and foremost, I would like to thank Professor Peter Boncz for his guidance throughout the past 6 months and all the interesting conversations we had during our meetings. Special thanks to my daily supervisor, Paul Groß, for his support throughout this thesis. Our frequent discussions were incredibly helpful in shaping this thesis. Finally, much gratitude to everyone at CWI database architectures group for making it a wonderful place for research, especially the table tennis crew!

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1

Introduction

Various studies on real-world workloads, such as "Get Real" [52], Redset [49], and SchemaPile [16], have shown that strings are prevalent in data processing, containing more than half of all table columns. However, strings are a much less researched subject compared to numerical data [9], despite being more troublesome to process. For instance, strings have a much larger memory footprint, which means that processing them typically results in a significant number of cache misses, high memory usage, and even spillage to disk. Furthermore, comparing strings requires iterating over each character as compared to a single instruction for numerical comparisons. Moreover, their variable size makes them less likely to be accelerated by modern hardware capabilities such as SIMD [23; 59]. Given all the reasons discussed, if a database system could directly execute queries on the compressed form of strings, it would gain a lot of benefit from less memory footprint and avoid the decompression overhead [12; 20]. Yet, executing on the compressed strings is not trivial [2; 23], and changing already established execution engines requires significant engineering effort across various layers of the system.

In this thesis, we focus on addressing the challenges of string processing within DuckDB [42]. We are particularly interested in DuckDB since it is open-source, features a state-of-the-art query execution engine, and is immensely popular. Specifically, we aim to improve string processing by implementing a per-query dictionary similar to the approach used in Unique Strings Self-aligned Region (USSR) [23] but in DuckDB. This dictionary will be built at run-time, and it will contain the most common strings of a particular query. These strings will benefit from faster operations, such as equality checks, applying hash functions, and copying.

This leads us to the following research questions, which we aim to answer in the remainder of this thesis:

RQ1: How can we implement and integrate a per-query, global dictionary for strings in DuckDB? Given that the USSR data structure was only implemented in Vectorwise, how would this adaptation look in a more modern system such as DuckDB?

RQ2: Considering the multi-threaded execution environment of DuckDB, how can we efficiently implement a global dictionary that achieves parallel efficiency?

RQ3: How much performance can be gained for string-heavy workloads by implementing such a data structure in DuckDB? Is there a possible chance of performance regression? If so, how to avoid it?

1.1 Contribution

- **Literature review**: We present a survey of dictionary usage, compressed execution, and data structures specifically designed for string processing. We analyze the adoption of these methods in popular database systems.
- **Unified String Dictionary implementation**: We contribute an open-source implementation of Unified String Dictionary, which is an adaptation of USSR implemented in the closed-source database Vectorwise [62].
- **Integration of Unified String Dictionary in DuckDB**: We discuss in detail how to integrate a query-wide dictionary in an already established execution engine as non-intrusively as possible.

All of our proposed changes have been submitted in a Pull Request (PR) to DuckDB[1]. Additionally, we fixed a minor issue that prevented dictionary vectors from being emitted[2]; This fix has already been merged into the main branch.

1.2 Outline

In Chapter 2, we present the preliminary background required to understand the rest of the thesis. This includes topics related to database management systems, compression methods, DuckDB, and the Unique String Self-align Region. Chapter 3 presents our literature study on various methods used in database systems to accelerate string processing. The Unified String Dictionary is introduced in Chapter 4, where we detail how we implemented and

[1] `https://github.com/duckdb/duckdb/pull/18184`
[2] `https://github.com/duckdb/duckdb/pull/17471`

integrated this feature into DuckDB. Chapter 5 details our benchmarking results. Chapter 6 summarizes our attempts to adapt the limited-size per-query dictionary to datasets with high domain cardinality. Finally, in Chapter 7, we conclude our work and discuss possible directions for future research.

2

Background

2.1 Database management systems

Database Management Systems (DBMSs) are fundamental software systems designed to efficiently manage, store, retrieve, and manipulate data. They are the backbone of modern applications, from small-scale software systems to large-scale enterprise and cloud services. A DBMS gives users a high-level abstraction of the data. It enables users and applications to interact with data using declarative query languages, such as SQL. Users do not need to worry about how the data is stored, how indexes are used, or how I/O operations are optimized.

Database systems come in many forms, specialized for different workloads, data models, and performance requirements. The traditional model is the relational database, where data is stored in tables with a fixed schema and queried using SQL. Examples include PostgreSQL [22], DuckDB [42], MySQL [36].

Database systems are optimized for different types of workloads, mainly Online transaction processing (OLTP) and Online analytical processing (OLAP). OLTP systems are optimized for write-heavy workloads, handling many short, concurrent transactions, such as inserts, updates, and deletes. In contrast, OLAP systems are designed for complex read-heavy queries over large volumes of data, often used in reporting, analytics, and business intelligence. These differences also influence storage formats: OLTP systems typically use row-based storage, which is efficient for accessing entire records. OLAP systems often use columnar storage, where data belonging to each column is stored together. This layout speeds up analytical queries that scan only a few columns across many rows, making it more efficient for read-heavy workloads. Hybrid systems also exist, combining features of

both OLTP and OLAP systems. Examples of such systems include SAP HANA [17] and Hyper [26].

Database systems can also differ in how they are deployed and accessed. Server-based databases like PostgreSQL and MySQL run as separate services and handle requests from client applications over a network or local connection. They support multiple users, concurrent queries, and are well-suited for large-scale or multi-user environments. On the other hand, in-process databases such as SQLite or DuckDB run within the same process as the application that accesses them. They do not require a separate server, are lightweight, and easy to set up.

2.1.1 Relational database systems

In this thesis, we focus on DuckDB, a relational DBMS. The relational model is a foundational concept in database theory and remains the dominant paradigm for data storage and processing in analytical systems.

Edgar F. Codd first proposed the relational model in 1970 [14]. It organizes data into relations, more commonly referred to as tables, where each relation consists of a set of tuples and attributes. This model offers a high level of data abstraction, allowing users to reason about data without needing to understand its physical storage.

The relational model relies on declarative querying using SQL, a special-purpose language introduced in the 1970s. With SQL, users describe what data they want, not how to get it. This separation allows the database to choose the most efficient way to execute the query.

The relational model also focuses on key concepts such as schemas, primary and foreign keys, data normalization, and set-based operations, including joins, selections, and projections. These features make it well-suited for structured data with clear relationships and constraints.

2.1.2 Query processing overview

Query processing refers to the steps a database system takes to turn a SQL query into results. It involves parsing, binding, planning, and executing the query. These steps ensure that the query runs efficiently and returns correct results. The required steps are depicted in Figure 2.1.

When the user submits an SQL query, the system first parses it. The parser checks if the query is valid and converts it into an internal tree structure called the logical plan. This

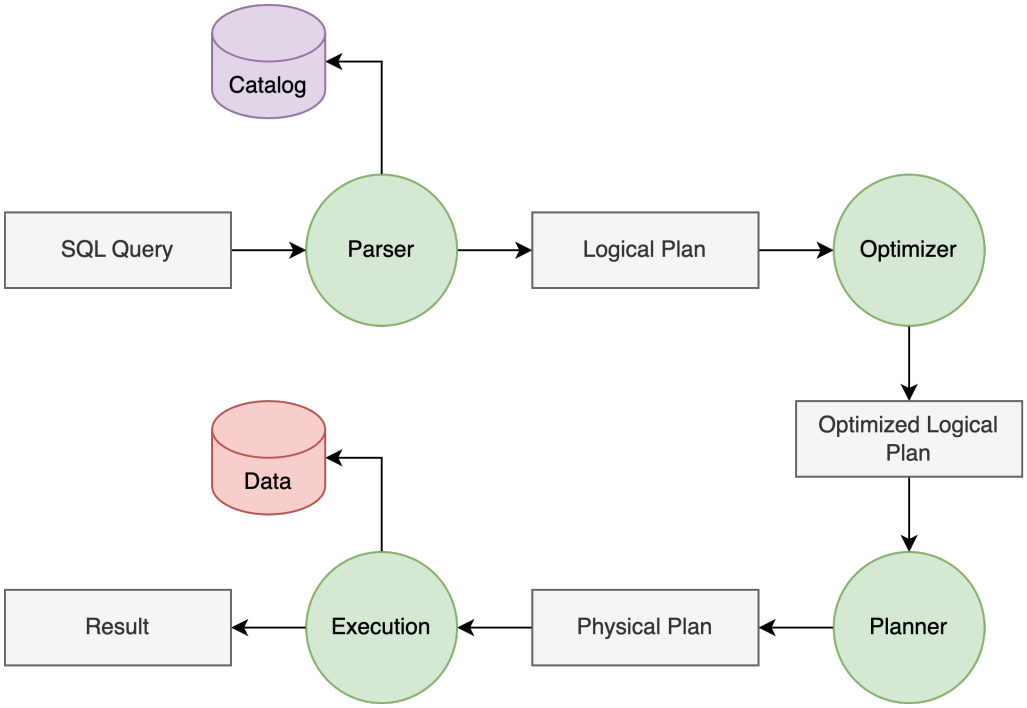


Figure 2.1: Query processing steps

plan describes the operations needed to produce the output, such as scans, filters, joins, and aggregations.

Next, the system optimizes this logical plan. The goal is to find a more efficient version of the query plan. The optimizer may push filters closer to scans, reorder joins, or rewrite expressions. After optimization, the system generates a physical plan. This plan specifies how each operation will be performed and in what order.

Finally, the system executes the physical plan, and the operators are run to process the data. There are different paradigms for a DBMS's execution engine. For analytical systems, vectorized execution and Just-In-Time (JIT) compilation are the dominant, state-of-the-art approaches.

Systems such as MonetDB/X100 [10], Vectorwise [62], and DuckDB take advantage of the benefits provided by vectorized execution, where data is processed in small batches called vectors. On the other hand, systems such as Hyper [26] and Umbra [38]apply JIT compilation.

2.1.3 Materializing operators

A query plan can contain many different operators, each responsible for a part of the query. Some operators scan data from tables, while others perform filter, join, or aggregate operations on the data.

Some operators can start producing output as soon as they receive input. Others, however, need to see all of the input data before they can emit any results. These are known as materializing or blocking operators. They first materialize the data in memory, storing it in different forms, before they can proceed. The materializing operators are the main point of interest for this thesis, as most expensive operations on strings take place there. In analytical systems, these operators can become the bottleneck for query execution, as large volumes of data need to be processed, and the query intermediates can quickly exceed the available memory space, leading to disk spillage [27].

In particular, we will discuss two specific algorithms used for the materializing operator of join and aggregation: hash join and hash aggregate.

Hash join

Join operators combine rows from two or more tables based on a join condition. In analytical queries, the most common algorithm used for this operator is the hash join. A hash join builds a hash table on the smaller input table, called the build side.

A hash table is a data structure that maps keys to values for fast lookups. In the context of a join, the keys are the values from the join column of the build side. To store a row in the hash table, the system applies a hash function to the join key. A hash function takes the key from an arbitrarily large domain and maps it to a fixed-size integer value, called a hash value. This value can be used to determine where the key should be stored.

Once the hash table is built, the system reads the larger input, often referred to as the probe side. For each row on the probe side, it applies the same hash function to the join key and then checks the corresponding location in the hash table. If matching rows are found, they are combined and added to the result. To confirm that the keys truly match, an equality check is also performed.

In DuckDB, the hash table does not only store the join keys but also includes any additional columns selected by the query. These are known as payload columns or non-key columns, and they are materialized in memory alongside the keys. Under memory pressure, for example, when the hash table grows past the available memory limit, the database can spill the materialized data back to disk.

Hash aggregate

The group-by operator is used to compute aggregate functions, such as `SUM`, `COUNT`, or `AVG`, often grouped by one or more columns. In analytical queries, the hash-based algorithm is commonly used for this purpose, named `hash_aggregate`.

Just like in a hash join, the `hash_aggregate` operator uses a hash table. The system applies a hash function to the group-by keys to determine where to store the aggregation state. Each unique key creates an entry in the hash table.

As input rows are processed, the operator looks up the corresponding entry in the hash table and updates the aggregate state, for example, by adding a value to the accumulated sum or increasing the count.

2.2 Compression

Compression plays an important role in analytical database systems by reducing overall storage footprint, memory usage, and I/O costs. It is especially important for string data, which is often larger and more memory-intensive than numerical data. In this section, we discuss general-purpose and lightweight compression techniques, with a particular focus on lightweight techniques, as they are prominent in DuckDB and form the central focus of this thesis.

2.2.1 General-purpose compression

This category of methods reduces data volume by identifying duplicate values within the byte stream. Popular methods include Zstd [1], Snappy [2], and LZ4[3]. These techniques often achieve high compression ratios. However, due to their high CPU overhead during decompression and their inability to provide access to individual values without decompressing larger blocks of data, they are not used directly during query execution.

2.2.2 Lightweight compression

Unlike general-purpose methods, lightweight compression techniques attempt to find patterns within the data itself to reduce data volume. They are particularly effective in column-store databases, where data from the same domain [2]. These methods support

[1] `https://github.com/facebook/zstd`
[2] `https://github.com/google/snappy`
[3] `https://lz4.org/`

fine-grained access to individual values, making them suitable candidates for use during query execution.

Bit packing

Bit packing is a lightweight compression technique that reduces storage by representing fixed-width values using the minimum number of bits required to store them. For example, value 50 can be stored using only 6 bits (0b110010) instead of 32 or 64 bits.

Run-Length Encoding

Run-Length Encoding (RLE) compresses data by representing consecutive repeated values as a single value paired with its run length. This method is especially effective for sorted or low-cardinality columns, where long sequences of identical values are common.

Dictionary encoding

Dictionary encoding is perhaps the most common compression method used in databases [2; 61; 25; 43]. The idea is to replace long, variable-length values with small integers, as depicted in Figure 2.2. There are many variations of dictionary encoding regarding the codes generated for unique values, such as order-preserving codes [7] or frequency-based codes [43]. The implementation typically involves an array containing all the unique values belonging to a domain, known as a dictionary. Each value in the column is assigned an integer code, which can be used to index into the array. Dictionary encoding is highly effective when the domain cardinality is low to medium, as it results in many repetitions and, therefore, a better compression ratio. Typically, another layer of numerical compression methods, such as run-length encoding and bit packing, is applied on top of the encoded integer values.

Fast Static Symbol Table (FSST)

One limitation of dictionary compression is that it works best when strings are complete duplicates; If two strings differ even slightly, the dictionary must store both entries at full size, diminishing the benefits of compression. For datasets where strings are very similar but not fully identical, FSST can be useful. This compression method, introduced by Boncz et al. [9], is a lightweight string compression technique designed with database workload in mind. It replaces frequently occurring substrings of 1–8 bytes with single-byte codes, allowing for fast decompression and, crucially, random access to individual strings. To encode and decode values with FSST, a symbol table is required. Since its introduction,

FSST has become a core string compression method in DuckDB, used both as a standalone technique and in combination with dictionary compression (DICT_FSST[4]). FastLanes [5] file format has also adopted FSST compression.

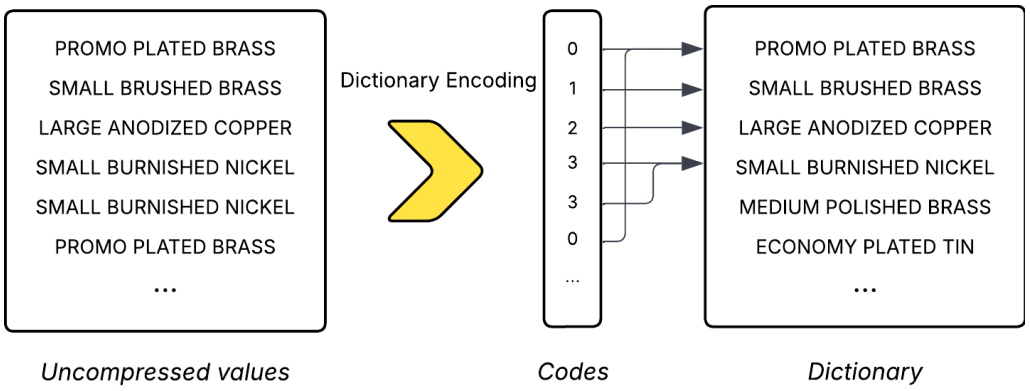


Figure 2.2: An example of dictionary encoding applied to the `p_type` column in TPC-H [48]

2.3 DuckDB

DuckDB is a high-performance, analytical, columnar-storage Relational Database Management System (RDBMS), supporting the Postgres SQL dialect. It is open-source under the MIT license and popular among data scientists, with 15,162,132 monthly downloads for its Python package [40]. In the following sections, we discuss various features available in DuckDB as they pertain to this thesis.

2.3.1 Execution engine

DuckDB features a push-based, vectorized query execution engine, enabling fast execution for analytical workloads. Like similar OLAP systems, it relies heavily on columnar storage for its execution model. In the following, we will discuss key concepts available in DuckDB's execution engine.

Vectors

Vectors are the unit of data that flow through DuckDB's execution engine. Each vector typically contains 2048 values of the same data type and represents a batch of columnar data. This vectorized model allows for efficient computation on blocks of data, reducing

[4] `https://github.com/duckdb/duckdb/pull/15637`
[5] `https://github.com/cwida/FastLanes`

the overhead of tuple-at-a-time processing [10]. DuckDB uses several specialized vector types, each optimized for particular use cases and compression methods. These vector types enable DuckDB to delay decompression as long as possible.

Each vector in DuckDB is generated from the `Vector` class [6]. This class has a few key attributes as described in the following:

- `vector_type`: Determines the type of the vector, specifying the physical representation of the data.
- `type`: Determines the primitive data type that the vector holds, such as integer, float, or string.
- `buffer`: A shared pointer to a `VectorBuffer` class that holds the data related to the vector itself.
- `auxiliary`: A shared pointer to a `VectorBuffer` class that holds the auxiliary data of a vector.

The `VectorBuffer` base class is inherited by different vectors and used to store the vector metadata.

Flat vector

This is the most common and straightforward vector type, where data is stored in a simple, uncompressed flat array. DuckDB provides an extensive API to access the underlying data in flat vectors using the `FlatVector::GetData` or `FlatVector::GetValue`.

Constant vector

A constant vector in DuckDB physically stores a single constant value, but logically represents a vector of up to 2048 values, all of which are equal to that constant.

Dictionary vector

To represent the data that is encoded with dictionary compression, DuckDB makes use of dictionary vectors. It uses the buffer attribute of the vector to store a `DictionaryBuffer`, containing a `SelectionVector`, an optional integer value for dictionary size, and a string value holding the dictionary ID. The `SelectionVector` contains the indices to the dictionary. `dictionary_size` determines how many unique values are available in the dictionary. The

[6] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/vector.hpp`

`dictionary_id` uniquely identifies which dictionary a vector belongs to. This value is set for dictionary vectors emitted from the storage for DuckDB's database files and Parquet reader.

Furthermore, this vector type uses the auxiliary attribute in the vector to store the `VectorChildBuffer`. This special vector buffer stores the unique entries in the dictionary in a flat vector. All of the described variables are accessible through an extensive API provided by DuckDB.

Unified Vector Format

During query execution, DuckDB often converts vectors into the Unified Vector Format (UVF). This representation of the data provides a consistent abstraction for all vector types. Therefore, the execution operators can directly operate on the UVF, hiding away the complexities of different compressions and data representations.

DataChunk

A DataChunk contains a set of vectors. This class represents the intermediate data moved between physical operators. All of the vectors are the same length and contain a subset of a relation. Physical operators in DuckDB use DataChunks as their primary interface: they receive input chunks from upstream operators, perform computation over the data, and write their output into another DataChunk that is then passed downstream.

Pipeline Execution

The physical query plan is divided into different pipelines that are executed by DuckDB's execution engine. Each pipeline starts with a source operator (e.g., a table scan) that produces data, and ends with a sink operator that generates the final results. DuckDB's push-based execution engine relies on the pipeline executor. This component pushes data from one operator to the next when ready. Individual operators themselves are passive; they only report their status, such as whether they require more data or have finished executing.

2.3.2 Storage

DuckDB has its own database file format for data storage. It is somewhat similar to Parquet in the way that data is laid out in horizontal cuts of 120k values, called row groups. Each row group can be comprised of several fixed-size blocks of 256KB, referred to as *Column Segments*. DuckDB applies compression per column segment.

In terms of compression methods, DuckDB heavily relies on light-weight compression techniques[41], such as bit packing, FSST, and dictionary encoding.

2.3.3 String representation

DuckDB follows Umbra in its string representation [27], implemented as the *string_t* [7] class in DuckDB. This class represents immutable strings that are used in the execution engine. The string type in DuckDB is a 16-byte header containing the length and the first four bytes of the string, known as the prefix. Moreover, based on the string length, the last 8 bytes can be used for inlining the rest of the string or containing a pointer to the full string in the heap.

Implementation

DuckDB has two variations of strings: inlined and non-inlined. These variations are implemented using C++ unions. As a result, both categories share the same memory layout. Thus, any code path that operates on strings must carefully check whether a string is inlined or not before performing operations. Furthermore, DuckDB provides an extensive API to access the relevant attributes for both string variations, depending on their length. Listing 2.1 specifies the private members of `string_t` for both variations.

```
private:
  union {
    struct {
      uint32_t length;
      char prefix[4];
      char *ptr;
    } pointer;
    struct {
      uint32_t length;
      char inlined[12];
    } inlined;
  } value;
```

Listing 2.1: Private attributes for DuckDB's string type. Using C++ unions, both struct data types for inlined and non-inlined variations share the same memory layout.

[7] https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/string_type.hpp

Equality check

To check for equality between two strings, DuckDB loads the first 8 bytes of both strings into a `uint64_t`. These variables now contain both the string length and the prefix. By comparing these integers, DuckDB already prunes the majority of equality checks that result in negatives without needing to dereference the pointers or perform a `memcmp` on possibly long strings.

If the lengths and prefixes are equal, DuckDB performs the same operation on the next 8 bytes. Since the string lengths are already known to be equal, this comparison will either be between the rest of the inlined strings or between the pointers to the heap-backed strings. If this comparison is not equal, the system first checks whether one of the strings is not inlined. If so, a full `memcmp` is performed on the pointers stored in `string_t`. The other code paths all lead to strings not being equal.

2.3.4 Relevant optimizations

DuckDB features many optimizations to enable its high-performance execution engine. Two specific optimizations in particular influence the design decisions and implementation aspects of this thesis: *TryAddCompressedGroups* and *Compressed Materialization*.

TryAddCompressedGroups

This optimization was recently added to DuckDB's `hash_aggregate` operator [8]. TryAddCompressedGroups enables executing aggregation directly on the unique values in the per-block dictionaries. Instead of applying hash functions on the indices in the selection vector, unique values are used to probe the hash table directly. Furthermore, this optimization caches the groups found for each unique value to be used for subsequent vectors. Although limited to single-column group-by and bounded by the total number of dictionary values, TryAddCompressedGroups leads to a considerable performance gain by not requiring hashing and probing every value.

Compressed Materialization (CM)

Added back in 2023, this optimization [9]places projection operators before and after each materializing operator, such as join and aggregation. The goal of this optimization is to represent integer and string values in different forms that require less memory footprint and enable faster operations. For example, if CM detects that strings in an attribute are

[8] `https://github.com/duckdb/duckdb/pull/15152`
[9] `https://github.com/duckdb/duckdb/pull/7644`

all less than 15 characters long, it will store them in a hugeint (16-byte) datatype alongside the string length. By doing this, operations such as hashing applied to the data in the materializing operator are significantly faster, as they need to deal with integer values instead of strings. After the materializing operator is finished, CM will decompress the data to its original form for the rest of the execution.

2.4 Compressed execution

Compressed execution is the ability of a query engine to directly operate on compressed data during query execution [2], without requiring full decompression beforehand. This approach is particularly beneficial in analytical systems, where the volume of data can be large and decompression costs can become a bottleneck. By preserving compression throughout the execution pipeline, the system reduces memory footprint, leads to less spillage to disk, and minimizes the amount of computation needed by operating on simpler or more compact representations of the data.

2.5 Unique Strings Self-align Region

The USSR is a data structure proposed by Gubner et al. [23], which is created on the fly for each query, containing the most common strings. The strings contained within the USSR are unique. As the scan operator iterates through the storage, it inserts strings into the USSR until it is full. Once full, strings have to be allocated the same way as before on the heap. By removing duplicate strings in this manner, the peak memory usage is reduced. One benefit of USSR is that it can be integrated into existing systems without much trouble since the strings are still represented as pointers whether they are backed by the heap or the USSR. This data structure would also solve the problem of dictionaries being a local feature for each row group, and would enable execution across the entire relation.

The size of the USSR is limited to 768 kB to remain small, efficient, and most importantly, cache-resident. USSR is composed of two regions, as depicted by Figure 2.3, Data Region (512 kB) and a linear probing hash table (256 kB). The data region contains the materialized strings and their precomputed hash. The data is stored in 64k slots of 8 bytes in a memory-aligned location. For efficient lookup, a fast linear probing hash table is used. Having such a data structure can help accelerate hash and comparison operations for the most common strings. After verifying whether a string resides in the USSR or not, by applying a mask on

the pointer, the pre-computed hash can be accessed directly by simple pointer arithmetic. Furthermore, exploiting the fact that only unique strings reside in USSR, equal pointers into the USSR mean the strings themselves are guaranteed to be equal.

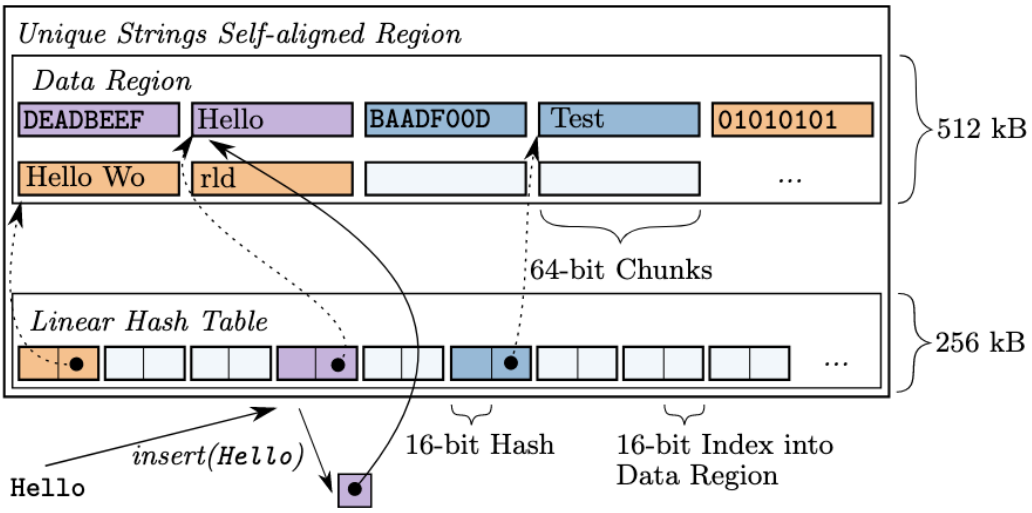


Figure 2.3: Unique Strings Self-aligned Region (image borrowed from [23])

To maximize the gains from having a global dictionary, only the strings in storage that are already dictionary-compressed should be inserted. Other primary candidates can be the string constants in the query text, such as the filter predicates.

3

Literature Study and related works

In this chapter, we review the available literature on topics related to string processing in analytical database systems. These include: compressed execution, use of dictionaries, and data structures designed specifically for strings. The structure is as follows: Section 3.1 explores compressed execution. Section 3.2 discusses various types of dictionaries used in databases for both storage and execution. Section 3.3 discusses various data structures that are designed specifically for strings in databases. Finally, we summarize our work regarding the explored methods, their benefits, and their adoption status.

3.1 Compressed execution

Historically, compression has been used to reduce the data storage size and minimize the I/O cost of processing large volumes of data. Different compression methods and their benefits for database performance have been extensively studied [3; 44; 2; 45]. Some systems eagerly decompress data upon loading it into memory, either to evaluate filter predicates or to populate internal data structures. This approach simplifies the database system's architecture and abstracts away all complications to the storage layer. However, literature suggests that significant performance gain is possible if the decompression is delayed as much as possible. This concept is further elevated for strings as they dominate query execution with their large and variable size. If a database could delay decompression and perform various query processing steps on the encoded string data, considerable performance gain could be expected.

3.1.1 Foundational works

One of the first papers that discusses the idea of executing queries on compressed data is Graefe and Shapiro [20]. They state that the performance of database systems relies on the available memory, whether in buffer pages or the memory used for operators' internal data structures, which is used for query processing. For the sake of simplicity, it was assumed that each domain is compressed with the same scheme. This paper primarily adopts a theoretical perspective, with evaluation based on an analysis of the number of I/O operations required, rather than a concrete implementation.

Moreover, Westmann et al. [53] also explored the idea of compressed databases. This work is significant since it was one of the first papers to implement the proposed ideas in their experimental database system, AODB, and report the performance results. They specify the importance of using lightweight compression on the finest granular level possible, field-level compression, to reduce the CPU cost of decompression. Furthermore, they assume a single compression scheme for the entire attribute. Regarding changes required for the query executor, they propose to extend the traditional `next()` function in the iterator model. To this end, `next()` will return pointers to an array of tuples rather than a single tuple. Moreover, to avoid decompression of the same field multiple times during query execution, `next()` is also extended to return already decompressed values.

In their influential work, Abadi et al. [2] explored the integration of compression into the query execution layer of columnar databases. While earlier works [8; 10] had already demonstrated the performance advantages of columnar storage and lazy decompression in read-heavy workloads, Abadi et al. focused specifically on lightweight compression schemes. Their work highlighted how columnar storage further increases the effectiveness of compression by increasing data similarity within columns and enabling encodings like run-length encoding, dictionary encoding, and bit-vector representations. They emphasized that compression should not be a storage feature alone but rather as an integral part of the execution engine. To achieve this goal, they extended the C-Store system with a compression submodule to perform various operations on compressed data.

3.1.2 Predicate evaluation

The table scan operator can be regarded as the baseline performance of an analytical database system, as OLAP workloads typically involve reading large volumes of data. To this end, the role of predicate evaluation is further elevated, and performing the evaluation on compressed forms of data has been extensively studied in the context of scans by [24;

33; 29; 55]. Furthermore, the following systems also support applying filter predicates on compressed data: IBM DB2 with BLU acceleration [43], SQL Server with its columnstore index (CSI) [30], QuickStep [39], and DuckDB. The general approach used in these systems is to push the predicate evaluation down to the storage layer. By doing so, they can apply the predicate directly to the compressed data, avoiding the need to decode unnecessary values.

This strategy applies to various string compression techniques, such as FSST and dictionary encoding. For FSST, Boncz et al. [9] describe how their technique was integrated into the Umbra [38] database as a proof of concept, enabling compressed execution of equality predicates without the need for full decompression. In the case of dictionary encoding, exact-match predicates can be efficiently evaluated by scanning the dictionary for matching raw strings and returning the corresponding integer codes.

3.1.3 Materializing operators

Materializing operators, such as join and aggregation, are where the most expensive operations on strings happen. These operators could potentially produce large query intermediate results that would require spillage to disk. It is highly desirable to reduce the memory footprint of these operators and perform various operations on the compressed data.

The paper by Lee et al. [31] introduces methods for executing joins directly on compressed data. Instead of requiring shared dictionaries across all join columns, the system dynamically translates encoded values from one domain into another at runtime. This is necessary because using the same dictionary for both sides of the join often results in limited compression, as it assumes all join relationships are known ahead of time and cannot leverage differences in data distribution between columns. The translation can be applied on either side of the join, depending on which is more efficient.

Their system also uses a catch-all cell to store rows containing values that cannot be encoded using the current dictionary. Even if only one column in the row is unencodable, the full row is stored uncompressed. This design improves flexibility for handling evolving data but complicates translation, as the same value might appear in both encoded and unencoded form during a join. Figure 3.1 depicts an example of their domain partitioning in the presence of a catch-all cell.

In addition to join columns, the paper extends compression to payload data using on-the-fly (OTF) encoding, which dynamically encodes payload values during query execution.

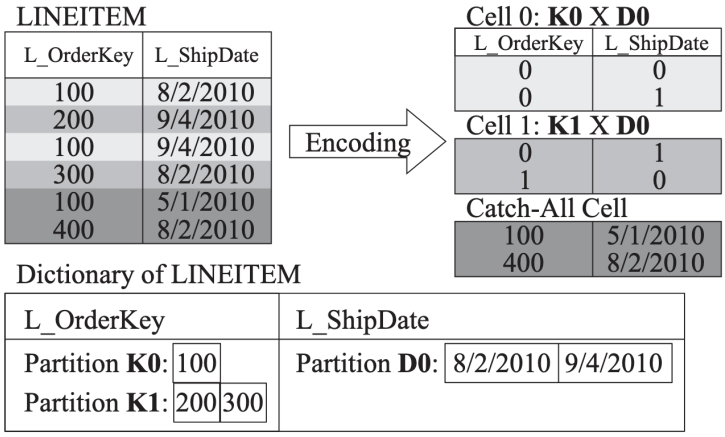


Figure 3.1: Data encoding with catch-all cell (image borrowed from [31])

This reduces hash table sizes and improves memory efficiency. All of the proposed ideas have been implemented and evaluated in IBM's Informix Warehouse Accelerator (IWA).

Regarding the group-by operation, systems such as SAP HANA [17] and IBM DB2 [43] support executing aggregation on the compressed column. DuckDB has also recently added limited support for dictionary and constant compressed data in its aggregate hash table operator.

3.1.4 Query optimization

The role of the query optimizer in compressed databases has been extensively studied. Westmann et al. [53] argue that the query optimizer should be made *compression-aware* by modifying the cost model to account for both the CPU overhead of decompression and the reduced I/O costs associated with compressed data. In doing so, the optimizer can more accurately estimate the performance trade-offs of different execution strategies. One important implication of this enhanced cost model is its impact on join ordering. For example, the optimizer can choose join methods and orders that exploit the smaller memory footprint of compressed relations.

Chen et al. [12] further investigate how a cost-based optimizer should handle compressed string attributes. They demonstrate that applying eager or lazy decompression uniformly can result in suboptimal plans for strings. To address this, they introduce transient decompression: an operator decompresses an attribute only for its local computation and still outputs the value in compressed form. This approach has the benefit of keeping intermediate results small. While transient decompression typically outperforms the other

strategies for numerical attributes, since they are cheap to decompress, its benefit for strings depends on the trade-off between I/O savings and the repeated decompression cost. Hence, the optimizer must choose, per operator, if and where to decompress. The authors offload solving this problem to a compression-aware query optimizer and propose one provably optimal dynamic programming algorithm, along with two faster heuristic approaches.

3.1.5 Query executor and code explosion problem

As stated by Abadi et al. [2], enhancing the query executor to operate on data that might be compressed with different encodings will lead to a code explosion in operators. To illustrate this, they provide pseudocode for a nested-loop join that can operate directly on compressed data. Although this would result in many useful optimizations, maintenance and binary size are real issues. That is why they propose an abstraction that hides many of the specific complexities. To this end, each compressed block exposes an API for different properties of the stored data. Furthermore, methods to get direct access to the underlying data and various information about the block are provided. This design allows execution operators to leverage a common interface for different compression formats, enabling compressed execution while keeping the code maintainable. They implemented their work in the C-Store database [11]. Another example of this problem can be seen in DuckDB's execution engine. DuckDB supports various lightweight compression techniques [41] along with specific Vector types to support them during execution. In this approach, the data does not need to be readily decompressed when brought into memory. This form of lazy decompression is excellent in providing various benefits for the entire system. However, this poses the same challenge faced by Abadi, as the code for each operator needs to take into account what type of Vector is currently being executed on and to enforce compression-specific optimizations. This can be further worsened by the join operator, as different columns may be compressed with different encodings. To remedy this, DuckDB resorts to Unified Vector Format. The operator can access the underlying data without needing to consider the specific Vector type. This is a rather good middle ground, as it avoids completely flattening the data for certain Vector types while keeping the code small and maintainable. However, this approach will deny the operator the opportunity to use compression-specific optimizations. The issue of finding simple methods to enable compressed execution without requiring significant changes to existing engines has also been studied by [35] for Vectowise [62].

3.2 Dictionary

One of the most common forms of enabling compressed execution is the use of dictionaries. If a mapping exists between different uniques values and their corresponding integer codes, the database can attempt to execute entire operations on the encoded data. Joins, aggregations, applying filter predicates, range queries, and sorting can all be exceedingly accelerated using dictionaries. Given the relevance of dictionary encoding and its widespread use for string columns, we aim to discuss the various flavors of dictionaries widely integrated into database systems.

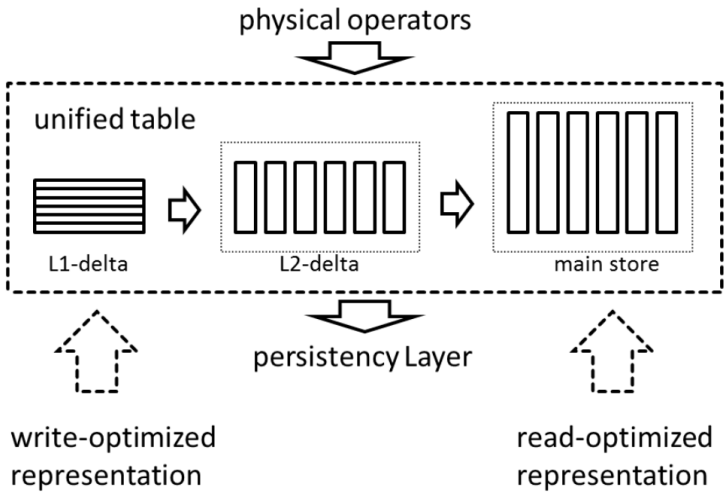


Figure 3.2: Data life cycle in SAP HANA (image borrowed from [47])

3.2.1 Global dictionary

Global dictionaries provide a single mapping between integer codes and unique values for an entire column domain. SAP HANA [17] is an in-memory database that makes full use of global dictionaries [47]within its system for lower storage size and faster query execution. This database enables write-heavy workloads and is also highly efficient in analytical queries. To understand how SAP HANA makes use of dictionaries, first, we need to discuss their data life cycles. Data in SAP HANA is managed through a multi-stage life cycle, as shown in Figure 3.2. It first enters the system via a write-optimized, row-major structure known as the L1-delta. It then propagates to a partially compressed columnar L2-delta structure, and finally reaches the main store, which is a fully compressed and highly optimized for read-heavy queries. Dictionaries are central to this architecture. While

the L1-delta holds raw values without compression, the L2-delta uses unsorted dictionaries to encode values, providing moderate compression and efficient update operations. As data is transitioned to the main store, a global, sorted dictionary is constructed for each column. Every value in the main store is represented by its position in the sorted dictionary.

The use of a unified global dictionary across stages supports efficient query execution by allowing consistent and compact value representation. Operators such as joins, filters, and aggregates can operate directly on encoded values without requiring decompression, resulting in faster query execution. Furthermore, the system dynamically merges and reorganizes dictionaries throughout different stages during asynchronous merge operations.

SAP HANA is not the only major database that utilizes global dictionaries for storage and query execution. DB2 with BLU acceleration [43]is yet another example of a columnar storage database that exploits dictionaries. It uses order-preserving, frequency-based dictionaries to perform all major SQL operations. Equality, range, and in-list predicates are directly applied to compressed values. Moreover, join and aggregations are also supported by this system. Another advantage of dictionary compression for them is being able to leverage SIMD instructions on multiple aligned, bit-packed values simultaneously.

The frequency compression used in DB2 BLU exploits skew in data by assigning the smallest codes to the most frequent values in the column. To further reduce the storage size, page-level compression is applied to make use of local clustering of data. Mini-dictionaries are stored on each page which can benefit from finer-granular compression.

To enable compressed execution on a column level, DB2 BLU provides partition-independent encoding, which is called Global Coding. Join and group by operations make use of this encoding for compressed execution.

Order-preserving dictionary There has been numerous works done on order-preserving dictionaries [4]. The goal of these dictionaries is to enable efficient range queries in addition to all the benefits that are provided by having a global dictionary. Binnig et al. [7] is perhaps the most influential work in this area, which exclusively focuses on string data type. They model a dictionary as a small table that specifies a mapping between string values and integer codes. Then, they propose solutions on how to build indexes on top of this table for efficient access for both encoding and decoding operations. In a data warehousing situation, they model a dictionary as a structure that can support efficient data loading, which requires encoding bulk string values with integer codes. Furthermore, to enable execution on sorted dictionary values, the data structure should support efficient lookups to rewrite the query predicates. Finally, the dictionary should support efficient

decoding in cases where query intermediates need to be materialized or the final result is provided to the users. Figure 3.3 depicts the different requirements for a dictionary-based order-preserving string compression: data loading, query compilation, and query execution.

Furthermore, they propose a shared leaf structure for the index structures used in encoding and decoding. Instead of maintaining two completely separate indexes, they suggest reusing the same leaf pages for both indexes. This is illustrated in Figure 3.4. Each leaf is a cache-sized chunk that stores variable-length strings and their codes. This layout supports efficient access for both encoding and decoding. To save space, strings are compressed incrementally using delta-prefix compression. The authors also propose two cache-conscious index structures that can utilize the leaf design: CS-Prefix-Tree and CS-Array-Trie.

Overall, this work introduces a possible direction for the efficient use of dictionaries in high cardinality columns when the total domain size is unknown in advance, which can be the case in data warehousing.

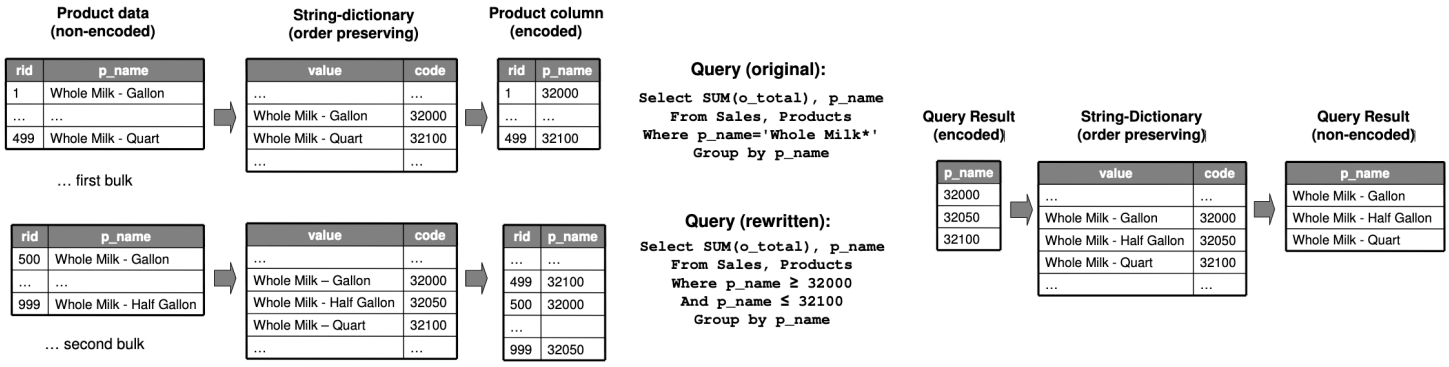


Figure 3.3: Order-preserving dictionary requirements (image borrowed from [7])

Another recent addition to this field is the work of Lue et al.[34], which attempts to bridge the gap between unsorted dictionaries and fully order-preserving dictionaries. Mostly Order Preserving Dictionaries (MOP) introduce a novel approach to dictionary encoding that supports efficient range queries without requiring complete knowledge of the dataset's value domain in advance. MOP tries to estimate the domain through sampling and pre-allocating a key space for the most frequent values. Values that fit within this ordered space benefit from fast, decoding-free range queries, while less common or late-arriving values spill over

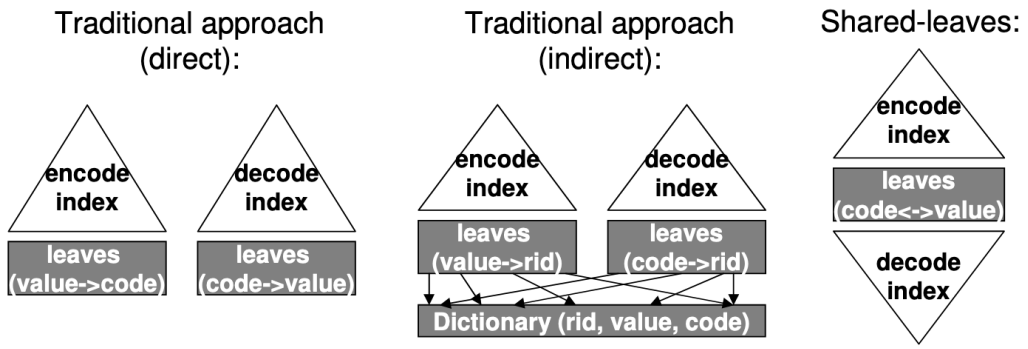


Figure 3.4: Shared leaves structure (image borrowed from [7])

into a disordered section. They also propose the idea of Cascading MOP, which contains multiple ordered sections.

Issues with global dictionaries There are major problems with using global dictionaries in database systems, as pointed out by [9; 23; 18]. First, random access to the dictionary is common throughout the query execution, requiring the dictionary to be memory-resident. However, this can be a considerable hurdle for analytical data systems that must manage datasets larger than main memory. Secondly, managing the global dictionary in a parallel or distributed setting is extremely cumbersome. This is because synchronization between different nodes is required to maintain a consistent mapping between codes and values. Finally, in use cases where inserts, updates, or deletes occur frequently, there is a significant overhead associated with global dictionary maintenance. For example, if new values are added and cannot be represented with the already allocated number of bits for integer codes, the entire dictionary needs to be re-encoded. These issues have precluded their wide adoption.

Given all the problems with global dictionaries for execution, the authors of [23] propose a dynamic dictionary. We have already discussed USSR in full detail in Section 2.5

3.2.2 Per-block dictionary

Given all the issues that a global dictionary can cause, many databases and open file formats resort to using per-block dictionaries. Therefore, dictionaries are seen as a local feature of data that is mainly used to reduce storage size [23]. Unfortunately, the use of a local dictionary for column-wide compressed execution is not possible. Apache Parquet

and ORC file formats both utilize local dictionaries [57]. The encountered values of each column are used to build a dictionary. They are stored as integers using typical lightweight encoding schemes commonly used for numerical data such as Run-length encoding with bit-packing used on top. Moreover, DuckDB also uses per-segment dictionaries similar to Parquet [41]. DuckDB deduplicates string values until it reaches the size of each segment, which is 256KB. Then, it will store the dictionary-encoded segment on disk. Data Blocks [29] makes use of per-block sorted dictionaries. Furthermore, they mention the downside of this approach in terms of redundancy for strings appearing in different blocks.

3.2.3 Differential dictionaries

Recent work by Foufoulas et al. [18] has explored adaptive schemes that aim to balance the trade-offs between local and global dictionary encoding in columnar storage. Local dictionaries enable fast random access and smaller offset sizes, while global dictionaries reduce redundancy across blocks but can be memory-intensive and slower in random access. To address the limitations of both, the authors propose adaptive dictionary compression. The method selectively applies either local or differential dictionaries based on a cost function that considers data distribution, offset sizes, and compression gains. Differential dictionaries encode only the new values relative to previous blocks. Higher compression rates are achieved by eliminating duplicate values across different blocks, and reduced dictionary look-up overhead is another key benefit of this scheme. Figure 3.5 depicts an adaptively encoded attribute.

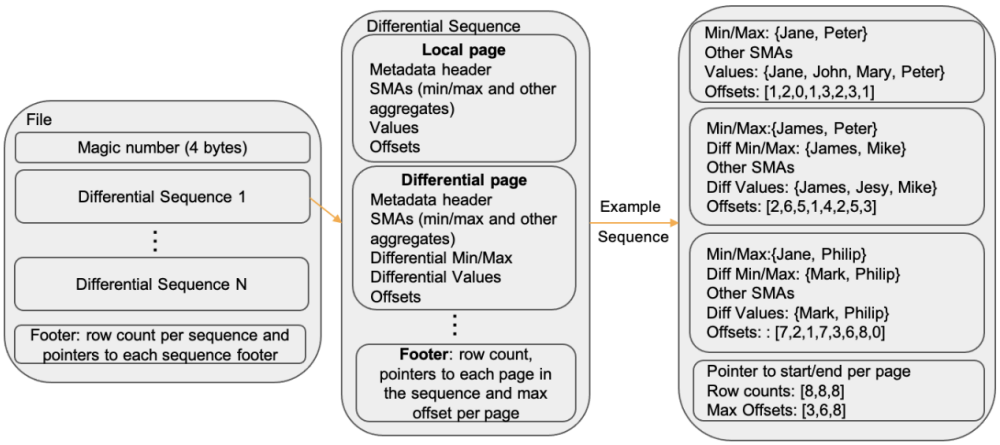


Figure 3.5: Adaptive dictionary encoding (image borrowed from [18])

3.3 String specific data structures

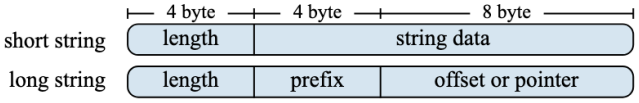


Figure 3.6: String header structure in Umbra (image borrowed from [38])

3.3.1 Umbra-inspired string representation

In the paper [38], which describes TUM's high-performance database system Umbra, the authors describe their string representation. Strings in Umbra are represented using two separate entities: The fixed-sized metadata and the variable-length body. The fixed-sized metadata or header is a 16-byte representation of the string that can be stored like any other data type. The first 4 bytes are used to store the string length. Thus, systems that use such a string representation can only store strings up to 4 GiB. Furthermore, the next 4 bytes are used to store the first four characters of the string. If the overall string length is equal to or below twelve, the rest of the string is stored in the remaining 8 bytes. However, if it is longer than twelve, a pointer is stored in the next 8 bytes, indicating where to find the entire string. The layout of the header is shown in Figure 3.6. This string representation does not include a capacity part in its layout, unlike std::string in C++, for example. To this end, it saves 8 bytes per string, but it also means that it is not possible to mutate or extend the string directly. Nevertheless, this representation suits the database use case very well. In databases, data is rarely modified in place, yet is read many times. This representation has already been adopted by major analytical data systems such as DuckDB[27], Polars[51], Facebook's Velox engine[50], and Apache Arrow[5]. In the following, we discuss the main benefits and key ideas behind its specific design.

Short-string optimization

Many strings in real-world datasets are short strings [58; 54]. Therefore, it is crucial to optimize for them whenever possible. The inlining property of Umbra strings greatly helps in this case, as short strings are stored in-place and no pointer dereference is required. Furthermore, it is more memory-efficient to store the rest of the string inlined rather than storing a 64-bit pointer.

Early-exit optimization

Traditionally, checking whether two strings are equal requires a memcmp operation, which may need to examine most of the string. With the Umbra-inspired strings, it is possible to first compare the length and the prefix before opting for the full string comparison. The majority of the strings in databases are typically not equal in length and do not share the same prefix. Thus, by utilizing this early-exit optimization, string operations can be performed much faster. Furthermore, we can avoid pointer dereferencing as much as possible with this property.

Copy in registers

The main reason behind this string representation being 16 bytes is that it is possible to pass this struct data type to function calls in two registers instead of passing it on the stack. According to CedarDB's blog post [1], it only takes four instructions to pass such a data type in the function call if it is 16 bytes, as compared to 37 instructions for the larger `std::string`.

3.3.2 String hash table

The paper "SAHA: A String Adaptive Hash Table for Analytical Databases" [58] proposes a highly specialized hash table design tailored for efficient string processing in analytical database systems. The authors argue that traditional state-of-the-art hash tables are ill-suited for string-heavy workloads because they ignore critical string characteristics, such as length. To address this, SAHA introduces a multi-layered hash table architecture. A highly optimized length-aware dispatcher is used to route incoming keys to one of five sub-tables based on their string length without causing regression. Figure 3.7 depicts the main constructs of SAHA.

Four of these sub-tables (S_0–S_3) are optimized for short strings, inlining up to 2, 8, 16, and 24 bytes of the string directly inside the hash table slot. This design eliminates the need for pointer dereferencing, significantly reducing cache misses. Hash table $S0$ is an array lookup table that supports up to 65536 values without storing the key. S_1–S_3 hash tables use a linear probing hash table. For string values longer than 24 bytes, hash table L is utilized, which stores the full hash value along with the pointer. This hash value serves as a salt that is first compared during lookup, reducing the likelihood of unnecessary full string comparisons by filtering out non-matching candidates early. Many other optimizations, such as hashing strings in batches and utilizing a specialized memory layout, are used to further enhance its suitability for analytical database systems. They

found that their proposed hash table outperformed other state-of-the-art hash tables by a factor of 1 to 5 based on the workload. This work has been merged into the analytical DBMS ClickHouse and is in full production [46].

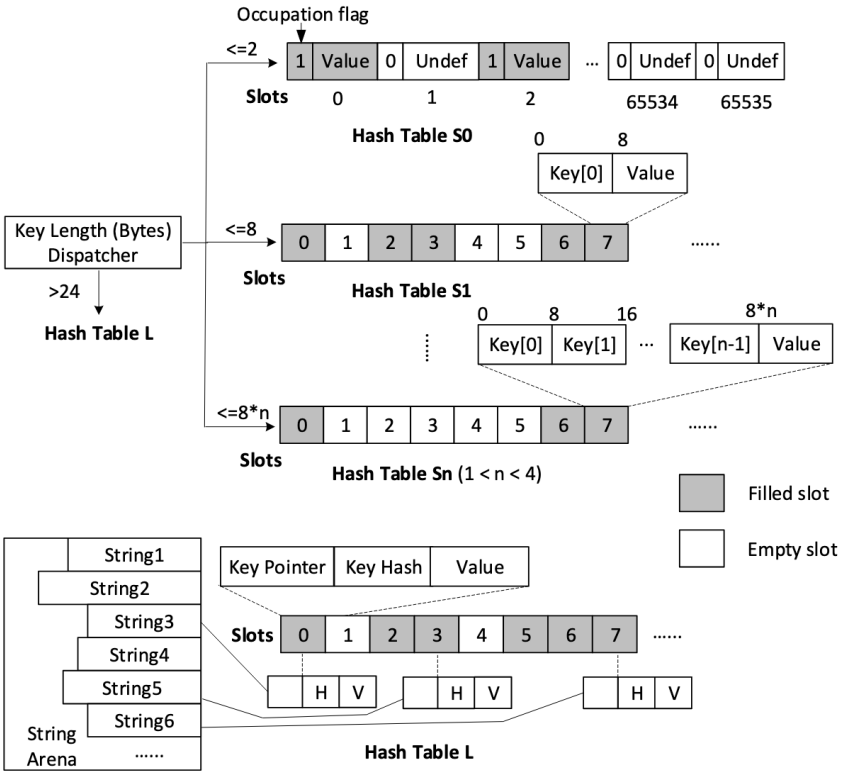


Figure 3.7: SAHA architecture containing the dispatcher and multi-layered hash tables (image borrowed from [58])

3.4 Summary

So far, we have discussed various database techniques and data structures that directly and indirectly aid in improving string processing in analytical workloads. Table 3.1 summarizes our survey by highlighting the benefits of each discussed topic and analyzing their adoption in widely used data management systems.

Method	Benefit	Adoption status
Compressed execution	Greatly beneficial for analytical workloads as large volumes of data need to be processed. Delaying decompression and executing on compressed data increases performance noticeably, especially for string values	It is widely used in various systems such as DuckDB, C-Store/Vertica, SAP-HANA, DB2 BLU, etc. However, it does require building the entire database architecture around the idea and is not easy to integrate into already existing engines without major overhauls. Its ultimate potential is held back by the code explosion problem
Global Dictionary	They enable consistent encoding for entire columns, which is particularly powerful for join-heavy and group-by-heavy workloads. Its order-preserving variant enables the efficient execution of range queries.	They are used in some in-memory databases, such as SAP HANA. However, they need to always remain in memory, which is highly undesirable for systems that require processing larger than main memory workloads. Furthermore, maintaining them in workloads that are ever-increasing in terms of domain cardinality is a major challenge. Due to these problems, their wide usage has been precluded.
Local Dictionary	They are mainly used as a feature in the storage to get better compression ratios. They also enable faster data scans.	They are widely used in open file formats, such as ORC and Parquet, as well as in database files like DuckDB. Their use in query execution is extremely limited since they are applied per block and do not enable column-wide encoding.
Umbra string representation	Novel string representation highly suitable for database strings during query execution. Its benefits for short strings and early-exit optimization greatly helps string processing.	It has been adopted widely for string representation in modern analytical database systems. It provides many benefits for execution engines on top of being easy to integrate into already existing engines.
String adaptive hash table	It provides a flexible and highly specialized hash table design for strings.	It is only implemented in Clickhouse. The added complexity of different tiers of hash tables makes integration in engines rather troublesome.

Table 3.1: Surveyed techniques' benefits and their adoption status.

4

Unified String Dictionary

To address the challenging problem of processing strings in DuckDB, we adapted and integrated USSR [23] into this analytical database system. We refer to this new data structure as the Unified String Dictionary (USD). There are features and requirements specific to DuckDB that make a one-to-one adoption impossible. Different string representations and the restriction on using global memory are the main challenges posed by the new environment.

This chapter provides a comprehensive overview of the implemented data structure, covering its benefits, core components, specific implementation details, and the various designs considered for integration into DuckDB's execution engine. It should be noted that all references to DuckDB's implementations are based on the latest features in the main branch, which are up-to-date as of July 8th, 2025 [1].

The remainder of this chapter is organized as follows. Section 4.1 provides a detailed overview of the USD data structure, including its memory layout, the underlying hash table design, and the mechanisms used to support concurrent insertions. Section 4.2 outlines the advantages of using USD, such as faster hashing and equality checks, reduced memory usage, and minimized copying for strings. Section 4.3 discusses the various design alternatives considered for inserting strings into USD and justifies the introduction of a new operator. Section 4.4 describes the implementation of this operator, and Section 4.5 explains the optimizer rule responsible for inserting the operator into query plans. Section 4.6 evaluates two different approaches for determining whether a string resides in USD, which is essential to fully leverage its benefits. Section 4.7 examines the lifecycle of USD, detailing when it is instantiated and destroyed during query execution. Finally, Section

[1] `https://github.com/duckdb/duckdb/tree/223ff0a7dba7900039d5910a009247ef097fff3c`

4.8 describes the engine-level modifications required to eliminate unnecessary string copies once data has been inserted into USD.

4.1 USD core data structure

4.1.1 Main components

Figure 4.1 depicts a high-level overview of the implemented data structure. Similar to USSR implementation, there are two main components that form the core of this data structure. In the following, we briefly describe them as they pertain to USD and detail how they differ from the original paper.

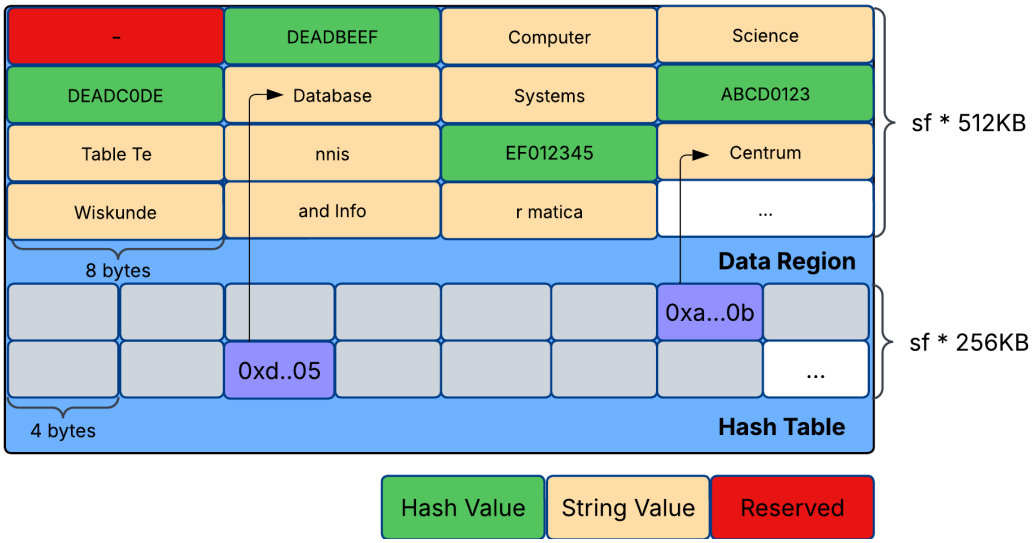


Figure 4.1: Unified String Dictionary components, including the linear probing hash table and the data region.

Data region

The data region contains the materialized strings alongside their pre-computed hash value. The overall layout of the data region consists of 8-byte chunks, and they will contain the string values. Although this memory layout could lead to fragmentation and inefficient memory use, it provides several important benefits.

First, we can use fewer bits to address a memory section in the data region. In other words, instead of using a pointer or an offset, we can simply use the slot index. By doing so, we can utilize the remaining bits in the hash table buckets to store a compact identifier derived from the string's hash value, referred to as a salt. Second, the hash value of each

string in USD is stored physically before the string itself. By doing so, we can directly access the hash value if we have access to the start of the string. Finally, by storing hash and string values in 8-byte-aligned chunks, we avoid crashes on platforms that require memory alignment for accesses.

In our design, the dictionary can be initialized with larger capacities to allow more string values to be inserted. This is achieved by passing a scale factor parameter when creating an instance of the USD. The scale factor, which must be a power of two, determines the static size of the data structure for the remaining duration of the current query. If no parameter is provided, the default size of 512 KB is used, similar to the original design.

Linear probing hash table

A linear probing hash table is used for fast lookup of strings within the data region. The hash table consists of 32-bit buckets. The total number of buckets is equal to the 8-byte slots available in the data region. Each 32-bit bucket is composed of two individual sections: the hash salt and a slot number within the data region. The slot number identifies a specific slot in the data region where the materialized string resides. The hash salt, on the other hand, is checked before resorting to a full string comparison during lookups.

Since we store the precomputed hash values in the data region, and in DuckDB the `hash_t` type is 64 bits, only half of the Data Region's slots contain string values. To this end, in the original USSR implementation, the hash table was always at a 50% load factor and below. This is further reinforced in our adaptation by inserting only non-inlined strings, due to the reliance on pointers for USD. Therefore, our USD hash table will always operate below a 33% load factor. The maximum load factor is mostly a worst-case scenario where all strings are between 12 and 16 characters in length. In real-world scenarios, the hash table operates in far less load factors, and the data region mostly contains string values.

Although we have not encountered use cases in our tests where the number of rejections for probing is noticeable, following the original implementation, we enforce a probing limit of sixteen to avoid long negative lookups. This will limit the maximum number of hash table slots to check in case of collisions.

4.1.2 Insertion

The main operation for the Unified String Dictionary is the insertion of strings. If the string does not already exist, the mechanism attempts to insert it as a new entry. However, if a string is already present in the dictionary, the insertion mechanism modifies the input

`string_t` to point to the existing materialized string. In the following, we discuss the step-by-step process of inserting strings into USD.

Preparation

The first operation done on the string is to use DuckDB's hash function for `string_t` on the input. There are two main reasons for this: Pre-computed hash values for strings are stored alongside the strings themselves in the data structure. Furthermore, we need to hash the input string to utilize our linear-probing hash table. The resulting hash value's first 32 bits are loaded in a variable, called hash prefix, to be used for probing the hash table. We apply a mask on the hash prefix to determine the hash table slot in which we should begin our probing. We refer to the remaining bits as the salt. Since comparing strings is an expensive operation, we would ideally like to be very confident that the value stored in the hash table and our input are equal before performing a full comparison. To this end, we will eventually store the salt in the hash table slot and compare it against the salt computed from the input string for future USD insert operations.

Probing

Once the required variables have been acquired from the input string, we begin probing the hash table to determine whether the string already exists or should be inserted. This is implemented as a for-loop that iterates up to a predefined probing limit. In each iteration, we compute a candidate bucket index based on the base index, which is derived from the hash prefix, and the current probe count. Depending on the value of the loaded hash table bucket, the following cases might occur:

Case 1: Empty Slot

If the hash table bucket value is zero, it indicates that the slot is unclaimed. In this case, we attempt to insert the string into USD. First, we check that the data region has sufficient space left. To achieve this, we can make use of the `current_empty_slot` variable, which keeps track of the next available slot in the data region for storing a new string. Using this variable, we can calculate the remaining space left in the data region and compare it against the required space by the input string and its hash value. If enough space is available, the insertion begins as follows: the string is copied into the next available 8-byte-aligned location in the data region, as indicated by `current_empty_slot`, and a null terminator (`'\0'`) is appended to mark the end of the string. The precomputed string hash is stored in the 8-byte chunk immediately before the string itself. Furthermore, a new

hash table bucket value is constructed, encoding both the hash salt and the data region slot index. This value is written into the hash table bucket to finalize the insertion. Finally, `current_empty_slot` is incremented by the number of 8-byte chunks used.

At this point, the input string has been materialized and can be referenced directly. We update the input `string_t` to point to the newly inserted string.

Case 2: Occupied Slot

If the loaded bucket value is non-zero, the slot is already claimed. This could mean either that it already holds a fully materialized string, or that another thread is attempting to insert a string into the same slot. Since handling concurrent insertions involves additional complexity, we postpone that discussion to the next section and, for now, assume that the slot holds a fully inserted string.

Under this assumption, we proceed by comparing the salt stored in the bucket with the salt of our input string. If the salts match, we extract the slot index encoded in the bucket and use it to locate the candidate string in the data region. A full string comparison is then performed to verify equality of the input string and the materialized entry in the dictionary. If a match is found, we update the input `string_t` using the `SetPointer` function to reference the existing materialized string.

If the salts do not match or the strings differ, we continue the probing sequence.

4.1.3 Concurrency control

For the sake of simplicity, the previous sections avoided the concurrency problems raised from having a dictionary where multiple threads can insert into simultaneously. Yet, it is crucial to discuss how USD functions in multi-threaded execution. Insertion is the primary operation that requires careful use of concurrency control methods to prevent data corruption or loss. This is because both the hash table and the data region need to be modified. The number of threads, average string length, and unique cardinality all play important roles in determining the performance penalty incurred. On the other hand, if the input string already exists in USD, the operation is a simple lookup up which consists of a series of read operations, and concurrency control can be avoided. In the following, we discuss possible concurrency control designs to allow for efficient insertion into USD.

Locking approach

A straightforward way to implement concurrency control in USD is to use locks. In C++, this can be done using `std::mutex` in combination with `std::lock_guard`, offering a

simple and safe mechanism. However, locks can block other threads from making progress and quickly become a performance bottleneck. To minimize contention, it is crucial to make locking as fine-grained as possible.

As discussed earlier, the primary point of contention in multi-threaded access to USD is during the insertion process. This operation involves modifying shared data structures and consists of the following steps: (1) Constructing and updating the hash table bucket value. (2) Incrementing the `current_empty_slot` counter to reserve space. (3) Copying the string and its hash into the data region.

All of these operations must be performed atomically, as they each modify the internal state of the data structure. In our implementation, we use a single `mutex` to guard the entire insertion sequence, including modifications to the hash table, the data region, and the `current_empty_slot` counter.

An important implementation detail is the need to suppress `ThreadSanitizer` [2]warnings for this data structure. Without suppression, DuckDB's continuous integration (CI) pipeline would flag normal reads during the probing phase as potential data races. This is because another thread might be concurrently modifying the same hash table bucket. Although our logic guarantees correctness by rechecking the bucket after acquiring the lock, ThreadSanitizer cannot detect this pattern. As a result, it flags the reads as potential data races unless we use slower atomic operations or explicitly suppress the warnings.

Lock-free approach

Our first implementation used locks to maintain concurrency control. However, during testing, we observed that when both the cardinality of the data and the average string length are high, the locking strategy incurs significant performance overhead. This is primarily because multiple threads are forced to wait behind long string-copying operations. To address this issue, we began exploring lock-free alternatives. The remainder of this section explains how we implemented a lock-free version of the USD.

A lock-free data structure avoids traditional synchronization primitives like mutexes, spinlocks, or semaphores. Instead, it relies on atomic operations such as compare-and-swap (CAS) and fetch-and-add to coordinate between threads while maintaining correctness. Our goal is to apply these atomic primitives to key variables within the data structure, thereby achieving even finer-grained concurrency control than possible with locks.

[2]`https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/.sanitizer-thread-suppressions.txt`

Insertion – Winner

In the first scenario, two threads attempt to insert a string and concurrently probe the same hash table bucket. They will compete for the right to update that bucket. We use the `compare_and_exchange_strong` primitive in C++, which atomically compares the current value of a memory location with an expected value and, if they match, replaces it with a new one. This ensures that only one thread "wins" the race to update the bucket. Both threads expect the bucket to be zero. The winner updates the bucket with a value consisting of the string salt value and a sentinel slot value.

This sentinel marks the bucket as "dirtied" and indicates that insertion is in progress. We reserve slot number *one* in the data region as the sentinel, meaning that actual string insertions must begin from slot *two* onward. We avoid using zero as a sentinel value. This is because a string with a hash salt of zero could make the bucket appear empty and lead to data corruption.

Once the winner dirties the bucket, it must reserve the required number of slots in the data region. We use the `fetch_add` primitive to atomically increment the `current_empty_slot` counter. This ensures that no other thread can claim the same region and that the memory remains consistent.

A capacity check is then performed to verify that enough space remains in the data region. If the check fails, the counter is rolled back to its previous value. This rollback is safe, as only one thread can increment the atomic counter past the region's end.

If space is available, the winner proceeds to copy the input string and its hash value into the reserved memory location. Once the copying is complete, the thread finalizes the bucket by replacing the sentinel value with the actual slot number used during insertion.

Insertion – Loser

The thread that loses the CAS operation will see that the bucket has been dirtied by another thread. It compares the salt stored in the bucket with its own `string_hash_salt`. If the salts do not match, the thread simply continues probing. However, if they match, there is a high chance that the same string is being inserted. In this case, the loser enters a wait loop until the dirty bucket is finalized. Once finalized, it performs a full equality check and, if the strings match, updates its pointer to reference the materialized string.

Insertion – Dirtied

If a thread encounters a dirtied bucket during probing, it performs the same salt check as

described above. Depending on the result, it either waits for the bucket to become finalized or continues probing.

Insertion – Finalized

If a finalized bucket is encountered during probing, the thread performs a full string comparison to determine whether the input matches the materialized value.

Overall, we found this lock-free approach to be significantly more performant. By removing the need for locks that hinder all participant threads, concurrency control becomes even more fine-grained. Now, only threads that are operating on the same hash table bucket need to wait, while other threads can continue execution.

4.1.4 Candidate strings

There are costs with regard to inserting strings in USD, such as hashing the input values to be utilized by the linear-probing hash table or paying for concurrency control. Therefore, it is reasonable to only put strings in the data structure that would benefit from being there, and the upfront cost of insertion is returned by the benefits provided later on. The following includes the possible candidates:

- Dictionary vectors: If the dictionary vectors are emitted from storage, we can be reasonably sure that it has already been determined that many duplicate values exist in this column. Thus, strings in the dictionary-encoded blocks are prime candidates to be put into our data structure.
- Constant vectors: Constant vectors are also valid candidates for insertion into USD. Since it is guaranteed to have duplicated values for the entire size of the vector. It should be noted that DuckDB does not emit constant vectors of strings from storage. However, there could be other use cases that will generate such vectors.
- Flat vectors: In most cases, if flat vectors are emitted from the storage, it signals that the column is not compressible and, thus, not a suitable candidate for USD. However, in the case of a join operation on primary and foreign keys, the primary keys might be a good candidate for insertion. This is because it is possible that these values are repeating on the probe side.
- Query Constants: Another prime candidate for insertion into USD are the query constants used as filter predicates. The goal is to simply accelerate the exact-match predicates.

In our implementation, we only consider strings longer than 12 characters, since they only store a pointer to the full string. We rely on the pointer to determine whether a string is backed by the USD or simply stored on the heap. The details of how we encode this information in the pointer will be discussed in later sections.

4.2 USD accelerated string processing

In the previous section, we described the internals of USD. In the following, we explain the benefits provided by the per-query dictionary and how queries can be executed faster when strings are stored in the USD.

4.2.1 Faster hashing

Following the design of the original paper, USD also stores the hash value alongside the materialized string right before it. This careful layout of the data enables accessing hash values for long, variable-length strings in constant time. It only requires the following instructions: (1) Check if the string resides in USD. (2) Perform a pointer arithmetic to obtain the location of the stored hash value, which is exactly 8 bytes before the start of the string. (3) Load the hash value directly.

The simplified logic of the hash function for *string_t* is shown in Algorithm 1.

Algorithm 1 Simplified string_t hashing logic in DuckDB with the addition of USD

```
1: function Hash(string_t input)
2:     if IsInlined(input) then
3:         return ComputeInlineHash(input)
4:     else if IsInUSD(input) then                ▷ (1) Check if backed by USD
5:         return LoadPrecomputedHash(input)
6:     else
7:         return ComputeNonInlinedHash(input)

8: function LoadPrecomputedHash(string_t input)
9:     string_ptr ← GetPointer(input)
10:    hash_ptr ← string_ptr −sizeof(hash_t)          ▷ (2) Pointer arithmetic
11:    return *hash_ptr                           ▷ (3) Directly load hash value
```

4.2.2 Faster equality checks

Equality checks on strings that reside in the USD are accelerated. Typically, equal strings in a column or across different columns, such as in a join operation, originate from different

sources. This is because of the way data is stored on different blocks on the disk and how DuckDB's scan operator brings them into the execution engine. Having different sources for equal strings will result in different memory backing them, and therefore, different `string_t` values for them. As a result, equality checks on equal strings will require a full `memcmp` over the entire string value, which can be expensive. However, if those strings are inserted into USD before the expensive equality checks happen, those strings will have the same representation throughout the rest of the query execution. Consequently, equality checks for them will be evaluated only on the `string_t` as described in Section 2.3.3. This would, in essence, enable execution on the compressed form of data that resides in the per-query dictionary.

4.2.3 Reduce memory pressure and avoid copying

Another major benefit of USD is that strings that reside in it only need to be allocated once during the entire query execution. Without such a data structure, duplicated strings might need to be allocated on the heap many times in different materializing operators.

For example, the intermediate result of a join operation will allocate memory on the heap for the payload columns, which might include many repeating values. Furthermore, full copies of strings need to be made, as the original values may reside on buffer pages that could become invalid during query execution.

However, if strings reside in USD, they will remain there for the entire duration of the query's execution. Therefore, making full copies of strings in the materializing operators can be avoided by simply referring back to the already materialized strings in the dictionary.

4.3 Integration design

In the previous sections, we discussed the internals of USD and outlined its potential benefits. In this section, we focus on the various design decisions regarding where to intercept the flow of strings emitted from storage and insert them into the USD. The general goal is to make sure that strings are inserted into the USD before they reach any *target operators*.

4.3.1 Target operator

We define target operators as those in the query plan that interact with strings in ways that inserting them into the Unified String Dictionary (USD) could result in performance gains. The following is the set of target operators we have identified, along with the rationale for

each: `LOGICAL_AGGREGATE_GROUP_BY`, `LOGICAL_DISTINCT`, and `COMPARISON_JOIN` perform hash and equality checks on strings. These materializing operators may also incur significant memory pressure due to large string payloads. Finally, `LOGICAL_FILTER` may evaluate string equality conditions but does not materialize its input.

4.3.2 Proof of concept integration

The first attempt at integrating USD in DuckDB utilized a singleton approach. A singleton is a class that has a private constructor. This way, it cannot be created by other methods. Instead, it provides a static GetInstance API to acquire the object. When it is called by other methods, it first checks if the object exists or not. If not, it will initialize the static object. This way, it is ensured that only one instance of the class is ever created. A singleton class is essentially a global object that any other methods can access by calling the GetInstance function. This approach allowed us to bypass DuckDB interfaces and integrate USD as quickly as possible. We used it to build a proof-of-concept to verify that DuckDB is able to benefit from USD. However, allocating global memory in an embedded database such as DuckDB is not possible, as it will lead to symbol conflicts. To this end, we explored other ways to properly integrate it with the system.

4.3.3 Design #1: Insert at individual operators

The first approach inserts strings into the USD at the level of individual target operators. Each operator's logic is modified to enforce additional checks and insertion routines for the valuable incoming strings.

The main advantage of this design is that it delays string insertion as long as possible, so that only strings actually used by the operator are inserted. This minimizes unnecessary overhead.

However, the major downside is that it requires duplicating similar logic across multiple operators, which reduces maintainability and increases the possibility of inconsistencies.

4.3.4 Design #2: Insert at storage layer

The second approach is the other side of the spectrum and attempts to insert strings at the storage level. One major issue with this approach is that we might be inserting strings in the USD that do not reach the target operators. Furthermore, this approach would have the same problem of code modification as the previous approach. This is because valuable strings could be flat, dictionary-encoded, or constant, all of which are emitted

from different interfaces and abstractions. Moreover, this approach is limited in its support for different data sources, as it only works with DuckDB-native files and does not extend to external formats such as Parquet.

4.3.5 Design #3: USD insertion operator

The other approach would be to create a new operator with the sole purpose of inserting strings into USD. It can be placed anywhere in the query plan and will control the flow of strings into USD.

The new operator design is highly inspired by the Compressed Materialization optimization in DuckDB (Section 2.3.4), as that also inserts logical projection operators into the query plan.

There is one important consideration with this design with regard to dictionary vectors. As mentioned in Section 2, dictionary encoding is applied on individual blocks within a column. Thus, the operators could receive multiple dictionary-encoded vectors that use the same underlying dictionary to encode their values.

It is of utmost importance to only insert the dictionary values once for each dictionary. To this end, we need to keep track of each dictionary's unique ID. This way, we can insert a dictionary upon encountering its unique ID for the first time.

Due to how parallelism works in DuckDB, we do not need to maintain a set of already seen dictionary IDs and can be sure that we will never encounter the previous dictionary again. Full details of how this can be achieved is discussed in the next section.

4.4 USD insertion operator

We chose to add a new operator to DuckDB for our per-query dictionary. This operator is responsible for handling string insertions into the USD during query execution and integrates directly into DuckDB's query pipeline.

Each operator in DuckDB consists of two variations: a logical operator and a corresponding physical operator. The logical operator is part of the logical query plan and is later used by the `PhysicalPlanGenerator`[3] to produce the corresponding physical plan. Therefore, our new operator requires two main components: `Logical_USD_Insertion` and `Physical_USD_Insertion`

[3] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/execution/physical_plan_generator.hpp`

4.4.1 Logical operator

`Logical_USD_Insertion` inherits from the `LogicalOperator`[4] class, which serves as the base class for all logical operators in DuckDB. Each logical operator is associated with a unique `LogicalOperatorType`[5], which identifies the kind of logical operation it represents. Hence, we introduced a new entry, `LogicalOperatorType::LOGICAL_USD_INSERTION`, to represent our new custom operator.

Attributes specific to the new logical operator are `insert_to_usd` and `insert_flat_vectors`. The former indicates which incoming columns should be inserted into the USD by the operator. The latter is a boolean flag that is set when flat vector values are to be inserted into the dictionary. The values of these attributes are determined according to the query being executed.

4.4.2 Physical operator

Just like the logical operator, our new physical operator inherits from DuckDB's `PhysicalOperator` base class.[6] Additionally, a corresponding entry is added to the `PhysicalOperatorType` enum class[7] to register our operator within DuckDB's execution engine.

Since the USD insertion operator is designed as an intermediate, parallel operator in the pipeline, we configure its execution properties as follows:

- `ParallelOperator()` returns `true`, signaling to the execution engine that the operator can be parallelized.
- `IsSource()` and `IsSink()` both return `false`, as this operator does not generate input nor is a sink operator.

During physical plan generation, the two configuration variables discussed in the previous section, `insert_to_usd` and `insert_flat_vectors`, are transferred from the logical operator to the physical operator.

[4] https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/planner/logical_operator.hpp

[5] https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/enums/logical_operator_type.hpp

[6] https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/execution/physical_operator.hpp

[7] https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/enums/physical_operator_type.hpp

Execution interface

Our physical operator overrides three main methods from the `PhysicalOperator` interface: `GetOperatorState`, `GetGlobalOperatorState`, and `Execute`. These are used by the `PipelineExecutor` class[8] for execution.

- `GetOperatorState()` returns a thread-local execution state. We implement this using a `USDInsertionState` class, inheriting from `OperatorState`[9]. It stores thread-specific data such as dictionary IDs and insertion outcomes.
- `GetGlobalOperatorState()` returns a global context shared across all threads, implemented as a new class `USDInsertionGState` inheriting from `GlobalOperatorState`. It tracks global variables such as locks, statistics, and shared execution flags.
- `Execute()` is called on every incoming `DataChunk`. It performs the USD insertion logic and returns `OperatorResultType::NEED_MORE_INPUT`[10] to indicate that the operator is ready for more data.

Insertion logic

Each incoming `DataChunk` is processed column-by-column. For each column, we first check if its data type is `VARCHAR`, and then verify whether it is marked in `insert_to_usd`. If these conditions hold, we insert the column's string data into the USD using different functions depending on its vector type, such as flat, constant, or dictionary. Flat vector insertion is skipped unless `insert_flat_vectors` is enabled.

Handling dictionary vectors

Dictionary-encoded vectors require extra care. This type of vector is mainly comprised of a `SelectionVector` containing the indices that point into a per-block dictionary. The actual string values reside in an auxiliary flat vector, accessible via the `DictionaryVector::Child()` API.

To avoid repeated insertions, we track which per-block dictionaries have already been inserted using the dictionary IDs associated with each dictionary-encoded vector. When a new dictionary ID is observed, we discard the previously stored ID and track the new one.

[8] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/parallel/pipeline_executor.hpp`

[9] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/execution/physical_operator_states.hpp`

[10] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/enums/operator_result_type.hpp`

This is consistent with how DuckDB processes dictionary vectors in `hash_aggregate` in its `TryAddCompressedGroups` code path as discussed in Section 2.3.4.

Control insertion flow

The operator also allows us to monitor and control the flow of strings into USD. Each string insertion into USD returns a status code indicating the result of the insertion operation. By collecting these results, we can (1) track how many values are accepted versus rejected, (2) estimate when the USD is becoming full, and (3) detect high-cardinality columns that quickly exhaust the dictionary's limited capacity. For instance, our experiments show that USD performs poorly with high-cardinality columns, as they quickly exhaust the available USD capacity. To stop this, we use a simple heuristic based on the average growth rate. If high cardinality is detected after observing 10 per-block dictionaries, the operator halts insertions for that column. Further static defensive constraints, such as maximum number of unique strings per columns and maximum number of unique values per dictionary block, are also used to avoid performance regression at all costs.

4.5 USD optimizer rule

In the previous sections, we detailed our reasoning for having a USD insertion operator. We believe the most suitable place to insert this operator into the query plan is within the DuckDB optimizer, as it has access to the entire logical query plan before any physical plans are generated. Beyond this, there are important practical reasons for implementing an optimizer rule. The USD may be assigned a noticeable amount of memory, which can be undesirable under high memory pressure, such as in embedded systems. At the same time, not all queries benefit from using the per-query dictionary. For example, if a query only scans the data and applies a filter, the overhead introduced by hashing, extra equality checks, and concurrency control may outweigh any potential benefits. Therefore, the goal is to activate this data structure only for queries that are likely to benefit from it.

In the following, we first discuss DuckDB's query optimizer and its important classes. Then, we delve into how we implemented the USD optimizer rule.

4.5.1 DuckDB optimizer

The DuckDB optimizer is invoked in the `CreatePreparedStatementInternal` function within the `ClientContext`, provided that the optimizer is enabled in the configuration. It takes the logical plan produced by the query planner as input and applies a set of

optimization rules to it. DuckDB includes many built-in rules for different optimization tasks, such as filter pushdown and join reordering. Some rules modify the query plan by inserting or replacing operators, while others reorder existing operators to improve performance.

Expressions
An expression is an abstract representation of a computation in a query. It is part of the logical query plan and can later be transformed into a physical operator. Each logical operator contains a set of expressions.

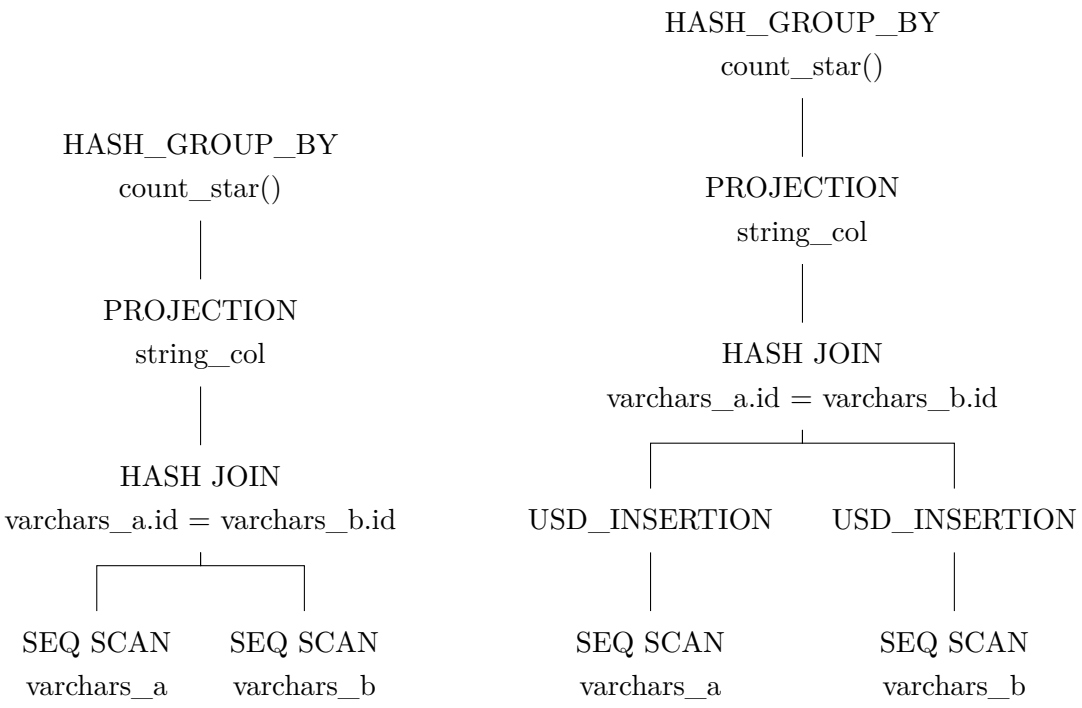
ColumnBinding
A column binding is a combination of a table index and a column index, used to uniquely identify a column within a query. Each `LogicalOperator` in the query plan implements the `GetColumnBinding` function, returning the set of column bindings related to its output columns.

ColumnBoundRefExpression
This expression type represents a `ColumnRef` that has been bound to an actual column from a table. The `ColumnBindingResolver` transforms this into a `BoundExpression`, which refers to an index in the physical `DataChunk` that is passed to physical operators in the execution engine.

4.5.2 Implementation

In this optimizer rule, we traverse the query plan in a depth-first search post-order manner. For each logical operator in the query during our traversal, we check if it is a target operator. To do this, we first check if the logical operator type is included in the set of target operators we discussed before in Section 4.3.1. Then, for each target operator, we check for additional conditions to determine if they actually operate on string values. For the group-by operator, we check if expressions used as the group-by columns are `BOUND_COLUMN_REF` and if their return type is VARCHAR. Similar logic applies to other target operators. For materializing operators such as join, we also check if the return types for payload columns include strings.

Once we have found a target operator, we create a new USD insertion operator and place it below the target operator. An example of a simple query plan before and after is depicted by Figure 4.2. Our goal is to add only one USD insertion operator between each



(a) Baseline Plan **(b)** Plan with USD Insertion

Figure 4.2: Simplified DuckDB query plans with and without the Unified String Dictionary. The query involves two materializing operators, both applied to string columns. Plan (b) will insert the strings into the per-query dictionary, affecting both join and aggregation operators.

leaf and the nearest target operator. The remaining logical operators on the path to the root will also benefit from strings in USD if they operate on them. This is because once strings are inserted into USD, they will remain there for the rest of the query execution.

To ensure that at most one USD insertion operator is placed between any leaf node and the root, the logical plan is rewritten while propagating a single Boolean flag along different paths. Each recursive call returns both the possibly modified subtree and this flag, which records whether a USD insertion operator already exists beneath the current node. After processing all children, the parent examines their flags: if any child reports `true`, the parent skips insertion to avoid duplication; if all children report `false`, the parent determines whether it is itself a target operator and continue to propagate the result through the rest of the call stack.

4.6 USD string recognition

This section discusses how DuckDB can fully leverage the benefits provided by USD. One of the core arguments of the original USSR paper and, in extension, USD, is that it is possible to integrate this feature in existing engines without major overhauls of the execution engine. It does this by not modifying the string representation. However, this has the downside that the system needs to recognize the strings that are backed by the per-query dictionary. In the remainder of this section, we will discuss two possible approaches that we implemented to achieve this. Both approaches require access to pointers, and therefore, only non-inlined strings are considered for USD insertion.

4.6.1 Aligned memory location

In the first approach, which is identical to how the USSR paper was implemented, strings are recognized by the memory area to which they point. In essence, if the strings are pointing into the memory location belonging to the data region of USD, we consider them to be USD-backed strings. To achieve this, we ensure the data region starts at a memory-aligned address. For example, if the data region is 512 KB, we choose a memory address that is 512 KB-aligned. Thus, the start of the region contains 19 zeros as the starting bits in its memory address.

By positioning the data region in this specific way, all the strings within the region can be recognized by examining their pointers. To achieve this, we required two variables: USD_MASK and USD_PREFIX. The USD_MASK will be applied to the string pointer in an AND operation, zeroing out the lower bits. USD_PREFIX is the aligned address where the data region is located. To recognize if a string resides in USD, we apply the USD_MASK to its pointer and then compare it against the USD_PREFIX. If equal, it means that the string undoubtedly is backed by USD and benefits such as faster hashing and single allocation can be applied to it.

Finding aligned memory location

The most important requirement for this approach is allocating a memory-aligned buffer for our data region. However, since DuckDB is built with C++11, it is not possible to allocate memory at a specific alignment with a specific size without relying on the operating system's API. This feature is somewhat supported by C++17 [11]. Therefore, we need to

[11] `https://en.cppreference.com/w/cpp/memory/c/aligned_alloc`

allocate extra memory and find our aligned region within the allocated buffer. We used the same tactic that was described in the USSR paper. We will allocate double the size of our data region. In this allocated buffer, we can always find an aligned address to store strings and their hash values, and also have enough memory to place the linear probing hash table either before or after the data region.

System modification

In terms of system modification, in every instance that the system may want to perform a hash operation or copy the string in whatever form, it needs access to the two variables discussed above, USD_MASK and USD_PREFIX. Another important note is that these variables cannot be allocated globally because DuckDB does not allow global memory as one of its core fundamentals. The specific reason can be attributed to the fact that multiple DuckDB instances can run in the same user space and process. By not allowing global memory, the system avoids symbol conflicts.

A suitable location to host these variables is the ClientContext, which contains key attributes related to the entire database. More importantly, every execution class and operator in DuckDB that may perform one of the actions that may benefit from USD-strings has access to ClientContext. However, even with the smart placement of the required variables in the ClientContext, many DuckDB interfaces have to be changed for this approach. This is because the core subsystems that the operators rely on for performing vectorized hash operations, materializing query intermediates in row format, and collecting the final results in columnar format do not have access to ClientContext. In our implementation, propagation of ClientContext through the engine required hundreds of interface changes, which is undesirable.

4.6.2 Pointer tagging

The second approach is to encode the fact that a string is backed by the Unified String Dictionary in the string representation itself. However, it is not possible to change the layout of `string_t` as increasing it would negate its benefits. Furthermore, if we wanted to use one bit in the length portion of the layout, it would halve the maximum length of strings that can be stored in DuckDB. This in itself would not be a significant issue, since currently strings of up to 4 GiB can be stored; however, it can cause compatibility problems. Finally, taking a bit away from the prefix would only limit the early-exit optimization of this string representation.

There is one more place where we can still borrow the one bit we need to recognize if a string is backed by USD or not, the upper bits in the pointer. In most architectures, only up to 48 bits out of 64 are used. This is because there are no systems available today that possess enough memory to require more than 48 bits to address their memory space. Therefore, we can simply set the highest bit in the pointer when strings are inserted into USD. At the moment of writing this thesis, there are no other use cases for the upper bits of pointers in `string_t` in DuckDB.

It should be noted that this approach would not work for Android devices. DuckDB also disables its salt usage in the upper bits of the pointers in its hash tables [21] when compiling for Android [12].

There are, however, different considerations regarding this change. Each specific CPU requires different properties for the pointers. For example, x86-64 processors require that pointers must be in their canonical form [56]. To avoid any problems caused by inconsistency in pointer values, the added tag to each USD-string pointer must be cleared before dereferencing. This is further complicated by the fact that the system may not always use the `string_t` API to modify the string attributes. Thus, all of these instances need to be modified to clear the tag by applying a mask value to the pointer, setting the upper 16 bits to zero. Furthermore, the `string_t` API [13] itself needs to be modified as described in the following:

- `GetTaggedPointer`: A new function is added to return the raw pointer in `string_t` without clearing the tag.
- Tag clearance: `GetPointer`, `GetDataWriteable`, `GetData`, and `Equals` are the functions that need to be modified to first remove the tag before returning the pointer value

4.7 USD lifetime

USD is a per-query data structure that is built at runtime with the valuable strings used in a query. This section details when it is created and when it should be destroyed.

[12] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/execution/ht_entry.hpp`

[13] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/string_type.hpp`

4.7.1 Construction

The decision to activate USD is up to DuckDB's optimizer. After the planning and binding section of query processing, the optimizer applies its rules to the query plan to further optimize the execution plan. We believe this is the perfect place to decide whether a query can benefit from USD. If the optimizer rule we discussed in Section 4.5 observes specific operators that operate on strings, it will activate USD. It does so by creating a USD object and placing it in the ClientContext. As this class is widely available throughout the system, particularly on the execution side, we found it to be the perfect location to host USD.

4.7.2 Destruction

USD will contain the materialized strings for an entire query. Therefore, to avoid use-after-free errors, it must remain alive until it can be assured that none of the strings within it are required anymore. We choose the moment when the query result is fully fetched by the user as the precise time to destroy USD and free the allocated memory to the system. DuckDB uses the QueryResult [14] class as the base class to represent the result of a query. When the user executes a query, the ClientContext is provided with a QueryResult to store the final result. After the query execution is finished, the ClientContext stores the result and is then destroyed. Since ClientContext hosts USD, we will move the USD class to the QueryResult. This can be done efficiently in C++ by using the move semantics (`std::move`). This movement of USD ensures that the string within USD is valid until the query result is fully retrieved by the user. USD is destroyed at the same time as the corresponding QueryResult class.

4.8 Preventing unnecessary copies

One of the main promises of USD is the zero-copy benefit it offers for strings. As a general rule, DuckDB copies the values backed by buffer pages when it needs to materialize them within internal data structures used for query processing. This is because it cannot simply rely on those pages being in memory for the entire duration of the execution, as the buffer manager might evict them to disk. However, if strings reside in USD, they are already materialized in memory and will remain there throughout the entire execution of the query. Therefore, there is no need to further make full copies of string values. In other words, the execution engine can get away with copying only the *string_t* header, which

[14] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/main/query_result.hpp`

contains pointers that refer within the USD. However, to enable the system to utilize this benefit, multiple changes are required. Primarily, we need to adjust the various DuckDB subsystems handling strings to treat non-inlined strings residing in USD as if they were inlined ones. By doing so, only *string_t* headers are copied and deep copies of the full string are avoided. We outline the necessary changes in the following.

4.8.1 ColumnDataCollection

According to the DuckDB documentation, `ColumnDataCollection` [15] represents data stored in columnar format, which is backed by the buffer manager subsystem of DuckDB. Many use cases require the data to be stored in columnar format during execution. For example, when DuckDB returns the result of a query, it uses the ColumnDataCollection to represent the output, as it is far more efficient than row storage.

Modifications:

There are specific code paths in the `column_data_collection.cpp` file which handles the complexities of the string type in DuckDB. One specific change that needs to be made in this subsystem is the handling of how strings are copied. In the `StringValueCopy`[16] function, some metadata along with the string that needs to be copied is provided to the function. If the string is inlined, the function simply returns the input as no extra modification is required. However, if the string is non-inlined, it will allocate memory on the heap and store the full string there. We have made changes to this function so that it also checks if the non-inlined string is backed by USD or not. If backed, it simply returns the input just like the inlined string code path. Similar changes are made to different parts of the same file to treat non-inlined strings in the same way as inlined strings to prevent extra copying and heap allocation.

4.8.2 TupleDataCollection

This class[17] represents the data stored in row format, which is backed by the buffer manager. Example use cases of this class include storing intermediate results in materializing operators, such as joins and aggregations. In these cases, it is much faster to access data in a row format rather than the columnar format [27].

[15] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/column/column_data_collection.hpp`

[16] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/common/types/column/column_data_collection.cpp`

[17] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/row/tuple_data_collection.hpp`

Modifications:
The changes to this subsystem are mainly applied to `tuple_data_scatter_gather.cpp`[18], which is used to store values, previously available in columnar format, in row major format and vice versa.

- `StringHeapSize`: This function calculates the heap size required to store a string value in the TupleDataCollection. If inlined, the string does not require any heap memory. Otherwise, it is allocated heap memory equal to its length. We made the changes so that USD strings are also treated like inlined strings.

- `TupleDataValueStore`: This function is used to store different values in the TupleDataCollection and is called during the scatter phase, where data in columnar format is stored in row-major format. The changes to this function allow USD strings to be copied by only storing the `string_t` header instead of the full string.

Although not related to `TupleDataCollection`, very similar logic for strings exists at the `ComputeStringEntrySizes` and `ScatterStringVector` in the `row_scatter.cpp` file. We made similar changes in the mentioned places as well.

4.8.3 Out-of-core execution

DuckDB can efficiently process workloads that produce larger-than-memory intermediates [27]. It does so by using the same page layout for both query intermediates and persistent data. This way, both can be efficiently spilled to disk. In this page layout, fixed-size rows are stored together in one page, while their corresponding variable-size rows are stored in another page. With regard to strings, the 16-byte header of `string_t` is stored in the fixed-size row. If the string is not inlined, the pointer residing in `string_t` points to the complete string in the variable-size page. During disk spillage, pages are taken to disk and brought back when required. However, they may be loaded back into memory in a different location. This results in the pointer in the fixed-size layout becoming invalid. DuckDB's workaround for this problem is to swap the pointers with offsets into the variable-size page. This operation is called pointer *swizzling*, and the reverse of it is called pointer *unswizzling*. Extra metadata is required to be maintained to compute the new pointers based on the offsets and the base address of the new page. Figure 4.3 illustrates the structure of DuckDB's page layout.

[18] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/common/types/row/tuple_data_scatter_gather.cpp`

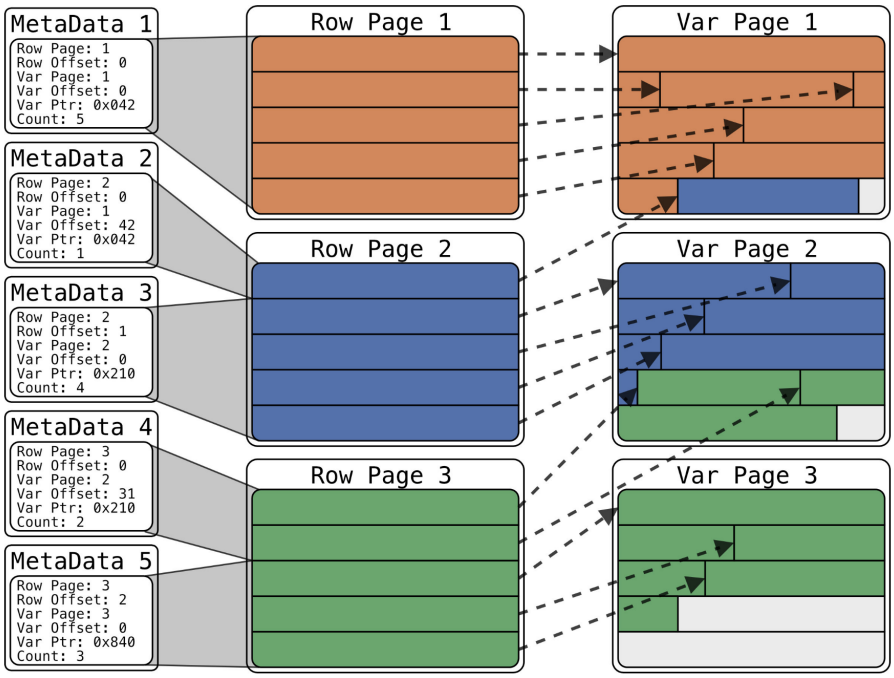


Figure 4.3: DuckDB's page layout for fixed-size rows and corresponding variable-size rows (Image borrowed from Kuiper et al. [27])

With the addition of USD to DuckDB's execution engine, changes are required to treat non-inlined strings as inlined strings, which do not require the operations discussed previously.

Modifications:

As the changes required are exactly like the ones mentioned for `ColumnDataCollection` and `TupleDataCollection` with regards to logic, we simply mention the main functions that were modified. These include `RecomputeHeapPointers` and `FindHeapPointers` in `TupleDataAllocator`[19], as well as `SwizzleColumns`, `UnswizzlePointers`, and `GatherVarchar` in `RowOperations`[20].

[19] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/types/row/tuple_data_allocator.hpp`

[20] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/include/duckdb/common/row_operations/row_operations.hpp`

5

Evaluation

5.1 Experimental setup

As DuckDB is mainly used by users with high-end laptops, we conducted our evaluations using a MacBook Air M2 with 16GB of memory. This machine features eight cores, divided equally between performance and efficiency cores. To measure performance, we opted to use end-to-end query execution times, since singling out individual performance factors in such a complex system is both difficult and unreliable. We compare DuckDB enhanced with USD and DuckDB's main branch against different workloads and report the results. Regarding the workloads, we used both standard benchmarks typically used for evaluating analytical database systems and synthetically generated datasets and queries.

5.2 Standard benchmarks

5.2.1 TPC-H

TPC-H [48] is a standardized decision-support benchmark developed by the Transaction Processing Performance Council (TPC) that models a business environment with complex queries. This benchmark focuses on measuring how efficiently a database can process large volumes of data through ad-hoc queries, aggregation, joins, and sorting operations. TPC-H is widely used in both academia and industry to compare the performance of different database engines under analytical workloads. In our experiments, we relied on DuckDB's TPC-H extension [1], which provides the complete suite of 22 read-only queries for TPC-H.

Results: The only query that could make use of the Unified String Dictionary in this benchmark is query number sixteen. This query features a triple column group by, and

[1] `https://duckdb.org/docs/stable/core_extensions/tpch.html`

one of these columns is `p_type`, which contains strings between 16 and 25 characters. Furthermore, it only has 150 unique values in its entirety. This makes it a perfect candidate for USD. The speed up for this query at different scaling factors is shown in Table 5.1. The other queries in the benchmark suite either did not contain operations on string columns that our optimizer rule would recognize, or the strings themselves were not qualified as candidates for USD.

Scale factor	DUCKDB_MAIN	DUCKDB_USD
1	0.0147	0.0132
10	0.0890	0.0708
30	0.251	0.201

Table 5.1: Performance results - Query 16 in TPC-H

It should be noted that queries 7, 8, and 19 in TPC-H apply filter predicates on potentially large strings that can be inserted into USD. However, our final USD implementation does not result in speed-ups for the mentioned queries since most of the filters are pushed down to the storage layer. In our earlier implementations (Section 4.3.2), the mentioned queries got faster by 10 to 15% depending on the scale factors. However, it is important to note that the achieved speed-up for filters pushed to storage should not be as noticeable if DuckDB applied the filter predicate on the unique values in the dictionary. In the current implementation, DuckDB resorts to UnifiedVectorFormat for ColumnData [2], potentially computing the same predicate evaluation for many duplicate values.

Other than the filters that are pushed down to storage, certain filters still remain in the query plan (query 7). These queries can still benefit from USD. Although the effect of USD is less noticeable due to the low number of values that are actually evaluated, particularly on smaller scale factors.

Finally, query 4 was one of the queries that got faster in the original USSR paper by about 30%. However, in DuckDB, Compressed Materialization is applied to the `o_orderpriority` columns, nullifying the benefits of USD.

TPC-H with string keys

As stated by the "Get Real" workload study [52], it is very common for real-world database tables to have strings as keys. Thus, in this experiment, we decided to change the integer keys in the nation and customer tables of TPC-H to Universally Unique Identifier (UUID)

[2] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/storage/table/column_segment.cpp#L294`

and convert them to strings. As a result, these tables will have 36 character strings as keys. We will then perform simple join queries on these modified tables. Our goal with this experiment is to test whether USD is also suitable for join operations on string columns.

Results: We can see from the performance results depicted in Table 5.2 that USD can also speed up queries containing join operations on string keys. In the two modified tables, the nation table only contains a total of 25 rows. Thus, the string key column is not compressed and is emitted as a flat Vector in DuckDB. However, due to our optimizer rule, we can use heuristics, such as the estimated cardinality of join columns, to detect join operations that can still benefit from USD. Therefore, we also insert the strings on the build side of the join, allowing for the full performance benefits of the Unified String Dictionary to be achieved.

Scale factor	DUCKDB_MAIN	DUCKDB_USD
1	0.025	0.020
10	0.072	0.060
30	0.203	0.160

Table 5.2: Performance results - Simple join query on modified TPC-H columns

5.2.2 TPC-DS

TPC-DS [37]builds on and extends the ideas of TPC-H, offering a more realistic and comprehensive benchmark for evaluating decision support systems. While TPC-H uses a relatively simple schema with a single fact table and eight tables overall, TPC-DS features a more complex data model. Furthermore, TPC-DS also includes 99 diverse SQL queries that are exceedingly more complicated than TPC-H queries, covering a broader range of scenarios.

Results: No queries in TPC-DS slowed down or sped up as a result of USD. This could be attributed to the queries themselves not using USD target operators or the data within the columns not meeting the requirements to gain considerable benefit from being inserted into USD.

5.2.3 IMDB

We also ran the IMDB join order benchmark used in [32]. The queries in this benchmark suit heavily involve joins on integer keys. There are string payloads in the join operations, yet

they are typically either from high domain cardinality or inlined. Overall, no performance gain or loss was observed in this benchmark.

5.2.4 ClickBench

Clickbench is yet another highly used benchmark for analytical database systems. Similar to IMDB, no performance gain or loss was seen in this benchmark. Other than a few specific queries that perform aggregation on high domain cardinality columns such as URL and Title, no other query will be recognized by the optimizer rule. We will discuss this benchmark in more detail in the next chapter.

5.2.5 Public BI Benchmark

Similar to the original paper, we decided to test our implementation on the CommonGovernment workbook, which is part of the Public BI Benchmark [15; 19]. This benchmark's queries and data are gathered from 46 of the biggest Tableau Public workbooks used in the "Get Real" workload study[52]. The CommonGovernment workbook specifically contains many strings, making it an excellent choice for our use case. The vast majority of the queries are simple group-by operations on string columns.

Results: All queries were run on their specified subset of the data. The vast majority of the queries contained single-column group-by, which is already covered by DuckDB's optimization as discussed in Section 2.3.4. However, there was considerable speedup regarding a few double-column group-by queries, as the results show in Table 5.3. The rest of the queries did not speed up or slow down.

Query number	DUCKDB_MAIN	DUCKDB_USD
2	0.133	0.112
3	0.048	0.039
21	0.085	0.078

Table 5.3: Performance results - CommonGovernment workbook

5.3 Synthetic micro-benchmarks

We also decided to make use of synthetically generated datasets. With these workloads, it is possible to control every aspect of the input, such as string length, alphabet set, domain cardinality of values within a column, and distribution of values. Our main goal with

these experiments is to limit-test the Unified String Dictionary with regard to different workloads. We generated the datasets using Python scripts [3].

In the following, we detail the various experiments we ran using synthetic datasets.

5.3.1 Variable length strings

In this experiment, our goal is to assess the performance result with regard to string length. To this end, we chose a small domain cardinality of 200 unique values per string column. Thus, every unique value can fit in USD without requiring a larger capacity. We aim to replicate a similar experiment done in the original USSR paper, but in DuckDB. The only variable in this experiment is the length of the strings in each workload. The microbenchmark query is a simple group-by operation on two string columns.

Results: As shown in Figure 5.1, USD will result in more performance gain as the string length increases. This can simply be explained by the fact that as strings get longer, typical operations such as applying a hash function, checking for equality, or copying them require more computation. Since USD accelerates the mentioned operations, it results in significant performance gains.

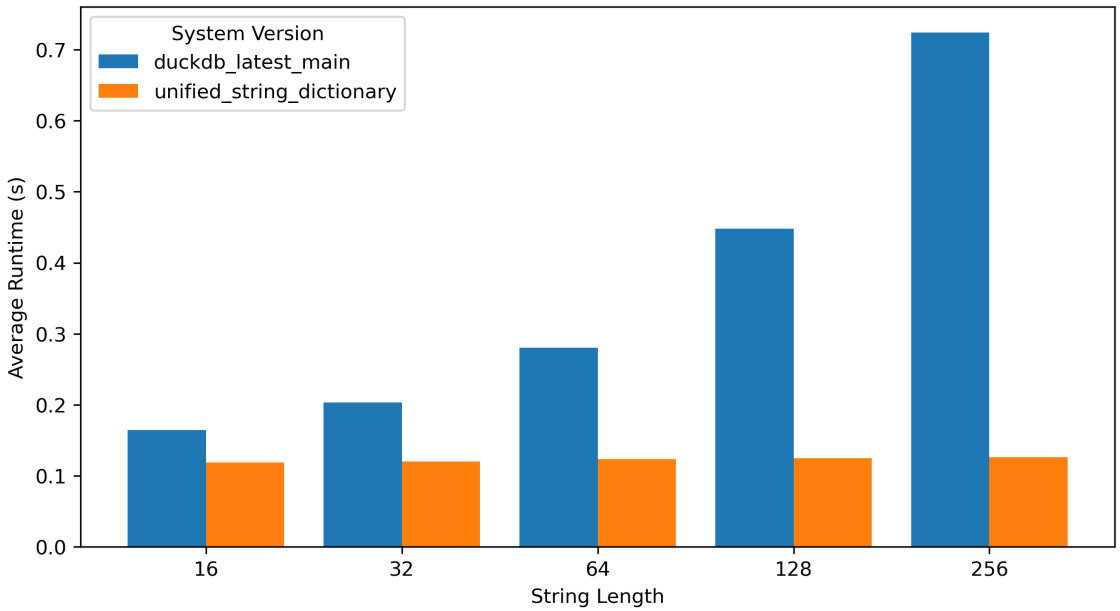


Figure 5.1: Performance results - Synthetic dataset, Variable length strings

[3] https://github.com/OmidAfroozeh/benchmarker

5.3.2 Low cardinality

Other than the effect of string length on the performance, it is important to limit-test USD with regard to different domain cardinalities. In this experiment, we aim to assess how this data structure deals with low domain cardinality. We define a column to be low cardinality if USD is able to fit all of its unique values without requiring its size to be increased beyond the typical L2 cache size. To this end, we run the same microbenchmark as the previous experiment on string columns containing 150, 300, 600, 1200, and 2400 unique values, respectively.

Results: Figure 5.2 showcases the results for low cardinality columns. As all unique values are able to fit easily within the small data region, this results in considerable performance gains.

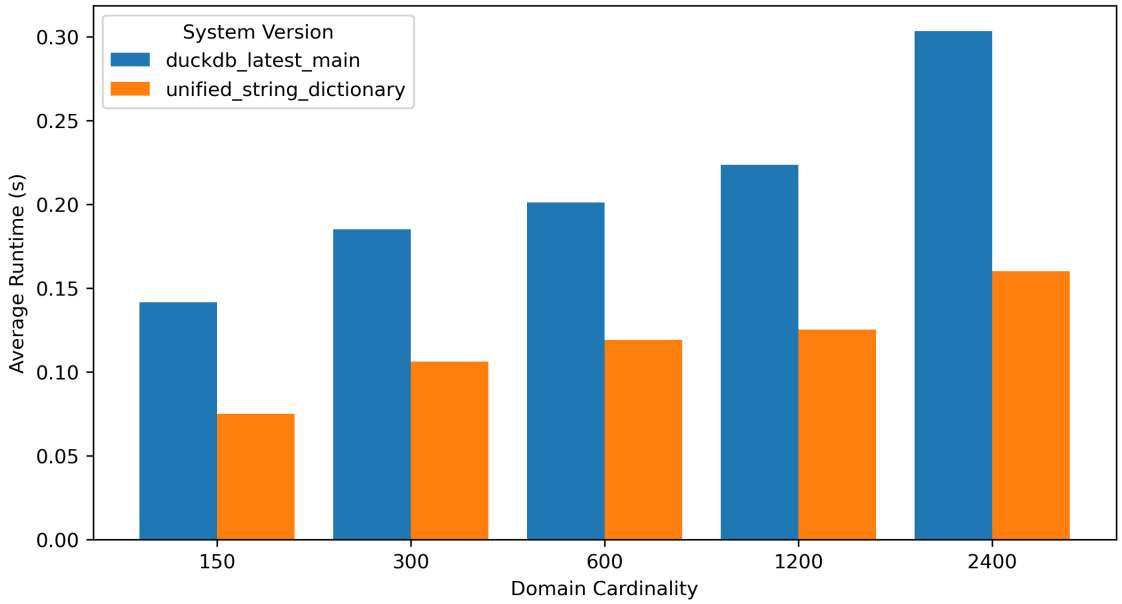


Figure 5.2: Performance results - Synthetic dataset, Low cardinality domain

5.3.3 High cardinality

Our main goal with this experiment is to test the data structure on much higher cardinalities. To this end, we decided to increase the data region size of USD to 256MB so that all unique values would be able to fit. Furthermore, the values in the column follow a Zipfian [60] distribution. The reason behind choosing this distribution is twofold: First, we wanted to replicate strings in real-world datasets and the English language. Second, a skewed distribution will cause the majority of the storage blocks to be dictionary encoded due

to the many repeating values. Therefore, they will be considered as candidate strings for insertion into USD. We also removed all constraints regarding the insertion of dictionary strings. Our primary goal is to determine the potential performance gain achievable with higher domain cardinality.

Results: In contrast to low cardinality columns, we observe that USD can be inconsistent with regard to high cardinality columns as depicted in Figure 5.3. From the results, we draw the conclusion that it is possible to gain some performance for high domain cardinality, but to a certain point. In general, when the data cardinality is high, our overall performance gain reduces since we need to invest more computation to insert every unique value in each per-block dictionary into USD. For example, in the synthetic dataset generated for 120,000 unique values, a single column is contained within 660 `Column Segments`. Each segment contains 15151 values on average. The per-block dictionary for each segment contains around 6561 unique values. During insertion, we add every single value in the per-block dictionaries to USD. Since the ratio between the number of unique values and encoded values is not high enough, it is unclear whether performance can be improved, considering the overhead of insertion and concurrency control. Furthermore, as random access to this large memory region is common throughout query execution, cache misses will accumulate and potentially result in performance regression.

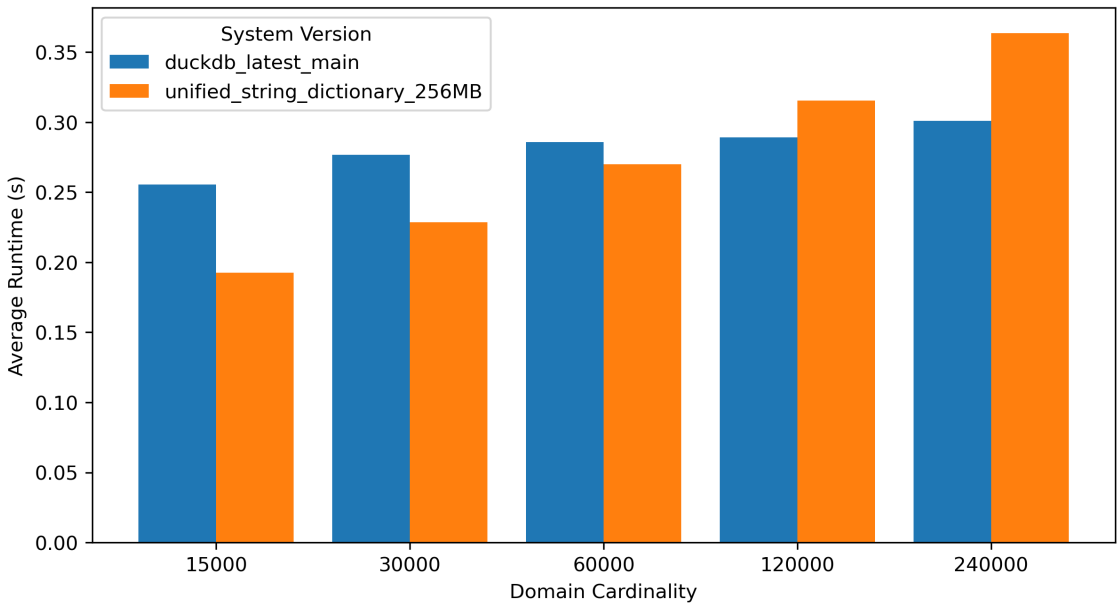


Figure 5.3: Performance results - Synthetic dataset, High cardinality domain

Luckily, it is very simple to avoid performance loss by simply avoiding high cardinality

columns as discussed in Section 4.4.

5.3.4 String payloads in materializing operators

In this experiment, we aim to show that the zero-copy benefit of the Unified String Dictionary affects materializing operators. To this end, we perform join operations on integer keys from tables that contain one string column each, and those are included in the select statement. The string payload columns in the tables are from a 100 domain cardinality and are 32 characters long.

Result: As depicted in Table 5.4, considerable performance gain is expected. Beyond this simple microbenchmark, the single-allocation benefit of USD is potentially beneficial for reducing the amount of data that is spilled to disk.

DUCKDB_MAIN	DUCKDB_USD
0.0147	0.0132

Table 5.4: Performance results - String payload columns

5.4 Results discussion

In the previous sections, we discussed evaluation results for USD on various standard and synthetic benchmarks. In the following, we aim to analyze and summarize the overall performance impact of USD for DuckDB.

Unified String Dictionary for DuckDB can result in performance gains if the following conditions regarding the workload are met. The query itself must contain materializing operators that are recognized by the optimizer rule. The string columns being operated on in the query need to be eligible to be inserted into USD. Finally, various statistics about the string values, such as the average length, domain, and overall cardinality, play important roles in how much performance is achieved. For workloads that satisfy the discussed conditions, the query can benefit from faster hashing and equality checks. Furthermore, duplicate values in the materializing operators can be allocated once, leading to less memory pressure and less copying.

An interesting observation is the difference in performance gain on various queries in standard benchmarks, such as TPC-H and the CommonGovernment workbook, when compared to the USSR paper results. DuckDB poses three main challenges that could potentially nullify the role of USD in achieving performance gains. These challenges are as follows:

Compressed Materialization effect

This optimization is one of the main features in DuckDB that can speed up operations on strings for materializing operators. For example, in queries 4 and 5 of TPC-H, the CM optimization inserts projection operators before and after the hash join and converts the strings into integers. USD would simply be an unnecessary addition to a query if CM is already applied to the string columns. However, Compressed Materialization is quite limited with regard to real-world data. This is because the existence of even a single outlier string value in the column with lengths longer than what is supported by CM would disable this optimization. Furthermore, since CM is dependent on the metadata stored in DuckDB's native file format, other data sources, such as Parquet, currently do not benefit from CM.

TryAddCompressedGroups effect

Many queries, both in TPC-H and the CommonGovernment workbook, involve a single-column group-by. For these queries, the majority of the benefits that were provided by USSR in Vectorwise do not translate to Unified String Dictionary in DuckDB. This is because of the TryAddCompressedGroups optimization in the hash aggregate operator, which renders USD completely ineffective.

DuckDB string representation

To make USD work, we had to make the design decision to only insert non-inlined strings. This design decision will lead to fewer possible candidates as many strings in real-world data are short [52]. This is perhaps the biggest change required in our adoption that results in less performance gain compared to the USSR in Vectorwise.

Despite the challenges and optimizations available in DuckDB, many potential workloads can be made faster, as demonstrated in our synthetic microbenchmarks.

6

Sampling-based approach

In the previous chapter, we discussed the internals of the Unified String Dictionary (USD) and explained how we implemented and integrated it into DuckDB's execution engine. Our evaluation of USD in DuckDB showed that USD could be a promising addition for efficient string processing, especially for low-cardinality columns. However, as observed in various experiments, USD performs poorly with high cardinality columns. As a result, we decided to disable this feature for these columns early in the execution pipeline.
In this chapter, we explore the idea of sampling the per-block dictionaries to investigate whether USD could still improve performance by inserting the most repeated values in high-cardinality columns. First, we motivate this idea by studying specific columns from the Clickbench dataset and their unique properties. Then, we present our hypothesis regarding leveraging frequent strings in datasets with considerable distribution skew. Next, we describe our sampling method and its integration into DuckDB's execution engine. Finally, we evaluate the results.

6.1 Clickbench study

The Clickbench benchmark [13] simulates workloads common in clickstream analysis, web analytics, machine-generated data, structured logs, and event data. It focuses on queries typically used in ad-hoc analytics and real-time dashboards. The dataset is based on real traffic from a major web analytics platform, anonymized while preserving key data distributions. The queries are designed to reflect realistic usage patterns, although not directly taken from production environments.

The Clickbench dataset contains two columns of particular interest: URL and Title.

These columns consist of long strings, and several Clickbench queries [1] (33, 34, 36, and 37) perform aggregations over them, making them suitable columns for USD. More interesting is the way DuckDB compresses these columns. For example, despite the URL column containing an extremely large number of unique values, most of the values are dictionary-encoded. We provide a simple data analysis below.

Clickbench's hits table contains 100 million tuples. The URL and Title columns have 18,342,019 and 9,425,424 unique values, respectively. Table 6.1 shows how DuckDB applies different compression methods to these columns, along with the percentage of values compressed by each.

Compression	URL	Title
FSST	28.8%	22.7%
DICT	69.6%	77.2%
Uncompressed	1.4%	0

Table 6.1: Compression statistics for URL and Title columns in Clickbench

From the frequency plot of the most repeated values in the URL column (Figure 6.1), we observe the highly skewed nature of the data. This skew explains why such a significant portion of the column is dictionary-encoded.

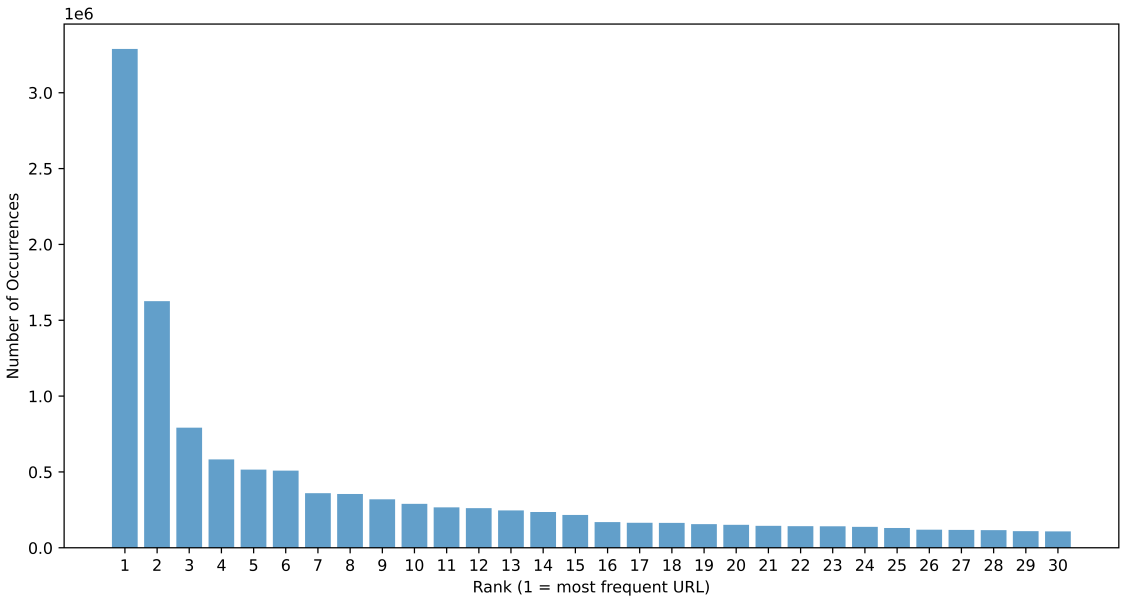


Figure 6.1: Frequency plot of most repeated URL values in Clickbench

[1] `https://github.com/ClickHouse/ClickBench/blob/main/mysql/queries.sql`

Hypothesis

Although the USD has limited capacity and cannot accommodate every unique value, we can still leverage it by focusing on the most valuable strings within a skewed dataset. By prioritizing these high-frequency strings, we can minimize insertion costs while getting the maximum performance benefits for the majority of the data. To achieve this, we identify the most frequently occurring values in each dictionary block and insert only these values into the USD. To efficiently determine these high-frequency strings, we employ a sampling-based approach. The following sections detail the implementation of this sampling strategy.

6.2 Sampling-enhanced USD

In this approach, our goal is to decide at run-time which unique strings within a per-block dictionary are the most valuable, thus generalizing to the entire dataset. To this end, we need to access the encoded values and build a frequency counter for each unique value. We considered three possible approaches, which we will explore and compare in the following.

6.2.1 Directly accessing column segments

This implementation aims to access the buffer-managed page that contains the entire dictionary-encoded block. This way, we have direct access to all the encoded values of each block. However, since we decided not to insert strings at the storage layer and opted for the USD insertion operator design, we do not have direct and easy access to this data. Fortunately, when DuckDB builds a dictionary vector from a dictionary-encoded segment, it encodes the pointer to this segment as the dictionary ID [2]. Therefore, we can use this indirection, and cast the dictionary ID to a pointer and access the page containing the segment. This is a rather limited approach, since the page might already be evicted by the time we try to access it. Furthermore, this approach is not supported by Parquet files as they use different interfaces. Additionally, we need to cover the extra cost of bit-unpacking, as all dictionary-encoded blocks in DuckDB are encoded.

6.2.2 Delaying vectors

As the previous approach has glaring issues, we decided to utilize the USD insertion operator. The idea is to delay vector progress in the query plan until a certain number of vectors have been seen. Then, we can perform our sampling and insert the most repeated

[2] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/storage/compression/dictionary/decompression.cpp`

strings within each dictionary. The main reason we need to delay the vectors is to prevent them from being materialized in the hash tables that some materializing operators build. If strings are materialized without being inserted into USD first, we lose the performance benefit of faster equality checks, since equal strings will have different memory backing them. Unfortunately, DuckDB's execution engine does not support such *delaying* operators. Therefore, we need to make copies of the vectors, which might require flattening them as currently implemented in the `DataChunk::Copy` function [3].

6.2.3 Streaming model

To address the limitations of previous approaches, we implemented a sampling method based on the streaming model. In this model, we make sampling decisions based solely on the incoming vectors emitted directly from storage. Unlike previous methods, we do not attempt to delay or buffer vectors; instead, we continue to evaluate the frequency of strings as they arrive.

We start by analyzing an initial sample of 2048 values. As more vectors arrive, we incrementally update the frequency counts for each encountered string. This process continues until an arbitrary percentage of the total values encoded with a single dictionary is processed, typically around 10-25%. To ensure that we capture the most valuable strings early, we initially set a low threshold for frequency, allowing frequently occurring strings to quickly enter the USD. This prevents valuable strings from being materialized without first being inserted into the per-query dictionary.

However, since this sampling is performed incrementally and without direct access to the entire set of encoded values, it operates on a best-effort basis. Therefore, its effectiveness varies depending on the characteristics of the data and the distribution of strings within each dictionary block.

The implementation is straightforward. For each new dictionary, two arrays are allocated, each sized to the number of unique values in the block. The first array tracks the frequency of each unique value, while the second indicates whether a value has already been inserted into the USD. These arrays are reallocated whenever a new dictionary is encountered and are updated throughout the sampling process.

[3] `https://github.com/duckdb/duckdb/blob/223ff0a7dba7900039d5910a009247ef097fff3c/src/common/types/data_chunk.cpp#L152`

6.3 Evaluation

To test our hypothesis, we reran the experiment from the previous chapter, focusing on high-cardinality columns. This time, we introduced a third system: DuckDB enhanced with sampling-based USD. The results are shown in Figure 6.2. Since we only insert the most frequent values from each per-block dictionary, no additional constraints were needed to filter out high-cardinality columns. Despite this, the approach did not lead to any performance improvements.

Across all workloads, between 25% and 50% of the total tuples were admitted into USD while inserting as few strings as possible. However, the added overhead from sampling and allocating memory for large dictionaries, combined with the best-effort nature of our strategy, may have canceled out the potential gains. Another possible factor is branch misprediction. The USD integration introduces extra checks in the critical execution path to determine whether a string is backed by USD. In low-cardinality columns, where all values fit in USD, these checks do not cause overhead. However, when only a subset of values is stored in USD, these branches become more challenging for the CPU to predict, potentially leading to frequent mispredictions and degraded performance.

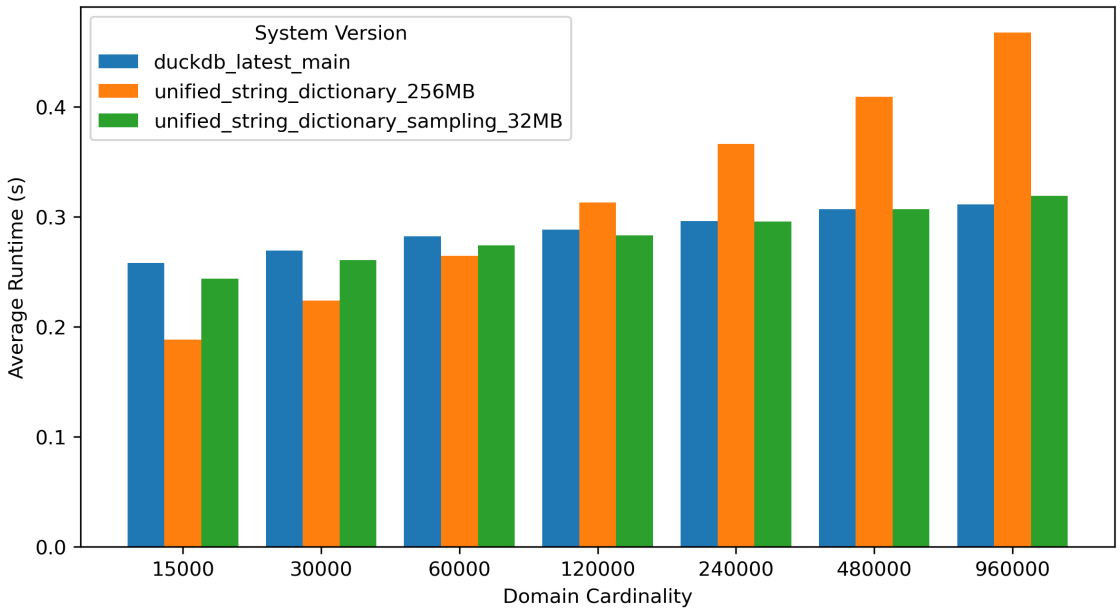


Figure 6.2: Performance results - Sampling evaluation, high cardinality columns

7

Conclusion and Future work

7.1 Conclusion

First, to answer the initial research questions we had at the beginning of this thesis:

RQ1: How can we implement and integrate a per-query, global dictionary for strings in DuckDB? Given that the USSR data structure was only implemented in Vectorwise, how would this adaptation look in a more modern system such as DuckDB?
The core data structure itself follows the USSR design as described in the paper with two major modifications. First, we allow the initialization of larger data regions for USD, allowing more unique strings to be inserted. Second, our implementation does not necessarily require the alignment of the memory region since we can encode whether a string is backed by the per-query dictionary in the upper bits of the pointer. To properly integrate this feature in DuckDB, we propose an optimizer rule that will determine whether USD should be activated for a particular query and which columns should be inserted. To carry out this task, we implemented an insertion operator that can be placed before materializing operators. Finally, several changes are required to the core execution engine of DuckDB to properly handle USD-backed strings during query execution.

RQ2: Considering the multi-threaded execution environment of DuckDB, how can we efficiently implement a global dictionary that achieves parallel efficiency?
We enforced the correctness of our data structure under concurrent insertion by implementing both locking and lock-free approaches. The locking approach initially used fine-grained mutex locks to ensure atomicity during insertions, but it became a bottleneck under specific workloads. To address this, a lock-free method using atomic

primitives, such as compare-and-swap and fetch-and-add operations, was developed, which achieves less contention and improves performance in concurrent execution.

RQ3: How much performance can be gained for string-heavy workloads by implementing such a data structure in DuckDB? Is there a possible chance of performance regression? If so, how to avoid it?

We evaluated USD on a wide range of benchmarks, including real-world and synthetic datasets. These evaluations showed that USD brings substantial benefits for low-cardinality columns. In these cases, compressed execution is possible across multiple operators, including joins and aggregations, resulting in performance gains of 1.3x to 7x. However, we also analyzed the limitations of USD in high-cardinality workloads, where the benefits drop off due to capacity limits and increased overhead. To avoid regression, we attempted sampling approaches and introduced defensive constraints that disable insertion into USD when high-cardinality columns are encountered.

Overall, this thesis shows that the Unified String Dictionary is a practical and effective tool for column-wide compressed execution on strings. It delivers notable performance improvements for specific workloads and integrates seamlessly with DuckDB.

7.2 Future work

7.2.1 Multiple USDs

Our current design assigns one per-query dictionary for all columns used in a query. Although this simplifies the implementation and integration, it does not lead to ideal data locality since values from different columns are stored together in the data region. A sensible approach is to assign a per-query dictionary to each column. This is especially feasible due to the pointer tagging approach, which provides a uniform solution for USD-string recognition. Furthermore, we can further utilize the optimizer rule to guarantee that join columns are inserted into the same USD.

7.2.2 Integration with hybrid execution model

In systems that utilize the hybrid query execution model [6], some parts of the query are executed on the cloud, while the rest are executed on the client's local machine. The data is serialized and transmitted over the network. A simple integration of USD in such systems could be achieved by using the per-query dictionary either exclusively on the cloud side or the client side. The other approach would be to reinsert the newly transmitted strings

into the per-query dictionary when the data is transferred. Whether this would lead to performance gain or not requires further benchmarking and careful modification of the optimizer rule.

7.2.3 Optimized sampling approach

One problem encountered during our attempts for the sampling-based strategy was DuckDB's limitation in terms of support for operator types. The USD insertion operator is an intermediate operator that cannot delay the vectors without causing extra overhead. It would be interesting to introduce a new operator class in DuckDB, in addition to source, sink, and intermediate operators. A so-called *sampling* operator. This operator is suitable for runtime and will need to observe the first few vectors without emitting any outputs. This should be supported by the pipeline executor so that no extra overhead, such as copying the first few vectors emitted from each dictionary block, is incurred. Furthermore, we believe that through further optimizations and the use of additional heuristics, it may still be possible to achieve even greater performance than we have currently attained.

Finally, an interesting direction for future improvement is to enhance the per-block dictionaries used during compression to store unique values while sorted by their frequency. Having access to such per-block dictionaries would eliminate the need for sampling altogether.

7.2.4 Compressed execution for sorting

USD will enable executing various operations, such as hashing, checking for equality, and copying, on the compressed form of the data. We believe it is also possible to enable comparison for sorting on the compressed data. To achieve this, a few key changes are required. First, a new API call must be added to USD to create a mapping between the unsorted offsets of strings in the data region, which correspond to the start of each string, and the sorted codes. Second, the optimizer rule must be modified to set a new flag in the insertion operator. This new flag will determine if the new API is called or not. The flag is set if the optimizer rule detects that a LOGICAL_ORDER_BY operator in the query operates on string columns. Once the flag is set, after an arbitrary number of values have been inserted into USD, the new API is called. The new mapping is created and can be used by the sort operator. In terms of changes to the execution engine, extra logic is required to detect USD-backed strings and perform the sort operation on the sorted codes rather than string values. Due to how sorting is implemented in DuckDB with the use of

key normalization [28], `string_t` API is not utilized, leading to further engine changes. Extensive benchmarking is also required to verify that no performance regression is caused as a result of inserting sort keys into USD.

7.2.5 Cost-based optimizer rule

Currently, strings are inserted into USD without considering any statistics or heuristics beyond checking for high cardinality. DuckDB files provide extra information about the data stored in each segment or column. The optimizer rule could be modified with additional heuristics to take into account domain-cardinality or string lengths to set limits for columns that have a lower chance of resulting in performance gains. Other heuristics, such as the ratio of encoded values to the unique values, are other options for consideration.

References

[1] *A Deep Dive into German Strings*. en-us. Section: blog. July 2024. URL: https://cedardb.com/blog/strings_deep_dive/ (visited on 02/23/2025) (30).

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems." en. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. Chicago IL USA: ACM, June 2006, pp. 671–682. URL: https://dl.acm.org/doi/10.1145/1142473.1142548 (visited on 02/06/2025) (1, 9, 10, 16, 19, 20, 23).

[3] Daniel J Abadi. "Query Execution in Column-Oriented Database Systems." en. In: () (19).

[4] Gennady Antoshenkov. "Dictionary-based order-preserving string compression." en. In: *The VLDB Journal The International Journal on Very Large Data Bases* 6.1 (Feb. 1997), pp. 26–39. URL: http://link.springer.com/10.1007/s007780050031 (visited on 07/07/2025) (25).

[5] *Arrow Columnar Format: Variable-Size Binary View Layout*. https://arrow.apache.org/docs/format/Columnar.html#variable-size-binary-view-layout. Specification page, accessed 2025-07-11. Apache Software Foundation, 2025 (29).

[6] RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves LeMaout, Boaz Leskes, Yao Liu, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. "MotherDuck: DuckDB in the cloud and in the client." en. In: (2024) (74).

[7] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based order-preserving string compression for main memory column stores." en. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. Providence Rhode Island USA: ACM, June 2009, pp. 283–296. URL: https://dl.acm.org/doi/10.1145/1559845.1559877 (visited on 06/23/2025) (10, 25–27).

[8] Peter Boncz, Stefan Manegold, and Martin L Kersten. "Database Architecture Optimized for the new Bottleneck: Memory Access." en. In: () (20).

[9] Peter Boncz, Thomas Neumann, and Viktor Leis. "FSST: fast random access string compression." en. In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pp. 2649–2661. URL: https://dl.acm.org/doi/10.14778/3407790.3407851 (visited on 02/04/2025) (1, 10, 21, 27).

[10] Peter Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." en. In: () (7, 12, 20).

[11] "C-store: a column-oriented DBMS." en. In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 1st ed. Association for Computing Machinery, Dec. 2018, pp. 491–518. URL: https://dl.acm.org/citation.cfm?id=3226638 (visited on 07/08/2025) (23).

[12] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. "Query optimization in compressed database systems." en. In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. Santa Barbara California USA: ACM, May 2001, pp. 271–282. URL: https://dl.acm.org/doi/10.1145/375663.375692 (visited on 02/17/2025) (1, 22).

[13] *ClickHouse/ClickBench*. original-date: 2022-07-11T20:36:51Z. July 2025. URL: https://github.com/ClickHouse/ClickBench (visited on 07/28/2025) (67).

[14] E F Codd. "A Relational Model of Data for Large Shared Data Banks." en. In: () (6).

[15] *cwida/public_bi_benchmark*. original-date: 2019-02-04T10:00:26Z. Feb. 2025. URL: https://github.com/cwida/public_bi_benchmark (visited on 02/14/2025) (60).

[16] Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. "SchemaPile: A Large Collection of Relational Database Schemas." en. In: *Proceedings of the ACM on Management of Data* 2.3 (May 2024), pp. 1–25. URL: https://dl.acm.org/doi/10.1145/3654975 (visited on 06/25/2025) (1).

[17] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications." en. In: *ACM SIGMOD Record* 40.4 (Jan. 2012), pp. 45–51. URL: https://dl.acm.org/doi/10.1145/2094114.2094126 (visited on 03/01/2025) (6, 22, 24).

[18] Yannis Foufoulas, Lefteris Sidirourgos, Eleftherios Stamatogiannakis, and Yannis Ioannidis. "Adaptive Compression for Fast Scans on String Columns." en. In: *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event

China: ACM, June 2021, pp. 554–562. URL: https://dl.acm.org/doi/10.1145/3448016.3452798 (visited on 06/30/2025) (27, 28).

[19] Bogdan Ghit, Diego Tomé, and Peter Boncz. "White-box Compression: Learning and Exploiting Compact Table Representations." en. In: () (60).

[20] G. Graefe and L.D. Shapiro. "Data compression and database performance." en. In: *[Proceedings] 1991 Symposium on Applied Computing*. Kansas City, MO, USA: IEEE Comput. Soc. Press, 1991, pp. 22–27. URL: http://ieeexplore.ieee.org/document/143840/ (visited on 06/27/2025) (1, 20).

[21] Paul Groß and Peter Boncz. "Adaptive Factorization Using Linear-Chained Hash Tables." en. In: (2025) (52).

[22] PostgreSQL Global Development Group. *PostgreSQL*. en. July 2025. URL: https://www.postgresql.org/ (visited on 07/30/2025) (5).

[23] Tim Gubner, Viktor Leis, and Peter Boncz. "Optimistically Compressed Hash Tables & Strings in theUSSR." en. In: *ACM SIGMOD Record* 50.1 (June 2021), pp. 60–67. URL: https://dl.acm.org/doi/10.1145/3471485.3471500 (visited on 02/04/2025) (1, 16, 17, 27, 33).

[24] Brian Hentschel, Michael S. Kester, and Stratos Idreos. "Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation." en. In: *Proceedings of the 2018 International Conference on Management of Data*. Houston TX USA: ACM, May 2018, pp. 857–872. URL: https://dl.acm.org/doi/10.1145/3183713.3196911 (visited on 07/20/2025) (20).

[25] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. "How to barter bits for chronons: compression and bandwidth trade offs for database scans." en. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. Beijing China: ACM, June 2007, pp. 389–400. URL: https://dl.acm.org/doi/10.1145/1247480.1247525 (visited on 06/23/2025) (10).

[26] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." en. In: *2011 IEEE 27th International Conference on Data Engineering*. Hannover, Germany: IEEE, Apr. 2011, pp. 195–206. URL: http://ieeexplore.ieee.org/document/5767867/ (visited on 07/20/2025) (6, 7).

[27] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. "Robust External Hash Aggregation in the Solid State Age." en. In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. Utrecht, Netherlands: IEEE, May 2024, pp. 3753–3766. URL: https://ieeexplore.ieee.org/document/10597735/ (visited on 03/31/2025) (8, 14, 29, 54–56).

[28] Laurens Kuiper and Hannes Mühleisen. "These Rows Are Made for Sorting and That's Just What We'll Do." en. In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. Anaheim, CA, USA: IEEE, Apr. 2023, pp. 2050–2062. URL: https://ieeexplore.ieee.org/document/10184754/ (visited on 08/12/2025) (76).

[29] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation." en. In: *Proceedings of the 2016 International Conference on Management of Data*. San Francisco California USA: ACM, June 2016, pp. 311–326. URL: https://dl.acm.org/doi/10.1145/2882903.2882925 (visited on 06/23/2025) (21, 28).

[30] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. "Real-time analytical processing with SQL server." en. In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015). Publisher: Association for Computing Machinery (ACM), pp. 1740–1751. URL: https://dl.acm.org/doi/10.14778/2824032.2824071 (visited on 07/11/2025) (21).

[31] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. "Joins on encoded and partitioned data." en. In: *Proceedings of the VLDB Endowment* 7.13 (Aug. 2014), pp. 1355–1366. URL: https://dl.acm.org/doi/10.14778/2733004.2733008 (visited on 02/06/2025) (21, 22).

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How good are query optimizers, really?" en. In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015). Publisher: Association for Computing Machinery (ACM), pp. 204–215. URL: https://dl.acm.org/doi/10.14778/2850583.2850594 (visited on 07/22/2025) (59).

[33] Yinan Li and Jignesh M. Patel. "BitWeaving: fast scans for main memory data processing." en. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York New York USA: ACM, June 2013, pp. 289–300. URL: https://dl.acm.org/doi/10.1145/2463676.2465322 (visited on 06/27/2025) (21).

[34] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. "Mostly Order Preserving Dictionaries." en. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. Macao, Macao: IEEE, Apr.

2019, pp. 1214–1225. URL: https://ieeexplore.ieee.org/document/8731521/ (visited on 06/04/2025) (26).

[35] Alicja Luszczak. "Simple Solutions for Compressed Execution in Vectorized Database System." Master's thesis. Vrije Universiteit Amsterdam, 2011 (23).

[36] *MySQL*. URL: https://www.mysql.com/ (visited on 07/30/2025) (5).

[37] Raghunath Othayoth Nambiar and Meikel Poess. "The making of TPC-DS." In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 1049–1058 (59).

[38] Thomas Neumann and Michael Freitag. "Umbra: A Disk-Based System with In-Memory Performance." en. In: () (7, 21, 29).

[39] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. "Quickstep: a data platform based on the scaling-up approach." en. In: *Proceedings of the VLDB Endowment* 11.6 (Feb. 2018). Publisher: Association for Computing Machinery (ACM), pp. 663–676. URL: https://dl.acm.org/doi/10.14778/3184470.3184471 (visited on 07/20/2025) (21).

[40] *PyPI Download Stats*. URL: https://pypistats.org/packages/duckdb (visited on 07/25/2025) (11).

[41] Mark Raasveldt. *Lightweight Compression in DuckDB*. en. Oct. 2022. URL: https://duckdb.org/2022/10/28/lightweight-compression.html (visited on 02/13/2025) (14, 23, 28).

[42] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database." en. In: *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, June 2019, pp. 1981–1984. URL: https://dl.acm.org/doi/10.1145/3299869.3320212 (visited on 02/08/2025) (1, 5).

[43] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. "DB2 with BLU acceleration: so much more than just a column store." en. In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1080–1091. URL: https://dl.acm.org/doi/10.14778/2536222.2536233 (visited on 06/23/2025) (10, 21, 22, 25).

[44] Gautam Ray, Jayant Haritsa, and Sunita Seshadri. "Database Compression: A Performance Enhancement Tool." In: (Sept. 2004) (19).

[45] Mark A. Roth and Scott J. Van Horn. “Database compression.” en. In: *ACM SIGMOD Record* 22.3 (Sept. 1993). Publisher: Association for Computing Machinery (ACM), pp. 31–39. URL: https://dl.acm.org/doi/10.1145/163090.163096 (visited on 07/22/2025) (19).

[46] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. “ClickHouse - Lightning Fast Analytics for Everyone.” en. In: *Proceedings of the VLDB Endowment* 17.12 (Aug. 2024). Publisher: Association for Computing Machinery (ACM), pp. 3731–3744. URL: https://dl.acm.org/doi/10.14778/3685800.3685802 (visited on 07/11/2025) (31).

[47] Vishal Sikka, Franz Färber, and Wolfgang Lehner. “Efficient transaction processing in SAP HANA database: the end of a column store myth.” en. In: () (24).

[48] *TPC-H Homepage*. URL: https://www.tpc.org/tpch/ (visited on 07/24/2025) (11, 57).

[49] 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.” en. In: *Proceedings of the VLDB Endowment* 17.11 (July 2024), pp. 3694–3706. URL: https://dl.acm.org/doi/10.14778/3681954.3682031 (visited on 07/05/2025) (1).

[50] *Vectors — Velox documentation*. URL: https://facebookincubator.github.io/velox/develop/vectors.html (visited on 07/11/2025) (29).

[51] Ritchie Vink. *Why We Have Rewritten the String Data Type*. https://pola.rs/posts/polars-string-type/. Blog post, accessed 2025-07-11. Jan. 2024 (29).

[52] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. “Get Real: How Benchmarks Fail to Represent the Real World.” en. In: *Proceedings of the Workshop on Testing Database Systems*. Houston TX USA: ACM, June 2018, pp. 1–6. URL: https://dl.acm.org/doi/10.1145/3209950.3209952 (visited on 02/04/2025) (1, 58, 60, 65).

[53] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. “The implementation and performance of compressed databases.” en. In: *ACM SIGMOD Record* 29.3 (Sept. 2000), pp. 55–67. URL: https://dl.acm.org/doi/10.1145/362084.362137 (visited on 07/07/2025) (20, 22).

[54] *Why German Strings are Everywhere*. en-us. Section: blog. July 2024. URL: https://cedardb.com/blog/german_strings/ (visited on 02/13/2025) (29).

[55] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. "SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units." en. In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009). Publisher: Association for Computing Machinery (ACM), pp. 385–394. URL: https://dl.acm.org/doi/10.14778/1687627.1687671 (visited on 07/20/2025) (21).

[56] *x86-64*. en. Page Version ID: 1301541626. July 2025. URL: https://en.wikipedia.org/w/index.php?title=X86-64&oldid=1301541626#Canonical_form_addresses (visited on 07/24/2025) (52).

[57] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. "An Empirical Evaluation of Columnar Storage Formats." en. In: *Proceedings of the VLDB Endowment* 17.2 (Oct. 2023). Publisher: Association for Computing Machinery (ACM), pp. 148–161. URL: https://dl.acm.org/doi/10.14778/3626292.3626298 (visited on 07/11/2025) (28).

[58] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. "SAHA: A String Adaptive Hash Table for Analytical Databases." en. In: *Applied Sciences* 10.6 (Mar. 2020), p. 1915. URL: https://www.mdpi.com/2076-3417/10/6/1915 (visited on 06/10/2025) (29–31).

[59] Jingren Zhou and Kenneth A Ross. "Implementing Database Operations Using SIMD Instructions." en. In: () (1).

[60] *Zipf's law*. en. Page Version ID: 1302941732. July 2025. URL: https://en.wikipedia.org/w/index.php?title=Zipf%27s_law&oldid=1302941732 (visited on 08/01/2025) (62).

[61] M. Zukowski, S. Heman, N. Nes, and P. Boncz. "Super-Scalar RAM-CPU Cache Compression." en. In: *22nd International Conference on Data Engineering (ICDE'06)*. Atlanta, GA, USA: IEEE, 2006, pp. 59–59. URL: http://ieeexplore.ieee.org/document/1617427/ (visited on 02/05/2025) (10).

[62] Marcin Zukowski, Mark Van De Wiel, and Peter Boncz. "Vectorwise: A Vectorized Analytical DBMS." en. In: *2012 IEEE 28th International Conference on Data Engineering*. Arlington, VA, USA: IEEE, Apr. 2012, pp. 1349–1350. URL: http://ieeexplore.ieee.org/document/6228203/ (visited on 02/20/2025) (2, 7, 23).