THE UNIVERSITY OF CHICAGO

EFFICIENT LOSSLESS COMPRESSION IN AND BEYOND COLUMNAR DATABASES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
HAO JIANG

CHICAGO, ILLINOIS
DECEMBER, 2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to thank my family for their heartful support during my Ph.D. study. My wife, Hong, is the captain of our family ship. She supports me both emotionally and physically. She suggests I start pursuing a Ph.D. She took care of all the family logistics so I can focus on my study and research. She is always confident in me, even when I lost confidence in myself. Without her, I will not be able to make it this far. My kids, Claire and Michael, bring joy and happiness to our family. The smiles on their face always give me the strength to struggle further. My parents and parents-in-law also support us to their best. I will endeavor to provide my family a better future.

I would like to thank my advisor Professor Aaron J. Elmore, for his excellent guidance and help during my Ph.D. study. He leads me into the battlefield of academia and shows me the way of survival here. He does not only give me a fish but also teaches me to fish. He gives suggestions but always respects the students' opinions. He is always willing to provide his best support during my study, research, internship and job hunting. Professor Elmore serves as my academic role model. When I become a professor, I want to be a professor like him.

I would like to thank my dissertation committee members, Professor Sanjay Krishnan and Professor Raul Castro Fernandez, for their insightful suggestions and comments on the dissertation.

I would like to thank my collaborators Chunwei Liu and John Paparrizos for their irreplaceable efforts in my research work. They work together with me during all my research projects. Their hard work and valuable suggestions are the key components to my achievement. It is a pleasure to work together with them.

My work is supported by a Google DAPA Research Award, the CERES Center for Unstoppable Computing, and gifts from FutureWei, NetApp, Cisco Systems, and Exelon Utilities. Results presented in this dissertation were obtained using the Chameleon testbed supported by the National Science Foundation. I would like to thank these institutes and companies.

ABSTRACT

Columnar databases have dominated the data analysis market for their superior performance in query processing with Big data. However, the extensive data size also brings challenges to data storage and transfer. While people often rely on lossless compression techniques to reduce storage size, database researchers overlook compression in row-wise databases. There are two primary reasons. First, available compression algorithms in row-wise databases are limited. Row-wise databases blend data fields of all types together. Byte-oriented compression algorithms such as Gzip and Snappy are the only choices. Second, Gzip-like algorithms process data in blocks and decompress an entire block before accessing a data row. The decompression is CPU intensive and has a significant impact to query performance. Lack of alternatives and implications to performance impede the applications of compression in row-wise databases. The prosperity of columnar databases changes this situation. Storing data in separated columns enable the application of compression algorithms designed for a single data type. There are also algorithms performing record-level compression, allowing the queries to skip irrelevant records and executes more efficiently. Compression in columnar databases thus reduces data storage and brings the opportunities of improving query efficiency. Besides relational databases, we also explore the benefit of compression in key-value stores. Key-value stores have wide applications, including game, IoT, Social Media, Mobile Devices, and Enterprise Applications. They could provide far better performance than the relational database in specific scenarios. This thesis proposes innovative compression algorithms and system designs to improve the storage and query efficiency in columnar databases and key-value stores. We address three challenges of lossless compression in columnar databases: better encoding algorithms, faster query on encoded data, and selecting proper encoding algorithms for data columns. We present PIDS, a novel compression approach for string columns; SBoost, a C++ library for fast queries on encoded data; and CodecDB, an encoding-aware database with a data-driven encoding selection. We explore the possibility of using compression to accelerate LSM-tree and present CoLoM. This key-value store utilizes a columnar layout and lightweight encoding to improve LSM-tree efficiency. We show that these innovations allow columnar databases and key-value stores to excel the competitors in storage efficiency and query speed.

CHAPTER 1

INTRODUCTION

Over the past decade, columnar databases dominated the data analysis market due to their ability to minimize data reading, maximize cache-line efficiency, and perform effective data compression. These advantages lead to orders of magnitude improvement for scan-intensive queries compared to row stores [60, 134]. As a result, academic research [114, 2, 63, 1], open-source communities [13, 12], and large commercial database vendors, such as Microsoft, IBM, and Oracle are embracing columnar architectures.

Besides columnar databases, the key-value store is another widely used data storage paradigm for big data processing. A key-value store is logically a dictionary that maintains a collection of key-value pairs. The key-value store does not keep a schema for its records. This is different from a relational database, where records are governed under table schemas. Instead, the key-value store allows each record to have various fields. The simple data structure allows key-value stores to have more flexibility in its applications compared to relational databases. Furthermore, key-value stores only support simple operations, such as get, put, and delete, allowing them to respond to operations faster and achieve higher throughput. These features propel their wide applications in game, IoT, social media, mobile devices, and enterprise systems [38].

The prosperity of columnar databases and key-value stores is primarily boosted by the drastically increased cyber-footprint of Internet users and IoT devices. Forbes reports [89] in 2018 that over 2.5 quintillion bytes of data are created every day, and this number is rapidly increasing. Such a huge data size poses significant pressure on data storage. As a result, many companies now delegate their data storage to cloud storage providers, which charge the client for data storage and data access. Reducing data size benefits both and can bring significant savings.

Lossless data compression has been widely used to reduced data size. However, studies on compression techniques in data storage before the era of columnar databases draw little attention in both research and industry. The reasons is twofold. First, many data stores, including row-wise database and key-value stores persist data as a list of tuples, in which data fields of different types are stored consecutively. Therefore, systems tend to rely on byte-oriented data compression approaches, such as Gzip and Snappy, to compress data fields of mixed types. Many compression algorithms that are designed for a single data type, such as dictionary encoding and bit-packed encoding, do not apply to data with mixed types and cannot be efficiently used. Gzip/Snappy becomes the de facto standard for compression in row-wise databases.

Second, Gzip-like compression approaches are block-based algorithms. These algorithms split input data into fixed size blocks, look for repetitive substrings within the blocks and replace them with shorter alternatives. As a result, decompression operations are also performed in blocks. When accessing a record, we need to decompress an entire block into memory then fetch the record from the intermediate result. This process is CPU intensive and time-consuming. Every query, even if it only access a small amount of data, needs to perform at least one block decompression before it can access the tuples. Applying compression in data stores thus significantly impacts query efficiency.

These two problems no longer exist with the introduction of columnar databases. Columnar databases persist each data column separately, with data fields of the same type stored together. Such a layout reduces the data entropy and enables more efficient compression. It also greatly expands the selection space of available compression algorithms by enabling compression/encoding algorithms designed for a single data type, such as run-length encoding, bit-packed encoding, and dictionary encoding. These encoding algorithms have preferences on datasets with particular characteristics. For example, run-length encoding works best on sorted datasets, and dictionary encoding prefers datasets with low cardinality. On these datasets, encoding algorithms are more efficient than general-purpose byte-oriented algorithms.

Many encoding algorithms for a single data type compress data entries individually. Each data entry (an integer, a line of text, etc.) in the input is transformed to an independent shorter representation in the output, and outputs from different entries do not overlap. These algorithms, usually referred to as lightweight encodings, have very low CPU consumption as the operations are usually simple and can often be performed in parallel [67]. Popular lightweight encoding schemes include dictionary encoding, bit-packed encoding, delta encoding, run-length encoding, and their hybrids [114, 22, 132, 78]. Lightweight encoding maintains entry boundaries during compression, allowing a query to access independent compressed entries without decoding irrelevant data. It also enables direct predicate execution on compressed data, skipping the decompression step [67]. These features make query execution on compressed data even faster than querying data without compression.

The same changes also happen in key-value stores. Many popular key-value stores, including Apache Cassandra [48], LevelDB [53], RocksDB [45], and HBase [8], are backed by a Log-Structured-Merge Tree (LSM-tree, [96]). The LSM-tree is a multi-layer associative array known for its support to write-intensive applications. It stores the keys and values on the disk in an interleaved manner, which is very similar to how a row-wise database stores data tables. Just as columnar databases enable better data compression and query efficiency in relational databases, introducing columnar data layout and compression techniques into the disk storage of an LSM-tree improve its query efficiency.

In this thesis, we proposes innovative compression algorithms and system designs to improve the storage and query efficiency in columnar databases and key-value stores. My thesis addresses the following questions:

- **Design New Encoding for Better Performance** Can we design new encoding algorithms for datasets with particular characteristics?

- **Speed Up Query on Encoded Data** How do we utilize the advantage of record-wise compression to make querying on encoded data faster?

- **Encoding Selection for a Data Column** There are many encoding algorithms available. Most of them also have a preference for data columns. How does the user choose among the algorithms in practice?

- **Columnar Layout in the LSM-Tree** How do we improve the LSM-tree performance using columnar layout and encodings?

We design Pattern Inference Decomposed Storage (PIDS), an innovative encoding method for string attributes in columnar stores. Using an unsupervised approach, PIDS identifies common patterns in string attributes from relational databases and uses the discovered pattern to split each attribute into sub-attributes. First, PIDS can achieve a compression ratio comparable to Snappy and Gzip by storing and encoding each sub-attribute individually. Second, PIDS can push down many query operators to sub-attributes by decomposing the attribute. This method minimizes I/O and potentially expensive comparison operations, resulting in the faster execution of query operators.

We develop SBoost, a C++ library utilizing SIMD instructions to speed up queries on encoded data. In most columnar data stores, performing queries on encoded data requires the data to be first decoded to memory, which is time-consuming. SBoost provides several novel SIMD-based algorithms to vectorize the execution and skip unnecessary decoding processes for higher efficiency. It achieves a throughput of over 10 billion numbers per second with a single thread and over 40 billion numbers per second with multi-threads for widely used bit-packed encoding and dictionary encoding. SBoost demonstrates excellent potential in speeding up query efficiency in analytic databases and in-memory stores by reducing TPC-H query time by over 60%.

We design CodecDB, a columnar database consisting of a data-driven encoding selector and an encoding-aware query engine. We notice that many existing columnar databases are encoding-oblivious. When storing the data, these systems rely on a global understanding of the dataset or the data types to derive simple rules for encoding selection. Such rule-based selection leads to unsatisfactory performance. Furthermore, when performing queries, the systems always decode data into memory, ignoring the possibility of optimizing access to encoded data. CodecDB demonstrates the benefit of tightly coupling the database design with the data encoding schemes. It chooses the most efficient encoding for a given data column. It relies on encoding-aware query operators to optimize access to encoded data. Storage-wise, CodecDB achieves on average 90% accuracy for selecting the best encoding and improves the compression ratio by up to 40% compared to the state-of-the-art encoding selection solution. Query-wise, CodecDB is on average one order of magnitude faster than the latest open-source and commercial columnar databases on the TPC-H benchmark and 3x faster than a recent research project on the Star-Schema Benchmark (SSB).

We propose CoLoM, an LSM-tree-based key-value store that utilizes the columnar layout in its disk storage. CoLoM supports using disk storage formats with different data layouts and encoding algorithms at each level of an LSM-tree, and uses a cost-based adaptive storage selection method to determine the optimal storage assignment at each level based on a given workload.

The rest of the thesis is organized as follows

- We describe the related background knowledge in Chapter 2
- We introduce PIDS in Chapter 3.
- We describe SBoost in Chapter 4.
- We discuss CodecDB in Chapter 5.
- We explore the design of CoLSM in Chapter 6.
- We describe the related works in Chapter 7 and conclude the thesis in Chapter 8.

CHAPTER 2

BACKGROUND

This chapter briefly reviews the related topics in this thesis. It includes columnar databases, lightweight encoding techniques, SIMD instructions and LSM-trees.

2.1 Column-oriented DBMS

A column-oriented DBMS is a database management system that stores relational data in columns instead of rows. A relational DBMS organizes data as logical two-dimensional data tables. Row-oriented and column-oriented storage differ in how they persist the table to disk. Data are persisted to disk row by row in a row-oriented system, with fields from the same row stored consecutively in order. In a column-oriented system, fields from the same column are stored in adjacency.

A column-oriented DBMS facilitates efficient column scans. In a row-oriented DBMS, a column scan needs to access the entire data table. A column-oriented system stores each column separately, and a column scan only accesses the target column. This reduces the disk IO and improves cache efficiency. In addition, this layout brings significant performance improvement to database operators such as filter and aggregation. In practice, we see orders of magnitudes performance improvement of columnar DBMS over traditional row-based DBMS on OLAP workloads [1].

A column-oriented DBMS also enables effective data compression. Storing data of the same type and similar content in the same place allows compression algorithms to find more repeating patterns in the data and achieve better compression results. A columnar layout also enables a wide choice of compression algorithms designed for single data type. These algorithms usually do not apply to row-oriented data layout as the data stream contains mixed types.

2.2 Lightweight Encoding

Lightweight encoding is a family of encoding algorithms featuring fast encoding and decoding speed, and reasonable well compression ratio. We briefly review several popular lightweight encoding algorithms here.

- `Bit-packed Encoding` Bit-packed encoding applies to data of integer type. It removes leading zeros from the binary representation of integers, uses n bits to loselessly represent each record a_i, where $n = \min(x|a_i < 2^x)$, and concatenates the bits as the encoding output.

- `Run-length Encoding` Run-length encoding applies to data of any type. It encodes a consecutive run of repeating numbers as a pair *(number, run-length)*. As an example, a list $[a_0, a_0, a_1, a_2, a_2, a_2, a_3, a_3, a_3, a_3]$ becomes $[a_0, 2, a_1,$
$1, a_2, 3, a_3, 4]$

- `Delta Encoding` Delta encoding applies to data of integer type. It stores delta value between consecutive numbers. For example, a list $[a_1, a_2, a_3]$ is stored as $[a_1, a_2 - a_1, a_3 - a_2]$. Delta encoding is often used together with bit-packed encoding. As the delta between numbers are usually smaller than the original numbers, bit-packing them can bring significant size reduction.

- `Dictionary Encoding` Dictionary encoding applies to data of any type. It maintains a bijection between data entries and integer keys (the dictionary), and stores the integer keys instead of original content. Bit-packed encoding can be applied to the integer keys to further achieve better compression.

Lightweight encoding algorithms usually target records of the same data type, and have many advantages compared to byte-oriented general purpose compression algorithms, such as Gzip and Snappy. Lightweight encoding algorithm employ simple record level transformations, which feature low CPU consumption and faster execution. This advantage makes lightweight encoding a perfect match to many database applications, as they need to decompress data for each query. Record-level transformation also allow lightweight encoding to preserve record boundaries in the encoded result. For example, in a bit-packed encoded stream of bit width w, the `i`-th record resides in the $w * i/8$-th byte. This feature allows *in-situ* queries on encoded data without decoding process.

2.3 SIMD Instructions

SIMD(Single-Instruction-Multiple-Data) instructions are widely supported by all modern CPUs. In particular, our algorithm focus on AVX-512/AVX2 instruction set available on recent Intel processors. AVX-512 instructions operate on 512-bit SIMD words, allowing them to manipulate 8 64-bit integers or 16 32-bit integers simultaneously. AVX2 instructions work on 256-bit SIMD words.

Our algorithms primarily utilize the following instructions. More details of the instructions can be found in Intel Intrinsics Guide [62].

- **horizontal add(hadd)** `hadd` instruction allows multiple adjacent integers (16 bit or 32 bit) in a SIMD word to be added simultaneously. Figure 2.1 shows how `hadd` of 32 bit numbers on 256-bit SIMD words. It can perform at most 8 32-bit add and 16 16-bit add with a single instruction.

- **permute** `permute` instruction allows the reordering of numbers in SIMD words. Our algorithms use `permutex2var`, which takes two SIMD words as input and a third SIMD word as permute instruction. $c = permutex2var(a, b, i)$ satisifies

$$\forall i \in [0, 8], c_i = \begin{cases} a_{n[i] \& 0x7} & n[i] \ \& \ 0x8 = 0 \\ b_{n[i] \& 0x7} & n[i] \ \& \ 0x8 = 1 \end{cases}$$

  `permute` can work on 8/16/32/64 bit granularities.

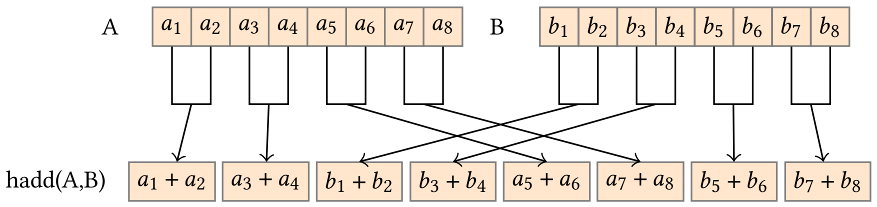

A
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
B
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
hadd(A,B)
$a_1 + a_2$ $a_3 + a_4$ $b_1 + b_2$ $b_3 + b_4$ $a_5 + a_6$ $a_7 + a_8$ $b_5 + b_6$ $b_7 + b_8$

Figure 2.1: How `hadd` works

- **arithmetic operations** includes `add, sub` operations. These operations perform pairwise integer arithmetic operations of integers stored in two SIMD words. They work on 16/32/64 bit granularities.
- **logical operations** include bitwise `and, or, xor` operations and `bit-shift` operations.

2.4 LSM-Tree

An LSM-Tree is a data structure designed for write-intensive applications. It stores key-value pairs and allows retrieving, updating, and deleting a value by a key. An LSM-tree provides good write performance utilizing an out-of-place write policy, where all insertions, updates, and deletions append new entries to a memory component. When the memory component is full, the LSM-tree sort-merges all entries by key and dumps the sorted run to disk. The LSM-tree also sort-merges shorter data runs into longer one under certain conditions, which is determined by the compaction policy it uses. During the merge, the LSM-tree discards all but the latest entry with the same key to reclaim the space. We conventionally use “level" to indicate the number of merges a run had experienced. For example, a run at level i had participated in i merges and have a size exponentially larger than runs at previous levels.

As entries with the same key scatter in the memory buffer and disk runs, a lookup on LSM-tree often needs to search more than one component and merge the results to obtain the final output. A point lookup searches runs from lower levels to higher levels and from younger ones to older ones within a level. It terminates as soon as the first entry with the target key is found. A range lookup searches every run for entries within the range and sort-merge the results. To make sure the sort-merge yields the latest version of the value for a key, the LSM-tree maintains the order of the generated runs. Thus, an entry from younger runs preempts entries from older runs with the same key. The older entries are also referred to as obsolete entries.

Upon deletions, an LSM-tree inserts an entry with a tombstone. If a point lookup encounters an entry with such a mask, it returns "not found." During a merge, an entry with a tombstone discards all older entries and itself.

Modern LSM-trees use fence pointers and Bloom Filters [20] to speed up lookups in a sorted run. Fence pointers are pointers to the first key of every block in a run. Thus, they comprise a single-layer skip list. Each run also has an in-memory bloom filter recording the keys in the corresponding run. A bloom filter is a probabilistic hash-based data structure with a zero false-negative rate but a non-negative false positive rate. A point lookup first probes the bloom filter and only access the on-disk run if the bloom filter returns positive.

The merge policy of an LSM-tree determines when and how to merge runs. Different merge policies seek balances between lookup and update operations. Frequent merges reduce the number of runs in favor of lookup operations at the cost of slower updates. Less merges work the other way. Two popular policies are leveling [96] and tiering [65]. With the leveling merging policy, there is only one run at each level. When a new run arrives at the level, the LSM-tree merges it immediately with the existing run. When the size of the run at a level exceeds a threshold, we move the run to the next level. With the tiering merge policy, there are multiple runs of the same size in one level. When the number of runs at a level reaches a threshold, the LSM-tree merges them into a larger run and moves the merged run to the upper level.

The space amplification of an LSM-tree refers to the factor of space occupied by obsolete entries. It is determined by both the workload and the merge policy. A large space amplification wastes more storage space and slows down lookups and updates as there are more entries to be searched and merged.

Popular LSM-tree implementations such as LevelDB [53] and RocksDB [45] maintain in-memory entries in a skip-list-based memory table for its speed improvement and space efficiency. LevelDB stores data in a Sorted-String Table (SSTable) when dumping a memory table to disk. SSTable stores binary key and value pairs interleaved and builds a sparse index on the keys to facilitate fast lookup operations.

CHAPTER 3
PIDS: ATTRIBUTE EXTRACTION FOR STRING COMPRESSION

Due to effective compression for minimizing storage and query latency, columnar systems are critical for modern data-intensive applications that rely on vast amounts of data generated by servers, applications, smartphones, cars, and billions of Internet of Things (IoT) devices. By persisting the same attribute from different records consecutively, columnar systems enable fast scan operations by minimizing I/O and efficient compression by keeping similar data physically close. Since compressed data must be decoded each time a query is executed, many columnar stores [13, 10, 12] allow for lightweight encoding schemes over traditional byte-oriented compression algorithms, such as Gzip and Snappy [2]. They provide significant size reduction at the cost of high overhead for decoding before reading data. Lightweight encoding is a family of compression algorithms applicable to data streams of the same type, and has less compression/decompression overhead [2] and *in situ* data filtering without decoding [67], potentially at the cost of reduced compression.

In evaluating popular lightweight encoding algorithms and Gzip on a large corpus of string attributes, we observe that even with the best lightweight encoding applied, Gzip can still further compress the encoded data on a large number of attributes. By examining these attributes, we observe that many of these attributes contain repetitive substrings across values. An example is shown in Figure 3.1a, where all rows contain “MIR”; “33F71” and “4096” also occur multiple times. These substrings are captured by Gzip, but not by lightweight encoding as compression is applied to the entire attribute value. This difference leads to the performance gap in the compression ratio between the two approaches.

We see that in many such attributes, these substring repetitions can be captured by a simple pattern. In Figure 3.1, we show excerpts from three attributes. It is obvious that `Machine Partition` follows a pattern MIR-{hex(5)}-{hex(5)}-{int}, where hex(x) repre-

MIR–00880–33BB1–512
MIR–00C80–33FB1–512
MIR–00000–337F1–4096
MIR–040C0–373F1–512
MIR–00000–337F1–4096
MIR–00C00–33F71–4096

(a) Machine partition

138A211	162
180B161	1126
120B181	771
228B149	550
177B153	362
183B109	507

(b) Ref ID

0101000020E61000000CFD083315C852C0F070116B33054440
0101000020E6100000AC88531328C352C028556E1E1E094440
0101000020E610000010A23DDD87C752COE0159AEE6C034440
0101000020E6100000CC490E29FDCE52C0289F52833B094440
0101000020E61000006495B3267FC752C048F580C01D004440
0101000020E61000004896C0D141C752C008D6E7CO6BFF4340

(c) Log ID

Figure 3.1: Sample attributes with identifiable patterns.

sents any hexadecimal number of x digits, and `Ref ID` follows pattern {hex(7)} {int}. The pattern for column `Log ID` is less obvious, but a closer look shows that all records have the same length, a common header consisting of a 17-digit hexadecimal number, “52C0” in the middle, and a common tail “40”. We can use these patterns to split string attributes into smaller components that we call *sub-attributes*. By extracting the sub-attributes and encoding them, we can potentially close the gap between lightweight encoding and Gzip.

We propose an innovative storage method, ***P****attern* ***I****nference* ***D****ecomposed* ***S****torage (PIDS)* to exploit patterns in string attributes to improve compression and query performance. PIDS employs an unsupervised algorithm to infer a *pattern* automatically from an input attribute, if applicable, stores rows that do not match the pattern as *outliers*, extracts *sub-attributes* from the matched rows using the pattern, and compresses them independently. PIDS is transparent to the user. While the sub-attributes are physically stored separately, PIDS provides a logical view that is identical to the original string attributes. PIDS rewrites query operations on the logical view to operate on sub-attributes to speed up execution.

The pattern inference algorithm in PIDS works by collecting a set of samples from the input attribute and uses a Programming-By-Example approach to extract patterns. Besides lexical similarities, such as common symbols, it also captures the semantic similarity within string attributes, allowing observing more hidden patterns. The inference algorithm provides a classifier recognizing whether or not a string attribute contains a valid pattern, which helps preclude them from the potential costly inference. PIDS also provides an intermediate language to describe the pattern, allowing it to quickly adapt to other inference algorithms or use patterns provided by the end-user.

PIDS enables more efficient compression in two ways. Exploring patterns from string attributes allows common sub-strings to be eliminated, such as removing `MIR-` from every instance in Figure 3.1a. Additionally, the extracted sub-attributes and outliers are stored as physically separated columns, on which different encoding schemes can be applied to improve compression efficiency. PIDS thus can provide a compression ratio that is comparable to Gzip, while supporting efficient encoding and decoding operations that are comparable to lightweight encoding. We empirically evaluate PIDS on a extensive collection of string attributes, showing that a large portion of them contain a valid pattern and get a compression benefit from PIDS.

PIDS uses patterns to gain insights on the attribute and facilitates efficient execution of common query operators (including equality, less and wildcard search predicates), and materialization. Given a predicate `"Machine Partition"= "ABC"`, we know immediately that it does not match any data that follows the pattern MIR-{hex(5)}-{hex(5)}-{int}. PIDS also enables a query framework to "push down" the predicates to the sub-attribute level, and potentially skip data not matching the criteria to save disk I/O and decoding effort. This is especially beneficial for wildcard queries. For example, knowing that the first sub-attribute of machine partition is a 5-digit hexadecimal, we can push down the predicate `"Machine Partition" = "MIR-00880%"` to its sub-attributes, and get an equivalent query sub_attr_1 `= 00880`. Compared to the original query, which performs a wildcard match on the entire string, the new query only needs to execute an equality check on one numeric sub-attribute, saving both I/O and computation effort. This brings up to 30x performance boost for operator execution.

The contribution of PIDS includes:

- An algorithm for discovering common patterns in string type columnar datasets.
- An intermediate language, PIDS IR, for pattern description and compiling efficient ad-hoc code for sub-attribute extraction and predicate execution.
- A PIDS prototype based on the Apache Parquet format supporting the automatic inference

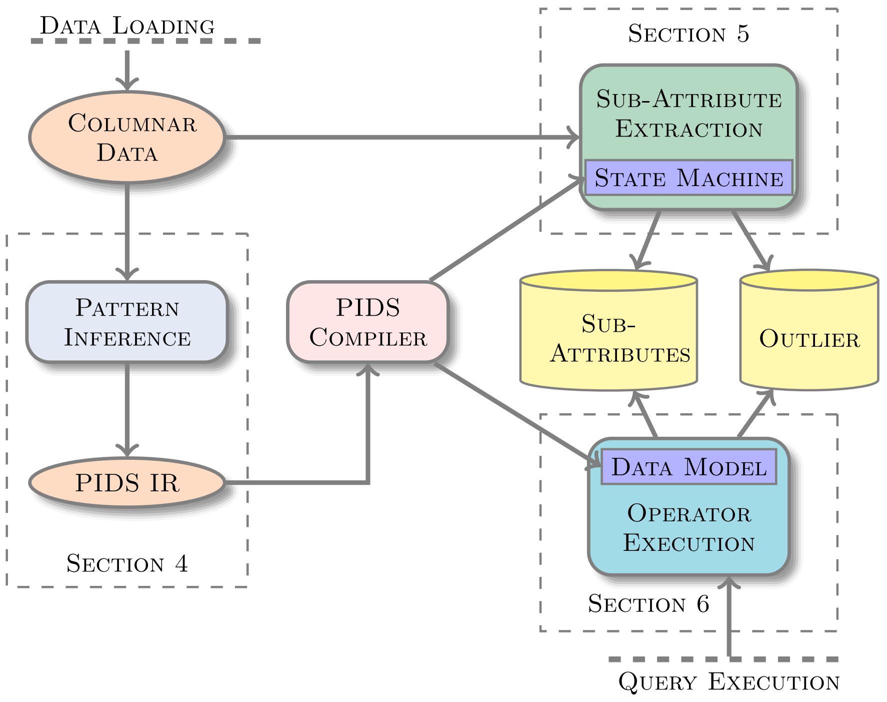


Figure 3.2: PIDS System Architecture.

of column pattern, sub-attribute extraction, and query operators on the sub-attributes.

3.1 Overview

In this section, we describe PIDS by using an example to walk through its components. Figure 3.2 shows the major components and execution steps in PIDS and their corresponding sections in the paper. The system takes a columnar string dataset as input, sampling data from each column to determine if there exists a common pattern in it, and generates a pattern expressed in PIDS IR, the intermediate representation used by PIDS to describe a pattern. The generated pattern, together with the input attribute, is sent to `Sub-Attribute Extraction`, which splits the data record into sub-attributes. To improve system efficiency, it employs the PIDS compiler to create a state machine for the pattern matching and substring extraction tasks. If PIDS cannot extract a pattern from the input columnar dataset, PIDS treats the data column as a single sub-attribute and uses existing lightweight encoding compression techniques, such as Dictionary Encoding, to compress it.

The extracted sub-attributes are then exported to external storage as separate columns, which are stored and compressed independently. As the pattern is inferred from samples, there will be a chance that some rows are not included in the sample and thus not described by the pattern. PIDS addresses this problem by maintaining an outlier store, which is separate from the sub-attribute columns. Rows that do not match the pattern are considered outliers and are stored in the outlier store in its original string form.

When executing a query operator, PIDS sends the request to `Operator Execution`, which is responsible for pushing down the operator to the sub-attribute columns, including the outlier column. The `Operator Execution` also uses the PIDS compiler to compile PIDS IR into data models and code for data loading.

3.2 Pattern Inference

In this section, we describe the pattern inference used in PIDS, which is a PBE problem with input-only examples. PIDS uses a heuristic-based search algorithm to infer patterns from examples. It treats each pattern as a state and defines a series of transformation rules between patterns. The algorithm starts from the pattern that is the enumeration of all input examples, and searches for patterns that are reachable from the starting state via the transitions. When the search ends, the pattern with the highest heuristic score is the final output.

First, we introduce PIDS IR, the intermediate language used to described a pattern in Section 3.2.1, then we describe the transition rules in detail in Section 3.2.2.

3.2.1 PIDS IR

PIDS IR is a concise language optimized for describing common patterns in string datasets. The grammar of PIDS IR is shown in Figure 3.3. The basic building blocks of a pattern are tokens, unions, and seqs. The token family includes basic types, such as `const`, `int`, `hex`, str, and sym. Instead of hard-coding a list of symbols [50], PIDS marks all characters that are not Unicode letters or digits as symbols, to be adaptive to multilingual applications. It also provides two collection types, union and seq. A union represents a set of patterns, of which at least one appears. A seq represents a list of patterns that all appear in order. As an example, the pattern shown in Figure 3.1a can be written in PIDS IR as

```
seq( const('MIR') sym('-') flhex(5) sym('-')
      flhex(5) sym('-') int )
```

Similar IRs used in a previous work [47] often target data-sets with more complex structures, such as a text corpus and log data, and provide support for nested data structures (i.e., dictionaries and arrays) and common data types (i.e., dates and timestamps). PIDS chooses not to support nested data structures. As many systems [13, 91] already provide native support for nested data structures, users who need these structures are more likely to directly leverage the native format, instead of packing the structure into a string column. It also does not include these common complex types. If these structures were to appear in the target dataset, they can easily be captured by the inference algorithm and represented in PIDS IR. This minimizes the number of built-in terms in the language, making it more concise. In Section 3.5 we show this also facilitate efficient queries.

Patterns written in PIDS IR can be easily transformed into other descriptive formats or machine code for pattern processing. For our prototype evaluation, we develop a library to generate regular expressions from PIDS IR and a compiler to directly generate Java bytecode from PIDS IR for query operator evaluation and sub-attribute extraction. The same approach can also be generalized to other languages and platforms, such as translated into LLVM [81].

3.2.2 Pattern Inference Algorithm

As described above, the inference algorithm works as a heuristic-based search, starting with generating an initial pattern, which is simply a union of all input examples, each as a const

```
pattern := token | union | seq

token   := const(val) // constant typed literal

        |  int        // int of arbitrary length

        |  hex        // hexadecimal number

        |  rangeint(min, max)  // int with min, max

        |  flint(len) // fixed-length integer

        |  flhex(len) // fixed-length hex integers

        |  str        // string of Unicode letters

        |  flstr(len) // fixed-length string

        |  sym(char)  // non-letter/digit characters

        |  empty      // no character

union   := union pattern

        |  pattern

seq     := seq pattern

        |  pattern
```

Figure 3.3: PIDS IR Grammar.

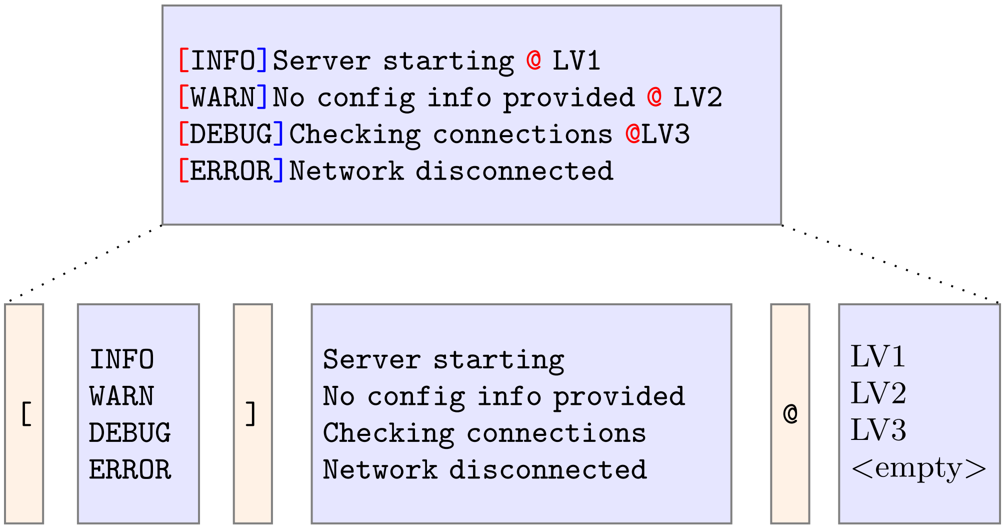

[INFO]Server starting @ LV1
[WARN]No config info provided @ LV2
[DEBUG]Checking connections @LV3
[ERROR]Network disconnected
[
INFO
WARN
DEBUG
ERROR
]
Server starting
No config info provided
Checking connections
Network disconnected
@
LV1
LV2
LV3
<empty>

Figure 3.4: Splitting a union with common symbols.

token, and iteratively applying transition rules to it until no further optimization can be done. We categorize the transition rules into two phases: Splitting and Pruning.

Splitting: In the splitting phase, PIDS performs a depth-first-scan on the pattern tree and looks into each union it encounters, searching for common tokens in union members. PIDS splits the union members into "columns" using these common tokens, and represents each column as a shorter union. This converts the original union into a seq of smaller unions and common tokens. We show an example in Figure 3.4, where the algorithm discovers "[", "]", and "@" as common tokens, and converts the original union into a seq containing three symbols and three shorter unions. In the splitting phase, PIDS applies three rules iteratively on the union to discover common patterns from its members.

The first rule, `CommonSymbol`, looks for symbols that are non-alphanumeric characters, such as hyphen, brackets, commas, and colons. These symbols commonly serve as separators between different parts of the input [47, 50]. For this reason, PIDS prioritizes the Common-Symbol rule by a "majority-take-all" approach. PIDS recognizes a symbol as a common one as long as it appears in a majority of the members (e.g., 80%). As shown in Figure 3.4, where the symbol "`@`" is extracted even if the last line does not contain it. An `empty` is left for the member without the symbol.

The second rule, `SameLength`, targets unions that have members with the same number of characters. In practice, it is fairly common that values from the same column have the same length, as in Figure 3.1c, which can serve as an indicator that some pattern exists for the data. This rule assumes that characters at the same position of each union member are the same type, and learns a character's type from all members. For example, given a union containing three const "ABDEAABD", "PA305402", and "UP25CE38", we can determine that the first two characters are letters and the remaining six characters are hex digits. This union can thus be split by the SameLength rule into two smaller unions.

Finally, if none of the above rules apply, PIDS uses the `CommonSeq` rule to look for common sequences of tokens among `union` members. This rule employs a dynamic programming algorithm to look for the longest common sub-sequence from two sequences. By applying this algorithm to the first two union members, we obtain a list of common subsequences. These subsequences are then compared against the next union member, leaving us with the common subsequences among the first three members. Repeating this process gives us the common subsequences among all union members, which can then be used to split the union.

`CommonSeq` also uses word2vec to recognize similar words in text and uses them as separators to extract patterns. Due to a lack of training corpus, PIDS does not train the word2vec model itself. Instead, it utilizes Glove [99], a pre-trained word2vec model, to convert the words in an input attribute into vectors. PIDS then goes through each record in the attribute, looking for a set of words whose pairwise cosine similarity is greater than a user-defined threshold. For example, when dealing with a string attribute of U.S. postal addresses, PIDS recognizes that the set of words "Road", "Street",and "Avenue" has a large pairwise cosine similarity. PIDS uses these words as separators to split input records.

Pruning: In the pruning step, PIDS cleans up the pattern generated in the splitting phase by removing redundant structures and merging adjacent tree nodes. This step creates a concise tree and allows the next iteration to be executed more efficiently.

PIDS executes three rules to prune the pattern tree. The `Squeeze` rule removes all unnecessary or duplicated structures, such as seqs and unions containing only one member. We list part of the transforming rule below:

```
seq(a) => a
seq(a, empty, b) => seq(a, b)
union(a) => a
```

where “a",“b" represent arbitrary patterns.

`MergeAdjacent` searches for adjacent tokens of the same type in a `seq`, and merges them together if possible.

The `Generalize` rule replaces a `union` of `const`s with generalized tokens such as `str` or `int`. For example, `union(213, 42, 442)` can be rewritten as either `int` or `rangeint(42, 442)`. Since `flint`, `rangeint`, and `flstr` contain more information about the underlying data, PIDS priorities them and only falls back to more general `int` and `str` upon failure.

By repeatedly executing these two phases, PIDS gradually constructs a concise pattern from the given samples. We use Figure 3.1a as an example to show how this works. In the splitting phase, PIDS utilizes the `CommonSymbol` rule to discover hyphens in the records as separators and obtains the following pattern.

```
seq( const(‘MIR’) sym(‘-’) union(00880, 04C80, ...)
     sym(‘-’) union(33BB1,33FB1,...) sym(‘-’)
     union(512,1024,...)  )
```

In the pruning phase, PIDS executes the `Generalize` rule on the unions, converting them into generalized tokens, and obtains the final result.

```
seq( const(‘MIR’) sym(‘-’) flhex(5) sym(‘-’)
     flhex(5) sym(‘-’) int )
```

3.3 Sub-Attribute Extraction and Compression

In this section, we present how PIDS extracts and stores sub-attributes. Section 3.3.1 describes the data extraction algorithm, Section 3.3.2 introduces how PIDS handles outliers, and Section 3.3.3 discusses how PIDS stores and compresses data and how it can improve compression efficiency.

3.3.1 Sub-Attribute Extraction

PIDS uses a state-machine based algorithm to extract sub-attributes from a target attribute. It randomly samples n rows for the target attribute and applies the pattern inference algorithm described in the previous section to generate a pattern from the samples. Let $s = seq(p_1, p_2, \ldots, p_n)$ be the pattern generated from the samples. Each p_i that is not a `const` or `sym` represents an extractable sub-attribute. We construct a directed acyclic deterministic finite state machine (DA-DFS) that describes s as follows.

1. Every token can be represented by a DA-DFS. `const` and `sym` correspond to DA-DFSs that accepting the string literals they hold. `int, hex, str` and their fixed length version correspond to DA-DFSs that takes the required number (can be infinite) of digits or letters.

2. If s_1, s_2 can be represented by DA-DFS $d(s_1)$, and $d(s_2)$, a DA-DFS for $seq(s_1, s_2)$ can be constructed by merging the last state of $d(s_1)$ with the first state of $d(s_2)$.

3. If s_1, s_2 can each be represented by DA-DFS $d(s_1)$, and $d(s_2)$, a DA-DFS for $union(s_1, s_2)$ can be constructed by merging the starting states of $d(s_1)$ and $d(s_2)$, and merging all states reachable from the starting state via the same transition sequence.

PIDS generates a DA-DFS from the pattern and marks the states representing the start/stop of each sub-attribute. When the state machine processes an input string, it will extract the sub-strings corresponding to each sub-attribute when the execution reaches these marked states.

State machines are usually implemented using a 2D array as a lookup table to store the transitions. As a result, each state transition involves several memory access operations. Since the size of the lookup table is the product of the number of states and alphabet size, for large alphabets, the size of the lookup table easily exceeds the L1 cache size as the number of states increases, making state transitions less efficient.

As the transition table is immutable during state machine execution, and the number of transitions from a single state is usually small, most entries in the 2D array table are empty. PIDS implements the state machine efficiently by hard-coding state transitions. It only uses `switch` and `if` statements to implement state transitions and sub-attribute extraction, eliminating memory access to improve efficiency.

We use the PIDS compiler to generate ad-hoc code given a pattern instance. In our prototype developed in Java, the compiler builds the DA-DFS from PIDS IR in memory, then uses the ASM library [24] to generate bytecode equivalent to the state machine. Finally, it loads the generated bytecode into JVM and applies it to the target attribute for sub-attribute extraction.

3.3.2 Handling Outliers

From Section 3.2 we know that the pattern generated by the inference algorithm is guaranteed to match all samples. However, some records may exhibit a different pattern that are not included in the samples. These records fail to be matched by the state machine and are called outliers. An example is shown in Figure 3.5, where the dataset contains two types of records, valid phone numbers and a constant value “undefined". PIDS matches the phone numbers, and treats the “undefined" records as outliers.

As outliers cannot be split into sub-attributes, we store them independently from where the sub-attributes are stored. To retain the offset information for each record, which is often used as a join key when columnar stores materialize multiple attributes, we put the records in the sub-attribute table at the same offset as they are in the original table, inserting

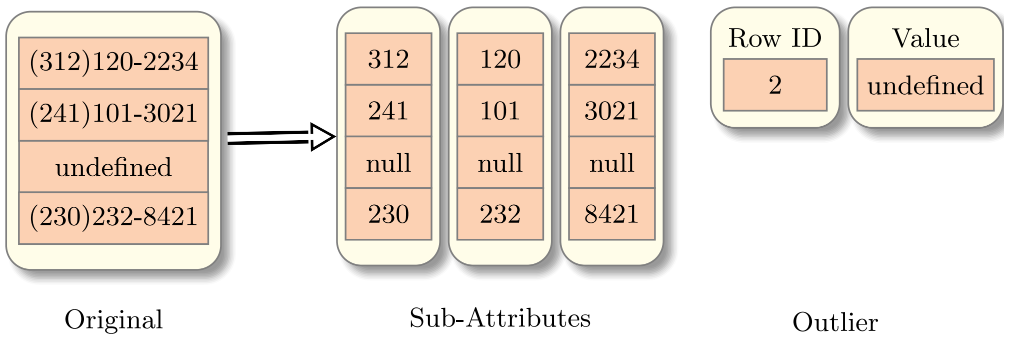


Figure 3.5: Sub-attributes are ordered as the original records and outliers are stored separately with an explicit row ID.

`null` at locations corresponding to outliers. Note that a null value in the original attribute will be treated as an outlier, and can be easily distinguished from null values that serve as placeholders for outliers. In the outlier store, we note the offset and the value explicitly as two columns.

We make this design decision based on two observed facts:

- The number of outliers should be small compared to the total number of records. PIDS targets datasets that can be described by a single pattern. If the number of outliers exceeds a certain level, then the dataset is not a good fit for PIDS.
- Most operators need to access either the sub-attribute table, or the outlier table, but not both. When an operator matches the pattern, we know it either targets the sub-attribute table or the outlier table. Only a limited number of operators, such as sorting and materialization need to access both tables.

As the number of outliers tends to be small, storing a row ID explicitly for them will save space, while still allowing us to restore outliers to their correct position during materialization. We execute operators that need to access both tables on each table separately, and merge the result. More details on operator execution can be found in Section 3.4.

3.3.3 Storage and Compression

Our PIDS prototype utilizes the Apache Parquet [13] storage format for its popularity and flexibility. A Parquet file consists of multiple row groups, which serve as a horizontal split of the columns. A row group contains several column chunks, each containing the data of one column in that row group. Data in a column chunk are stored continuously on disk and can be loaded efficiently with a sequential read. The data in column chunks are organized into pages, with each page as the unit for encoding and compression.

PIDS stores and compresses each sub-attribute independently as a column, using a dictionary and bit-pack-run-length hybrid encoding. It maintains an independent dictionary for each column chunk that translates distinct entries in the target sub-attribute to an integer code, then use run-length and bit-packing to compress the integer codes.

In practice, we notice that there are two types of attributes on which PIDS does not work well. The first type is attributes with no patterns, such as attributes with single word or natural language text. The second type is attributes with low cardinality. If an attribute already has a low cardinality, the extracted sub-attributes will likely to have similar cardinality. For such attributes, directly compressing the original attribute using a single dictionary is more efficient than compressing each sub-attribute using separated dictionaries. PIDS uses a classifier to recognize these attributes and compresses them as a single sub-attribute. To efficiently recognize and exclude these attributes, PIDS employs a k-nn classifier with Euclidean distance that weights nearest neighbors using their inverse squared distance. We use our public dataset as the training set and label attributes as positive if PIDS encodes smaller than the best encoding, and negative otherwise. We evaluate the accuracy of the model using 5-fold cross-validation (fitcknn with kfoldPredict in Matlab) with the following features from a string attribute:

- Ratio of Distinct Values $\frac{\text{cardinality}}{\text{num of records}}$
- Mean and variance of record length

- Mean of Shannon Entropy of each record

3.4 Operator Execution

To support efficient query execution directly on sub-attributes, we describe how PIDS supports common query operators, including predicate filtering and materialization. Many operators can be "pushed down" to sub-attributes, such that the operator can be decomposed into several independent operators on each sub-attribute, and the results from each sub-attribute can be combined to obtain the final output. Pushing down operators to sub-attributes also enables incremental execution, where we execute operators on the sub-attributes one at a time, skipping records on sub-attributes for which previous executions have determined that an attribute cannot satisfy the operator. For example, when executing equality predicate on a phone number, if the first sub-attribute (area code) does not match the predicate constant on some rows, we know that these rows do not match without examining the other sub-attributes. When scanning the remaining sub-attributes, we can skip these rows, saving I/O and decoding effort, thus speeding up the execution. Query rewriting from string operations (e.g. `like`, $=$, $\neq$, or $<$) to sub-attribute operations is done by PIDS and is transparent to the user. For operators that cannot be pushed down to sub-attributes, PIDS materializes sub-attributes into an in-memory data structure before applying the operator. For exposition we start with an assumption that all sub-attributes are of fixed length. In Section 3.4.4, we introduce how PIDS handles comparisons on sub-attributes of variable length.

3.4.1 Efficient Data Skipping

PIDS implements data skipping by using a bitmap to mark the positions of rows that need to be evaluated on the next sub-attribute, and update the bitmap after each sub-attribute evaluation. For our Parquet-based prototype, data skipping occurs at three levels.

- Column Chunk Level. PIDS first consults the bitmap to see if a column chunk contains rows to be accessed, then uses zone map information to determine if a column chunk can be skipped for a given operator. Skipping a column chunk saves disk I/O.

- Page Level. Similar to column chunk level, PIDS uses both bitmap and zone map to perform page-level skipping. Skipping pages saves decompression effort.
- Row Level. When scanning data on a sub-attribute, PIDS uses a bitmap to locate the next row to be read, skipping all rows in between. If the rows are encoded using lightweight encoding, PIDS skips the bytes without decoding them. Skipping rows thus saves decoding effort.

3.4.2 Predicate Filtering

We define a predicate as a tuple (OP, a), where OP is the operation and a is the constant. For example, (less, 5) on column x evaluates to true for all values where $x < 5$. PIDS implements three relational predicates `equal`, `less`, and `like`. Other predicates, such as `greater-equal`, can be obtained through logical combinations of the existing ones.

All three predicates support pushing down the operators to sub-attributes. When executing a predicate, PIDS first matches the constant against the pattern. If they do not match, execution terminates with no disk access involved. Otherwise, we use the match result to push down the predicate to the sub-attributes. We use x to represent the target column, which consists of sub-attributes $x_1, x_2, \ldots, x_m$, and assume that the constant a of the predicate has a match $(a_1, a_2, \ldots, a_m)$ to the pattern.

Equality Predicate

The equality predicate (equal, a) can be decomposed as $x = a \iff \wedge_{i=1}^{m}(x_i = a_i)$, that is, $x = a$ if and only if the equality $x_i = a_i$ holds for all sub-attributes x_i.

To skip as many rows as possible, PIDS uses a histogram to estimate the selectivity of $x_i = a_i$, and scans x_i in increasing order of selectivity. For example, if $x_1 = a_1$ is expected to match 10% of rows and $x_2 = a_2$ is expected to matches 5% of rows, we first execute $x_2 = a_2$, then $x_1 = a_1$ to allow more rows to be skipped. PIDS uses a bitmap to mark the positions where all sub-attributes x_i scanned so far satisfy $x_i = a_i$ and the equality check on the next sub-attribute is performed only on the marked positions.

Algorithm 1 Pseudocode for Equality Predicate

```
Bitmap posToScan= <full>
k = SORT_BY_SELECTIVITY(x, a)
for i = 1 to m do
    Bitmap colEqual = EQUAL(x[k[i]], a[k[i]], posToScan)
    posToScan = AND(posToScan, colEqual)
end for
return posToScan
```

When performing an equality check on sub-attributes, PIDS utilizes the encoding dictionary to translate the predicate constant into an integer code and performs the equality check on encoded data directly [67] to save decoding effort. If the constant is not in the dictionary, the entire sub-attribute can be skipped.

Less Predicate

When the sub-attributes are all fixed length, a less predicate $(less, a)$ can be pushed down to sub-attributes as a combination of $x_i < a_i$ and $x_i = a_i$. If $x_1 < a_1$, we have $x < a$. Otherwise if $x_1 = a_1$, we proceed to check x_2. Again, if $x_2 < a_2$, we have $x < a$. Otherwise if $x_2 = a_2$, we proceed to x_3. This process is repeated until all x_i have been processed. Formally, this can be written as

$$x < a \iff \vee_{i=1}^{m}[\wedge_{j=1}^{i-1}(x_j = a_j) \wedge (x_i < a_i)]$$

PIDS maintains two bitmaps, `result` for the positions satisfying $x < a$ so far, and `posToScan` for the positions to scan on next sub-attribute. They are updated as follows in the i-th iteration.

$$\begin{aligned}\texttt{posToScan}_i &= \texttt{posToScan}_{i-1} \wedge (x_i = a_i) \\ \texttt{result}_i &= \texttt{result}_{i-1} \vee (\texttt{posToScan}_{i-1} \wedge (x_i < a_i))\end{aligned} \tag{3.1}$$

PIDS iterates through all sub-attributes, computes $x_i = a_i$ and $x_i < a_i$ on rows marked by `posToScan`, and updates the bitmaps using Equation (3.1). When the iteration ends, `result` stores all records satisfying $x < a$.

Algorithm 2 Pseudocode for Less Predicate

```
Bitmap result = <empty>
Bitmap posToScan= <full>
for i = 1 to m do
    Bitmap colEqual = EQUAL($x_i, a_i$, posToScan)
    Bitmap colLess = LESS($x_i, a_i$, posToScan)
    colResult = AND(posToScan, colLess)
    result = OR(result,colResult)
    validPos = AND(posToScan, colEqual)
end for
return result
```

Like Predicate

Some like predicates, such as prefix or suffix search, may only need to access a limited set of sub-attributes, which is identifiable by matching the predicate to the pattern. For example, assume we have a pattern for phone numbers as `seq( sym('(') flint(3) sym(')') flint(3) sym('-')`
`flint(4))`. A prefix search (like, '(345)44%') has a unique match on the phone number pattern as (345, 44%, %), and can be pushed down to sub-attributes as $x_1 = 345 \wedge x_2 \sim 44\%$, where we use $\sim$ to denote the like predicate. Similarly, a suffix search (like '%442') has a match (%, %, %442) and can be pushed down as $x_3 \sim \%442$. These predicates are then evaluated using a similar approach as in the equality case. We first sort sub-attributes based on their selectivity and iterate through each x_i, executing the predicates. This allows PIDS

to greatly simplify the execution of many prefix and suffix queries as it can directly skip sub-attributes that are not included in the predicates. This approach also applies to some wildcard queries. For example, (like, %4242%) can be pushed down as $x_3 = 4242$ when we discover that only sub-attribute x_3 contains four-digit numbers.

A challenge to this approach is that in some cases the constant containing '%' can have multiple matches on the pattern. For example, "(123)%432%" has two matches against the phone number pattern, (123, 432, %), and (123, %, %432%). They lead to different execution results, and both need to be included in the final result.

PIDS solves this by collecting all possible matches, pushing each of them down to the sub-attributes, and merging and simplifying the generated expression. The example above can be written as follows when pushing down to sub-attributes: $(x_1 = 123 \wedge x_2 = 432 \wedge x_3 \sim \%) \vee (x_1 = 123 \wedge x_2 \sim \% \wedge x_3 \sim \%432\%)$. This is simplified to $x_1 = 123 \wedge ((x_2 = 432) \vee (x_3 \sim \%432\%))$. PIDS first executes $x_1 = 123$, getting a bitmap, and uses that bitmap to skip rows when scanning x_2 and x_3.

3.4.3 Materialization

PIDS implements two types of materialization operations, string materialization and fast materialization. String materialization reads fields from all sub-attributes, and composes them back to the original string format according to the pattern. This is applied to a column when a projection is performed. However, sometimes we materialize a column not for output, but only to execute operators that cannot be pushed down to sub-attributes, such as joins or hashing for group-by aggregations. In these cases, we only read fields from each sub-attribute and keep the values in an in-memory structure and not do convert the values into strings. It also excludes the content of the pattern. We call this *fast materialization*.

PIDS executes both types of materialization by reading out each sub-attribute in order and storing the values in an in-memory structure. When performing string materialization, PIDS further converts each field in the structure to strings and injects them into the proper position in the pattern. Although the algorithm is straightforward, it can become a performance bottleneck since each sub-attribute brings overhead for decoding and formatting, and this overhead accumulates as the number of sub-attributes increases.

PIDS applies many optimization techniques to mitigate overhead, including a fast algorithm to convert an integer to string, and a cache-friendly implementation to read the sub-attributes in blocks. In the prototype we developed using Java, we create a sizeable native memory as a buffer to avoid the instantiation of too many string objects and relieve the overhead brought to JVM garbage collection.

3.4.4 Sub-Attribute of Variable Length

In operators involving comparisons, such as less predicate and sorting, we push down the operators to sub-attributes based on the assumption that the ordering of the original attribute is uniquely determined by the ordering of its sub-attributes. For example, on an attribute x with two sub-attributes x_1, x_2, $x_1 < a_1 \vee (x_1 = a_1 \wedge x_2 < a_2) \implies x < a$. This is true when the sub-attributes only contain rows of the same length, but the situation becomes more complicated when some sub-attributes contain rows of variable length. An example is shown below on the left, where x_2 has variable size of 2 to 4.

x_1 x_2		Padding x_2		Add Length
e_1:313-3195-T	$\rightarrow$	3195	$\rightarrow$	3195$\underline{4}$
e_2:313-42-T	$\rightarrow$	42$\underline{00}$	$\rightarrow$	4200$\underline{2}$
e_3:313-420-T	$\rightarrow$	420$\underline{0}$	$\rightarrow$	4200$\underline{3}$

By simply extracting the sub-attributes and comparing them, we have $e_2.x_2 = 42 < 3195 = e_1.x_2 \implies e_2 < e_1$, while the right order should be $e_1 < e_2$ under string comparison. To correct this, we pad the data to restore their correct order. This padding is applied to data on the fly when performing comparisons and has no impact on data stored on disk.

In the example above, we see that the symbol following x_2 is a dash, which has a smaller ASCII code than the digits. Thus, we right-pad x_2 with 0 (marked in red, underlined) to length 4, which restores the correct order of e_1 and e_2. This padding now makes e_2 and e_3 indistinguishable, for the correct order of $e_2 < e_3$. As shorter entries in x_2 are smaller, we append the entries with their original length (marked in blue, underlined) to break the tie. After the padding, the entries in x_2 satisfy $e_i.x_2 < e_j.x_2 \implies e_i < e_j$. When the following separator is greater than a digit, the algorithm right-pads the entry with 9 instead of 0. And as shorter entries are larger when the separator's ASCII code is larger than a digit, e.g., '42**:**'>'429**:**', the algorithm appends maxLen-len(value) in this case, where maxLen is the maximal length of the target sub-attribute.

If a sub-attribute of variable length is a string type, we pad the first symbol following that sub-attribute to the value, and perform natural string comparison. For example, to compare addresses with two sub-attributes "Chicago, IL" and "Milwaukee, WI", we compare "Chicago," and "Milwaukee," by including the comma.

3.4.5 Handling Outliers

As described in Figure 3.5, PIDS stores outliers in a separate location in the original string format, along with its row ID. When we choose a valid pattern, the number of outliers should be relatively small, and we reasonably assume that the entire outlier table can be materialized in memory.

As outliers are stored in their original string format, all operators can be applied directly. When executing operators, we merge the outlier result with the result from the main table. Depending on the type of operator, different merging strategies are adopted.

For predicate execution, we generate a bitmap sized to the original data, marking the row ID of outlier records that satisfy the predicate, then perform a logical OR operation between the main data bitmap and the outlier bitmap to get the final result. For equality predicate, we notice that if the predicate matches the pattern, the result must be either be in the main table or the outlier table, but not both, so we only need to query one table, saving the logical OR operation.

For materialization, we need to merge the result from the main table with the outliers. We first materialize the outlier table as a memory buffer. When materializing data from the main table, we check null values that indicate the occurrence of outliers, and use the null value's position as the row ID to look up the memory buffer for the corresponding value. The value is inserted into the results from the main table.

3.5 Experiments

In this section, we present the experiment results showing that PIDS improves both compression and query efficiency in a columnar store. We develop a prototype of PIDS in Java and Scala, using the Apache Parquet columnar storage format [13]. Our experimental platform is equipped with 2x Xeon Silver 4116 CPU, 192G Memory, and a NVMe SSD. It runs Ubuntu 18.04 LTS, OpenJDK 1.8.0_191, and Scala 2.12.4. For all throughput results, we report the average throughput of ten runs of a fixed duration after warm-up. We run the experiments on target attributes with uniformly randomly generated predicates for each execution.

Datasets: We use two datasets in the experiments. To justify that PIDS is widely applicable, we use a dataset consisting of 9124 string attributes, collected from various real-world data sources, such as open government sites, machine learning, social networks, and machine logs. The data sources details can be found at https://github.com/UCHI-DB/comp-datasets.

To evaluate operator execution efficiency, we choose four representative string attributes: Phone Number, Timestamp, IPv6, and Address. We generate IPv6 and Phone Number data uniformly, Timestamp data uniformly in a 10-year span, and Address data using a TPC-DS data generator [95]. For each attribute, we target a number of records that require 128 MB of space in PIDS. Table 3.1 shows examples of these attributes and the number of records.

Baselines: In our experiments, we compare PIDS against popular string encoding/compression algorithms, including Parquet with *no encoding*, Parquet with lightweight encoding, Snappy, Gzip, and a Block Re-Pair Front-Coding
(BRPFC) dictionary [79]. BRPFC encodes an attribute using a dictionary, sorts the dic-

Table 3.1: Representative attributes used in our evaluation.

Attribute	Sub Attrs	Rows (million)	Example
Phone	3	35	(312)414-4125
Timestamp	8	20	2019-07-25 12:30:01 3424.24232
IPv6	8	10	3D4F:1342:4524:3319:8532:0062 :4224:53BF
Address	8	13	121 Elm St., Suite 5, Chicago, Cook County, IL 60025

tionary entries, then applies Front Coding and Re-Pair [78] on the entries. We implement BRPFC in the Apache Parquet framework and verify that our implementation has comparable performances to the original version. Due to a lack of support for SIMD in Java, we implement a scalar version of BRPFC, and use the SIMD improvement factors reported in the original paper to estimate the performance of a SIMD version [79].

In these baseline algorithms, the attributes are stored as string types in Parquet. For lightweight encoding, we empirically choose the encoding schemes in Parquet with the smallest compressed file size on the target attribute, referred to as *Best Encoding* in the rest of the paper.

Many DBMS have specific data types for timestamp data, usually backed by 64-bit integers. Storing timestamps as integers helps achieve better compression, but requires extra effort if users want to query partial fields, such as month or date. We include this approach in the baseline, referred to as *64-bit Int* in the experiments, to show that PIDS facilitates more efficient query operators and comparable storage benefits.

3.5.1 Compression Efficiency

In this section, we evaluate the compression efficiency of PIDS. In Table 3.2 we show the compressed size and compression ratio of PIDS against the baseline methods on the dataset of 9124 string attributes, and on the attributes recognized by the classifier. We see that in both cases PIDS has achieved the best compression ratio, and is 20% smaller than Gzip,

Table 3.2: PIDS achieves the best compression ratio on both the entire dataset of 9124 String Attributes (All), and the set of attributes filtered by the classifier (Cls).

	Raw	PIDS	Best Enc.	Snappy	Gzip	BRPFC
All	106G	14.6G	18.8G	33.1G	18.2G	26.6G
Cls	45G	4.2G	8.4G	9.8G	5.1G	7.8G

which is the second best. The classifier recognizes 2868 attributes as compressable and achieves an accuracy of 91.2%. PIDS infers valid patterns from 4,596 (50.73%) attributes. Of these attributes, 3,214 fully match the pattern (no outliers), and the overall average outlier percentage is 0.6%. The average length of attributes with a pattern is 19, and the average number of sub-attributes is 7. We extracted 32,105 sub-attributes, with 50% integer and 50% string.

In the top sub-figure of Figure 3.6, we evaluate PIDS and the baseline methods on the four attributes used in operator evaluation. For compression, PIDS outperforms all other approaches on the 4 attributes and achieves a 2-3 times size reduction compared to Snappy, a popular compression method used in many columnar stores. Figure 3.6 also shows that Timestamp compressed with PIDS has similar size as that stored in 64-bit Integers. In PIDS, all sub-attributes are bit-packed, and the total number of bits they occupy is no larger than a 64-bit integer.

The bottom sub-figure of Figure 3.6 shows the throughput of end-to-end compression for all approaches, which measures the time consumption including pattern inference, extraction, encoding, and persistence. BRPFC has a throughput that is 10x slower than others largely due to the recursive pairing step, making it almost invisible in the figure. PIDS's throughput varies by attribute. On the phone attribute, PIDS is slightly faster than Snappy, while on address and IPv6, PIDS is about half as fast as Snappy. To understand the factors impacting PIDS's encoding performance, we study how much time each step consumes. As pattern inference takes a constant amount of time, and disk IO is 10-20x faster compared to the overall throughput, we focus on the data extraction and encoding steps. In Figure 3.7a,

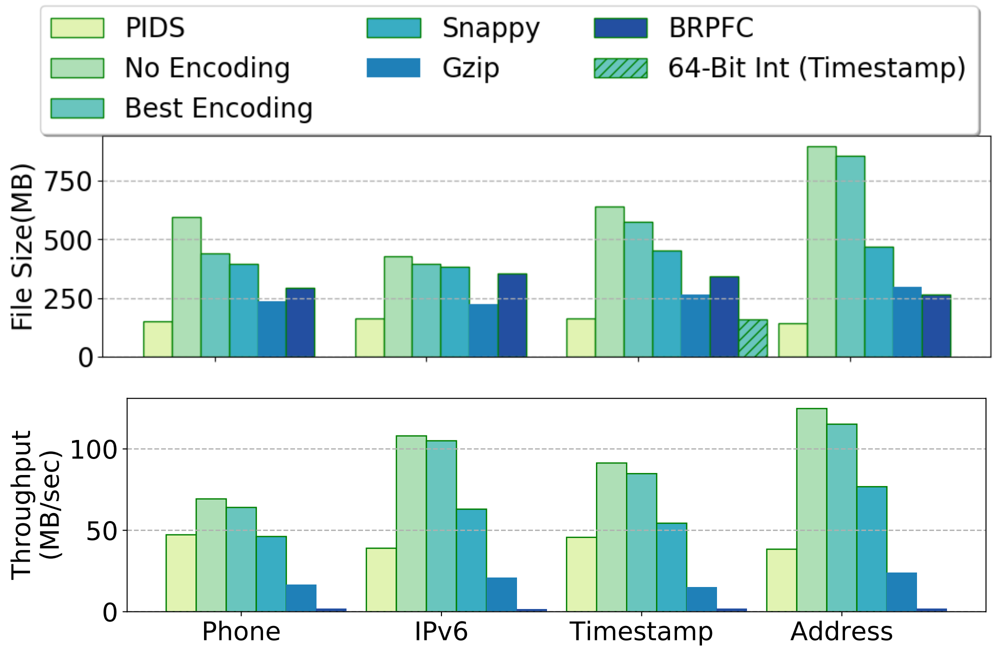
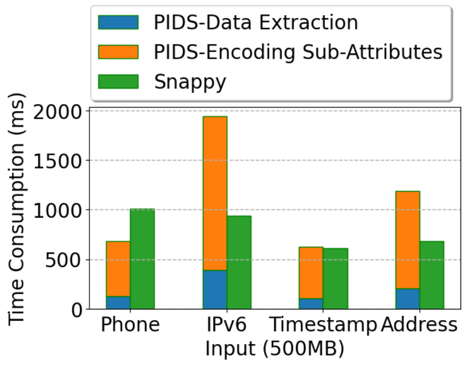
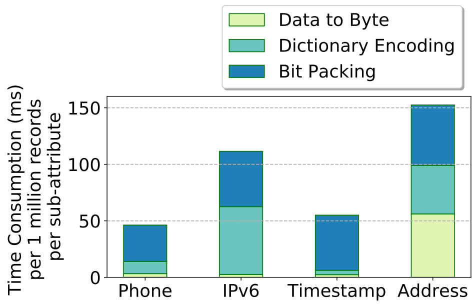


Figure 3.6: Comparing PIDS compression performance in both size and end-to-end compression throughput.



(a) PIDS spends most time on encoding sub-attributes.



(b) Average Time Consumption Per Sub-Attribute

Figure 3.7: Compression time breakdown. PIDS spends more time on attributes with more sub-attributes, higher cardinality, and more string sub-attributes.

we see that PIDS spends about 80% on the encoding step, which is also the source of the variance in the throughput.

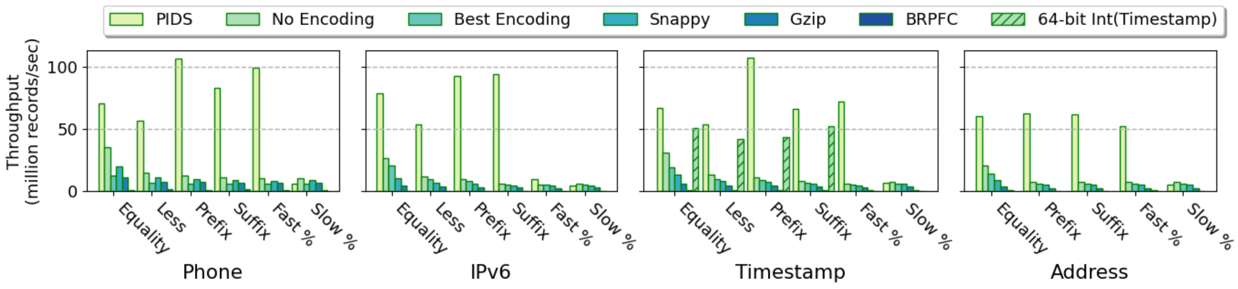


Figure 3.8: Predicate Execution. PIDS is 2-30x faster than all baselines on all four attributes for Equality, Less, Prefix and Suffix, and most Fast Wildcard Predicates. PIDS is also 20% faster than Timestamp stored as 64-bit int.

As encoding is done on each sub-attribute independently, in Figure 3.7b we study the decomposition of time consumption to process one sub-attribute on the four attributes. IPv6 and Address spend significant time on dictionary encoding because their sub-attributes have a larger cardinality and need to maintain a larger dictionary. The average cardinality for the sub-attributes in IPv6 is 65,536, and for Address it is 100,000. Whereas Phone is 4,000, and Timestamp is 3,784. Moreover, Figure 3.7b shows that Address spends much time on converting data to bytes, which the other three attributes do not. Table 3.1 shows that while Timestamp, IPv6 and Address all have 8 sub-attributes, all sub-attributes in Timestamp and IPv6 are integers, while 6 of 8 sub-attributes in Address are strings. Encoding strings to bytes takes much more time than encoding integers. The analysis above suggests that PIDS performs better on sub-attributes with low cardinality and few string types.

3.5.2 Operator Execution

In this section, we show that PIDS accelerates common query operators. Comparing the same operators on the same dataset stored in string format, PIDS brings at least 2 times performance improvement to all predicate execution tasks. In some tasks, such as prefix search, the improvement can be over 30 times.

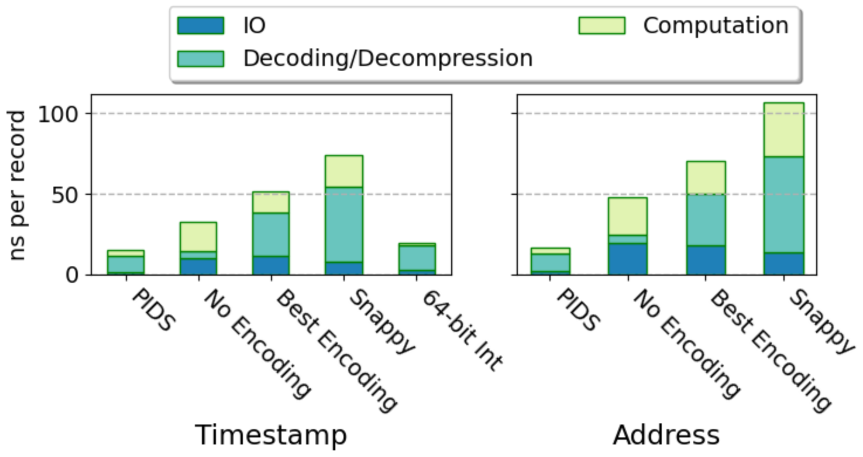


Figure 3.9: Equality cost breakdown (Gzip and BRPFC are excluded for clarity due to too high of time cost).

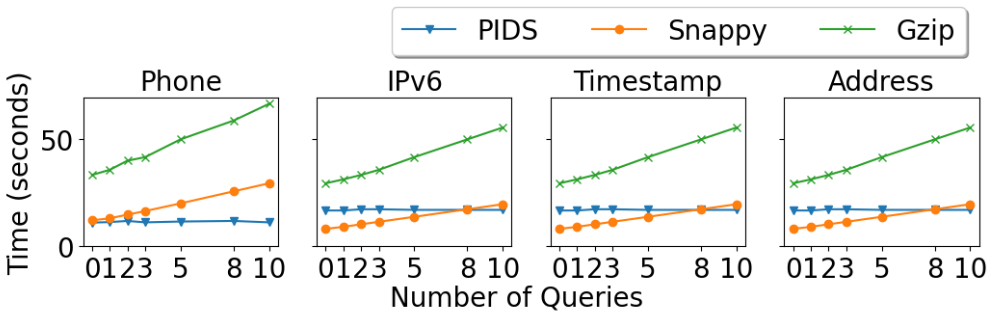


Figure 3.10: Time consumption of compressing an attribute and performing multiple equality queries.

Predicate Filtering

We compare the performance of PIDS on various predicates against the baselines. In addition to the straightforward equality and less predicates, we also test prefix, suffix, and wildcard predicates. Examples of these queries are given in Table 3.3. We experiment with two types of wildcard predicates. “Fast Wildcard” is a wildcard predicate that has a single match against the pattern. When executing a fast wildcard predicate, PIDS can push down the predicates to only the involved sub-attributes, ignoring other sub-attributes, and speed up the execution. For example, “%33:27%” is a fast wildcard predicate for the timestamp attribute, as it only matches the minute and second sub-attributes. A “Slow Wildcard” is a wildcard predicate that has more than one match against the pattern. In the worst case, PIDS needs to scan all sub-attributes when executing these predicates. "%12%" is a slow wildcard for the timestamp attribute as it can potentially match any sub-attribute for timestamp. To execute this predicate, all eight sub-attributes and the whole data file need to be accessed.

The experimental results are shown in Figure 3.8. We do not test the less predicate on the address attribute since it makes no practical sense. On the equality and less predicates, PIDS beats all string-based competitors by 2-10 times. This improvement primarily comes from data skipping by progressively filtering sub-attributes. For example, when executing an equality predicate on the IPv6 attribute, PIDS scans on average only 2.006 sub-attributes per execution, and accesses 24.35% of the whole data file.

On prefix, suffix, and most fast wildcard predicates, PIDS beats all string-based competitors by 10-30 times. As we have seen in Section 3.4.2, we can convert these predicates to equality or like predicates on one or two sub-attributes, making it even more efficient than equality predicates. The only case where fast wildcard does not have obvious performance improvement is on the IPv6 attribute, where all sub-attributes have the same length. A wildcard predicate such as $x \sim$%1A2B% can match any sub-attribute, requiring executing equality predicates on all eight sub-attributes. However, as an equality comparison is faster than a wildcard search, PIDS manages to obtain 2 times throughput compared to its fastest competitor. The only case where PIDS does not work well is the slow wildcard, where it needs to execute like predicates on all sub-attributes. Nevertheless, PIDS still yields similar performance to Snappy.

For Timestamp stored as 64-bit integers, we implement prefix query with range predicate, and suffix query with modular operation to compare against PIDS. We see that on equality, less, prefix and suffix queries, PIDS is consistently 20% faster than 64-bit integer, due to data skipping. In addition, 64-bit int performs poorly on wildcard predicates, which require converting 64-bit Int to date string. The numbers are too small to be visible in the figure.

In Figure 3.9, we show a cost breakdown of equality predicates on the timestamp and

Table 3.3: Examples of wildcard queries.

	Prefix	Suffix	Fast Wildcard	Slow Wildcard
Phone	(377)62%	%524	%42-47%	%24%
IPv6	24BD:52%	%1B2D	%243B%	%1D%
Timestamp	2017-09%	%24389	%33:27%	%16%
Address	121 Elm St.%	%IL,32036	%Chicago,IL%	%Cook%

address attributes, to help us understand how PIDS achieves its performance boost. We do not include Gzip and BRPFC in the figure as they consume much more time compared to others. Including them makes the details of other results hard to interpret. Not surprisingly, both Gzip and BRPFC spends the majority of the time in decompression. We see that the string-based competitors spend a considerable amount of time on I/O and incur large overhead in computation (primarily string comparison). Best (Lightweight) Encoding and Snappy both spend a significant amount of time on decoding/decompression. PIDS has a smaller file size, which saves I/O and it uses a dictionary to translate all predicates into integer comparisons, saving computation effort. By pushing down the predicates to sub-attributes, and performing data skipping, PIDS further reduces I/O and decoding overhead.

PIDS executes the pattern inference and data extraction task only once when it persists an attribute as a collection of sub-attributes. These steps in PIDS are similar to the compression step in Snappy and Gzip. When PIDS executes a query, it accesses each sub-attribute directly by pushing down the predicate. The query operation thus involves no pattern inference or data extraction operation and is transparent to the user. In Figure 3.10, we compare the end-to-end time consumption to read 500MB textual records from disk file, perform compression and persist to disk, then conduct multiple equality queries operations against the compressed file with Snappy, Gzip, and PIDS. Each query performs data decompression before execution. While PIDS takes slightly longer time when compressing data on some attributes, the overall time consumption is compensated by the fast query execution. Only after 8 queries, PIDS has a shorter total time on all four attributes. This is more promising, considering that PIDS also has a better compression ratio.

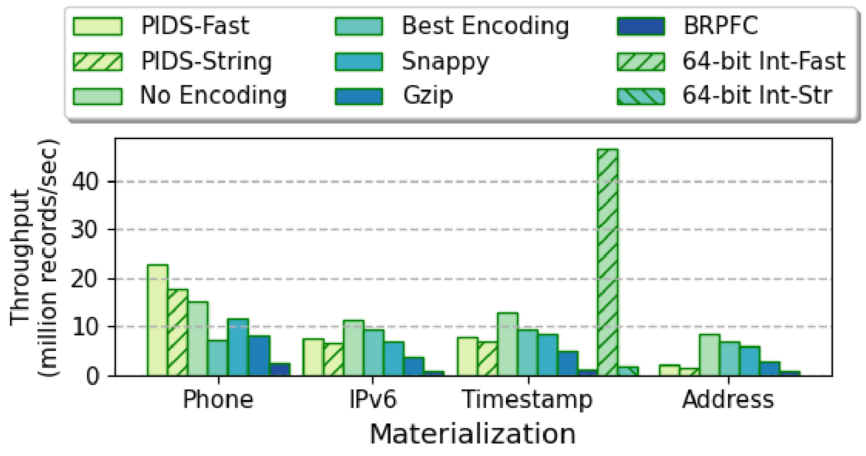


Figure 3.11: PIDS uses an optimized integer-to-string algorithm, and achieves a throughput at least as good as Snappy when doing string materialization.

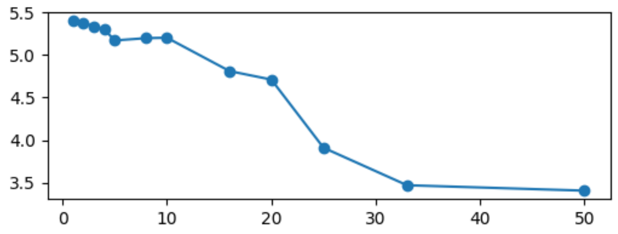


Figure 3.12: Phone string materialization with increasing the percentage of outliers.

Materialization and Outliers

Figure 3.11 shows an experiment with the two types of materialization operators: fast materialization, which loads sub-attributes into an in-memory data structure that can be used by other operators, and string materialization, which generates string results for output. We denote them by PIDS-Fast and PIDS-String respectively. For the 64-bit Int representation of timestamp data, we also show 64-bit Int-Fast, which reads 64-bit integers into memory, and 64-bit Int-String, which uses Apache Commons' `FastDateFormat` to format 64-bit integers into timestamp strings.

Since materialization requires access to the whole dataset and no data can be skipped, it is not surprising that PIDS no longer beats competitors by a large margin. Nevertheless, we see that it still outperforms all other competitors on the phone attribute. On the IPv6 and timestamp attributes, PIDS outperforms Gzip, and has a similar throughput to Snappy.

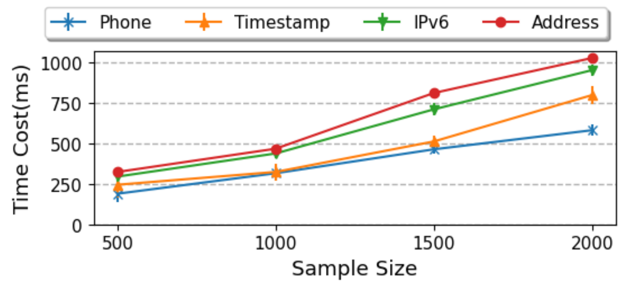


Figure 3.13: Pattern inference latency.

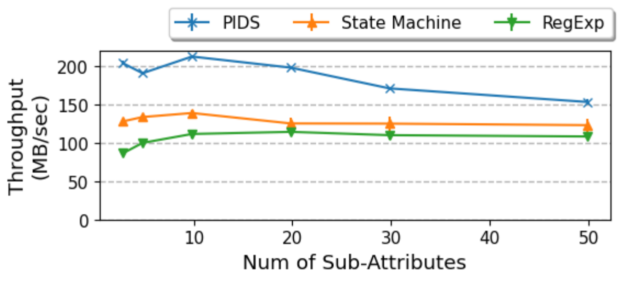


Figure 3.14: Sub-attributes extraction performance.

Only on the address attribute, does PIDS have a slightly worse throughput than Gzip. A cost breakdown shows that when the number of sub-attributes increases, more time is spent on decoding each sub-attribute, and decoding string sub-attributes is slower than decoding integer sub-attributes, primarily due to decoding bytes to UTF-8.

We notice that for the timestamp attribute, although 64-bit Int-Fast is 4 times faster than other approaches, 64-bit Int-String is so slow that it is almost invisible in the figure. We saw similar results when executing wildcard predicates on 64-bit integers. The profiling result shows that over 40% of the time is spent on formatting integers to strings. In PIDS, we introduce an optimized integer to string algorithm inspired by the integer constant division algorithm [119], which formats an integer to string 37 times faster than `String.format` in Java. With this algorithm, PIDS-String is only 10% slower than PIDS-Fast and remains competitive against other approaches.

In Figure 3.12, we test the impact of outliers on the string materialization of phone numbers by controlling the percentage of outliers. We show the throughput normalized against having no outliers. We observe a minimal impact on the throughput when the percentage of outliers is less than 1% (shown in the small box). Additional increases to the percentage of outliers create a proportional impact on throughput. Considering that most attributes we observe have less than 1% outliers, this result shows that outliers have a negligible impact.

3.5.3 Pattern Inference and Data Extraction

Applying PIDS on an attribute requires executing the pattern inference algorithm on the attribute, and using the inferred pattern to extract sub-attributes. In this section, we show that PIDS accomplishes these tasks efficiently.

In Figure 3.13, we show the time used to infer a pattern from the attributes while varying the sampling size. We see with sample size of 2000, the inference latency is around 1 second. This latency is negligible for large data loading tasks as it is an one-off operation.

We also study how the sample size affects accuracy in random sampling, measured by the coverage of a pattern on an attribute. Assuming the data in an attributes can be covered by non-overlapping patterns $p_1, p_2, \ldots, p_n$, each with coverage c_i. The pattern PIDS generates will cover p_i when the sample includes at least one record from p_i. With uniform sampling, a sample size $\mathbb{E}(S) = \max(\frac{1}{1-c_i})$ is sufficient to cover all p_i. We experimentally verify that a sample size 500 is sufficient to achieve coverage of 99%, and sample size 2000 can reach coverage of 99.95%, on the 4596 string attributes that PIDS extracts a valid pattern. To further improve coverage, other sampling methods such as adaptive and biased sampling [76] can also be employed to guarantee instances with small numbers also appear in the sample, at the cost of higher inference latency.

It is crucial to have an efficient data extraction algorithm for attribute decomposition. In Figure 3.14, we show a micro-benchmark comparing PIDS's sub-attribute extraction algorithm with the widely used regular expression-based algorithm and a state machine-based algorithm based on recent work on extracting structures from relational attributes [61]. We varied the number of sub-attributes, with each sub-attribute containing five numerical digits and split by a comma. When the number of sub-attributes is smaller than 10, PIDS achieves over 2 times throughput compared to regular expression, and 50% improvement compared to a state machine. The throughput of PIDS diminishes gradually when the attribute length increases, but still outperforms the two competitors by 30-50%.

Next, we compare PIDS's pattern inference and data extraction algorithm against Data-

Table 3.4: Comparison of PIDS and Datamaran on inference accuracy. PIDS finds more patterns than Datamaran on the entire dataset of 9124 string attributes.

Category	**Entire Dataset**	**After Classifier**
Both find a pattern	1758(19.04%)	566 (19.73%)
Both find no pattern	4597(49.89%)	17 (0.59%)
Only PIDS finds a pattern	2841(30.83%)	2285 (79.67%)
Only Datamaran finds a pattern	18(0.22%)	0(0%)

maran [50], a state-of-the-art solution for extracting structural information from log-like datasets. We conduct the comparison from the perspective of both `correctness`: being able to generate accurate patterns from given attributes, and `time efficiency`: better throughput on pattern inference and data extraction. We evaluate an open-source implementation from the authors.

To compare the accuracy of pattern inference, we apply both PIDS and Datamaran to 1) the entire dataset of 9124 string attributes, 2) the dataset of attributes marked as sound by the classifier we introduced in Section 3.3.3. We mark an extracted pattern as valid if it (1) contains more than one sub-attribute and (2) has more than 50% coverage. In Table 3.4, we show that PIDS manages to find a pattern on 2841 attributes(30% of total) that Datamaran does not, while Datamaran only find 18 patterns that PIDS does not. After applying the classifier, PIDS works even better to recognize 2285 patterns that Datamaran does not. Besides, this result also cross-validates the effectiveness of the classifier: it manages to filter out most attributes that neither PIDS nor Datamaran extract patterns. We note that Datamaran was designed to target log-like data files, which may contribute to some of its inferior performance.

We are also interested in the attributes on which PIDS and Datamaran do not agree. We categorize the 31% data columns on which PIDS finds patterns, but Datamaran does not, into three types, and explain how PIDS handles them.

- **Type 1**: Attributes contains symbols not recognized by Datamaran. Datamaran relies on

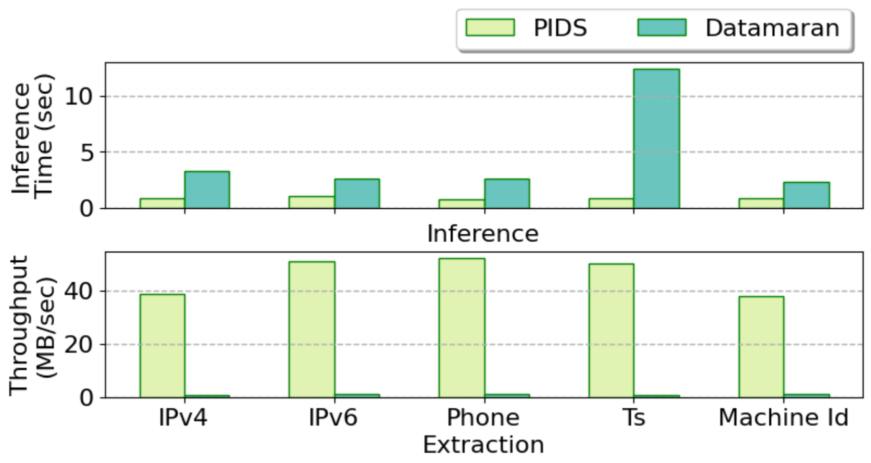


Figure 3.15: Comparison of PIDS and Datamaran on time efficiency. PIDS is 2-15x faster than Datamaran on inference and 20-40x on data extraction.

a hard-coded symbol table to split the records. PIDS overcomes this by inferring common separators from context and works in a truly unsupervised manner.

- **Type 2**: Attributes contain English words or phrases. Some attributes, such as addresses, contain English words or phrases. Datamaran recognizes all spaces in these attributes as separators and generates erroneous patterns. PIDS can recognize words and phrases, and treats them as integral components.

- **Type 3**: Attributes with optional sub-attributes. For example, if an attribute consists of 50% date string, and 50% timestamp string, PIDS recognizes the date sub-attributes as mandatory and the time ones as optional.

There are 18 attributes on which PIDS infers no pattern, but Datamaran does. We manually inspect these attributes and see that these attributes contain only one distinct value. PIDS infers a pattern consisting of only constant from these attributes, and consider such pattern as invalid. We believe this also shows PIDS is more effective in the sense of recognizing meaningful pattern.

We then compare the time efficiency of pattern inference and data extraction between PIDS and Datamaran, using 5 representative string attributes, each containing 1 million records. Datamaran samples 2000 records from the target attribute for inference, and we configure PIDS to follow the same setting. In Figure 3.15, we see that PIDS is 2-15x faster than Datamaran in pattern inference. In the data extraction task, PIDS achieves 20-40x throughput comparing to Datamaran. Considering that PIDS is implemented in Java and Datamaran in C++, this performance boost is significant.

3.6 Conclusion

In this chapter, we introduce PIDS, a technique to extract sub-attributes from relational string attributes in columnar stores, and execute query operators on them. We build a prototype of PIDS based on the Apache Parquet storage format to show that PIDS can improve both compression ratio and query operator execution efficiency. When executing query operators, PIDS is 2-30 times faster than execution on the original string attributes.

CHAPTER 4

SBOOST: SPEED UP QUERY PROCESSING ON ENCODED DATA

Columnar databases such as C-Store[114] and MonetDB[60] have never been playing more important role in this big data era. Different from traditional RDBMS, which stores data in a row-by-row fashion, columnar databases adapt a column-oriented fashion to organize its data. In a columnar database, values from different columns are physically separated, and data in the same column is store consecutively. Such a physical layout allows a faster scan on data columns benefiting from sequential access, and irrelevant columns to be skipped during query to avoid I/O overhead. These features make columnar databases the perfect choice for query and analysis on gigantic datasets.

Columnar store persist similar data from the same column consecutively, allowing efficient encoding techniques to be adopted. Abadi et. al.[2] show that in addition to space saving, executing query on encoded data also exhibits great potential in improving query efficiency. In practice, lightweight encoding algorithms, which trade compression ratio for much faster decompression operations, are more preferred as they allow decoding to be performed on the fly without obvious impact to query performance. Widely used encoding schemes includes bit-packed encoding, dictionary encoding, delta encoding and run-length encoding.

Many previous researches focus on improving the query performance on encoded data. Using new hardware features such as single-instruction-multiple-data (SIMD) instructions is among the most promising techniques. Willhalm et. al. [122] demonstrates a new algorithm using 128 bit SIMD instructions to decode 4 bit-packed integers in parallel. Polychroniou et. al.[101] propose using SIMD to speed up selection scan. Variations of encoding schemes are also developed to further explore the potential of SIMD processors. BitWeaving encoding[87] and BP-128[83] are variations of bit-packed encoding. SIMD-PFOR[83] is a variation of patched encoding. These variations all exhibit significant better performance comparing to

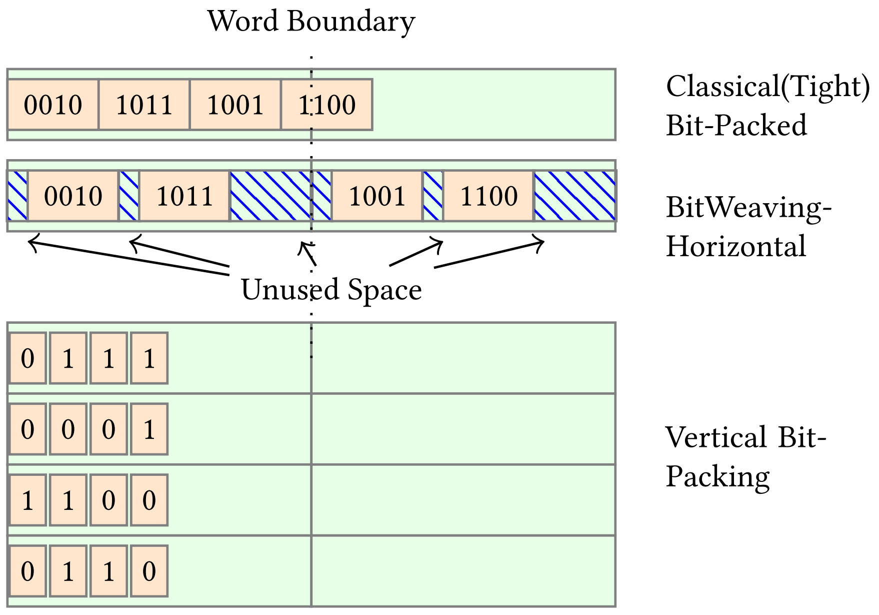


Figure 4.1: Variations of Bit-packed Encoding

corresponding scalar version and demonstrate that the method of using SIMD to speed up encoding/decoding operations in database systems has great potentials.

However, most of these latest algorithms work only on customized variations of encoding schemes that either need extra space in storage format, or requires data to be re-organized in special format, making them space-inefficient and incompatible with standard encoding specifications. We take BitWeaving as an example. In standard "tightly packed" bit-packed encoding, number are packed close to each other. There's no separator between entries, and entry may cross word boundaries. One of the variation format BitWeaving proposes, named BWH, requires a separator bit between entries, and all entries resides in 64-bit words. This can lead to a space waste of at most 30%. Another variation, BWV pack data tightly, yet requires data to be stored vertically instead of horizontally, e.g, adjacent bits in same entry are separate into adjacent words. These differences are visualized in Figure 4.1. In addition to space wasting, converting existing data that are already encoded with standard encodings to the new storage format is time-consuming and impractical considering the enormous amount of existing datasets.

To fill the gap, we propose several novel SIMD-based algorithms for fast scanning /

decoding data stored in standard encoding format including bit-packed encoding, run-length encoding, delta encoding and dictionary encoding. Our scan algorithms work directly on encoded data, efficiently skipping decoding process, saving both CPU effort and memory space. To the best of our knowledge, we are the first to propose SIMD algorithms for standard delta encoding, run-length encoding and dictionary encoding.

We implement these algorithms in SBoost, a columnar datastore based on Apache Parquet. SBoost works on standard encoding schemes widely used in real-world, thus is readily available for existing industrial datastore, yet it outperforms existing solutions by at least a order of magnitude. SBoost is able to achieve a throughput of over 10 billion numbers per second with a single thread, and 40 billion numbers with multi-thread in our experiment environment. Moreover, we have shown that SBoost has great potentials of improve TPC-H query efficiency for both on-disk and in-memory queries.

The contributions of this paper include the follows.

- **Fast Table Scan on Bit-packed encoded data.** During a table scan, predicates such as equality and range search will be applied on data to obtain a comparison result. Previous method requires data to be either fully or partially decoded before the comparison can be performed. We propose a fast SIMD-based table scan algorithm on bit-packed data. The new vectorized algorithm allows executing predicates directly on encoded data, effectively skipping entire decoding process and thus achieve ultra-fast scanning speed.

- **Parallel Decoding and Table Scan for Delta encoded data.** Decoding delta encoded data involves an iterative add operation through all data entries. We introduce a new vectorized algorithm for decoding delta encoded value, and further support efficient predicate execution on the decoded data.

- **Fast Table Scan for Run-length and Dictionary encoded data.** Run-length Encoding replace consecutive identical values with one copy of the value followed by the number of repetitions. Dictionary encoding replace long, variable length data with short fixed-length integer code. The output result of these encodings are often bit-packed to further reduce data size[58]. Using query rewriter to convert query on the encoded data to predicates on underly bit-packed data, and utilizing our fast bit-packed scan algorithm, we propose fast SIMD-based table scan algorithms for both run-length and dictionary encoded data.

In Section 4.1, we describe the design and implementation of SBoost framework for table scan and decoding. In Section 4.2, we describe in detail the algorithms proposed for each encoding scheme. Section 4.3 demonstrates the experiments conducted on various dataset and analysis of results. Section 4.4 conclude our contribution and describe future research possibilities.

4.1 System Design

We build SBoost, a columnar data store supporting SIMD-based fast table scan based on Apache Parquet[13], the prevailing open source columnar store.

Figure 4.2 briefly depicts Parquet's storage data structure. Every Parquet file is comprised of multiple column chunks, each consists of consecutive binary data buffers storing encoded column data. Column chunks also contains a zone map, providing for each column contained in the chunk the max/min value. SBoost utilizes Parquet code to load file from disk into memory and locates data buffers for indicated columns, which are stored off java heap memory. It then invoke SIMD algorithms, which are implemented in C++, through JNI.

SBoost defines two APIs **dataScan**, and **decode** for each encoding scheme. **dataScan** executes a predicate on encoded column, and output a bitmap indicating values satisfying the predicate. **decode** decodes data to ready-for-output format.

For columns appears only in select but not in project, SBoost applies **dataScan** directly

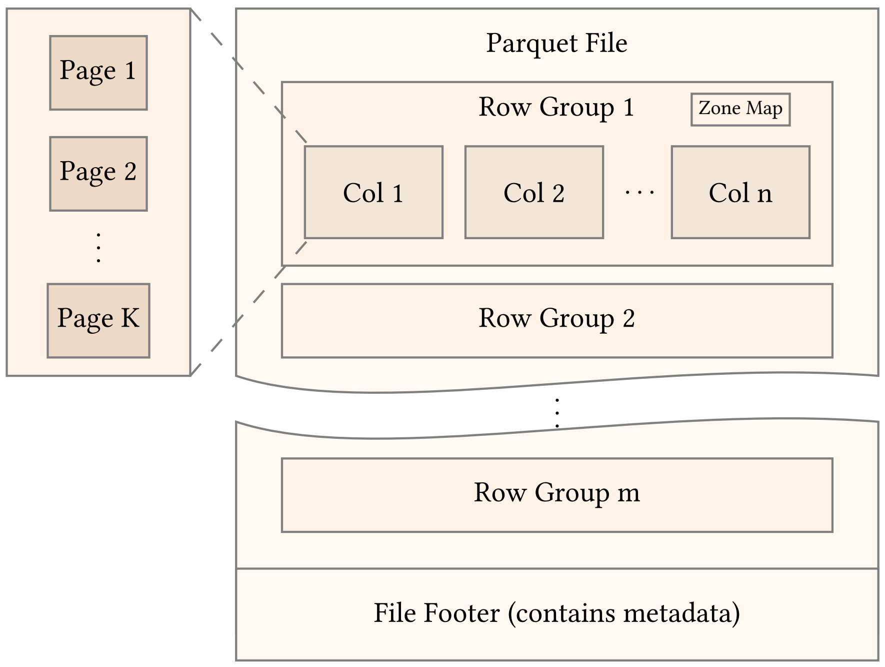


Figure 4.2: Parquet Columnar Store Format

on encoded data buffer to generate the bitmap, which can be further used to filter other columns. Most previous methods decode data before they can be feed to predicate, which incurrs both CPU overhead and unnecessary Java object creation. SBoost provides highly parallelized algorithms involving minimal decoding operations greatly reducing both CPU and memory consumption.

For columns appears only in project but not in select, SBoost executes **decode** on them. SBoost designs some novel algorithms utilizing SIMD parallelization to speed up decoding process.

For columns involved in both select and projection, SBoost first use **dataScan** to generates bitmap on the column, and use the bitmap result to efficiently perform data skipping, saving time for decoding operations on unmatched data.

Figure 4.3 describes how queries are executed by SBoost.

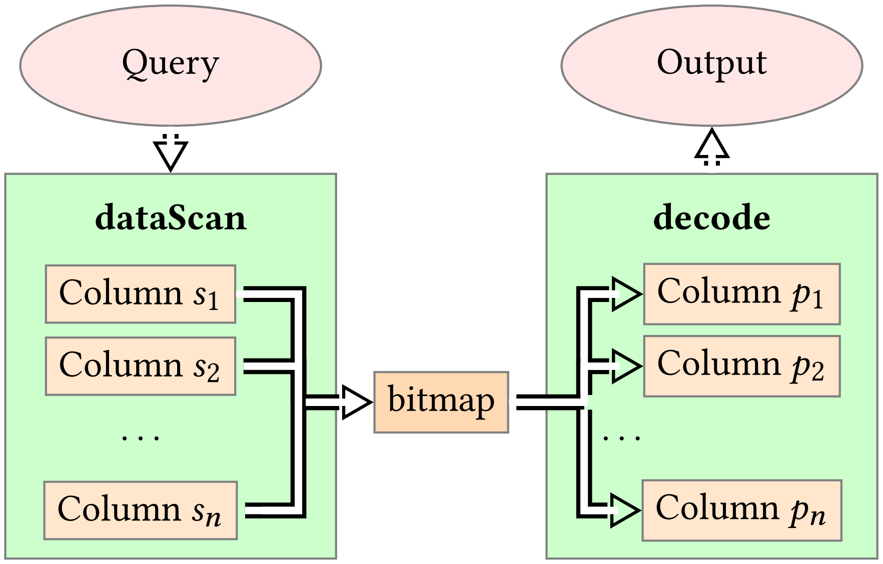


Figure 4.3: Query Execution in SBoost

4.1.1 Operator for Predicate Execution

SBoost supports common predicates including equal / not equal / greater than / less than and their logical combinations. We implement these predicates using two operators: `equal`, which tests whether the target is equal to a given value a, and `less`, which tests whether the target is less than a given upperbound a. These operators take as input the encoded data, and output bitmap.

It is easy to see that together with zone map, all predicates and their combinations can be implemented using these two operators with simple logical operations. For example, less-equal$(x, a) = x \leq a$ can be obtained by $less(x, a) \mid equal(x, a)$, and $range(x, a, b) = a \leq x < b$ can be obtained by $less(x, a) \oplus less(x, b)$. When introducing `dataScan` algorithms, we will focus on describing how we implement `equal` and `less` operators.

4.2 Algorithm

In this section, we detail the SIMD algorithms we design for each encoding scheme to speed up predicate execution and decoding on encoded data.

In subsequent sections, we use uppercase letters to denote SIMD words and lowercase letters for scalars. We use subscripts to indicate elements in SIMD words. E.g., for a SIMD word A, we use $A_0, A_1, \ldots, A_n$ to denote the data entries in it, in small-endian fashion. Entry size varies and will be clarified upon mentioning.

4.2.1 Data Scan for Bit-Packed Encoded Integer

In this section, we introduce our algorithm using AVX-512 for `dataScan` on bit-packed encoded integer. It also serves as the foundation of some subsequent algorithms.

Preprocessing

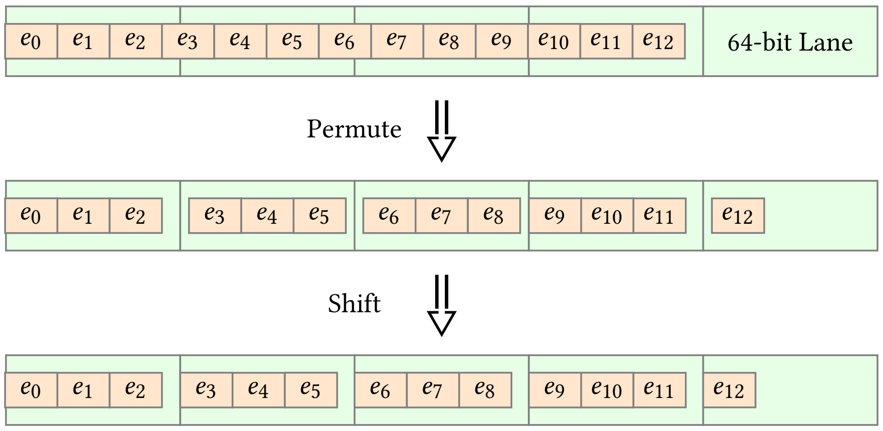
The first step of our algorithm is loading encoded data in 512-bit SIMD word, and align them to 64-bit lanes. This is demonstrated in Figure 4.4. We use `_mm512_permutex2var_epi8` to reorder bytes in 512-bit lane, sending bytes belonging to each entry into corresponding 64-bit lanes, then use `_mm512_srlv_epi64` to shift data to be aligned to lane boundary. For efficiency, the permute and shift instruction used are pre-computed to save run-time effort.

The purpose of this operation is to get data ready for the arithmetic operation we are going to perform in the next step. Intel only provide arithmetic instruction within 64-bit lane. This forces us to align data into 64-bit lanes, greatly limit the number of entries we can process.

As an alternative solution to the problem, we propose an software implementation of 512-bit arithmetic operations using AVX-512. The detail is described in Section 4.A. Currently, the algorithm's performance suffers from lacking of direct hardware support. However, our simulation result show that if this instruction is provided in the future, the pre-processing step can be skipped and we can further improve the efficiency of SBoost's bit-packed scanning algorithm. We present simulation results of this variation and more discussion in Section 4.3.

Equal Operator

Given a SIMD word X containing n entries, each consisting of e bits, and a scalar a, the `equal` operator checks how many entries in X are equal to a. Let M be the most



e_0 e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12}
64-bit Lane
Permute
Shift

Figure 4.4: Load Bit-Packed Entry into 64-bit Lanes

significant bit (MSB) mask that has 1 at the MSB of every entry, and 0 everywhere else, e.g., $\forall i, M_i = 1 \ll (e-1)$, A a SIMD word having every entry equals a, e.g. $A_i = a$. The algorithm computes

$$D = X \oplus A$$
$$R = D \mid ((D \,\&\, {\sim}M) + {\sim}M)$$

and return R as a sparse bitmap containing equality test result in the MSB of each entry.

$$X_i = a \iff (R_i)_{msb} = 0 \tag{4.1}$$

We demonstrate how this algorithm works with an example. Let X be a SIMD word containing two 3-bit entries $\{X_1 = 3, X_2 = 5\}$, and a be 3, we have $X = 0b101011, A = 0b011011$. The MSB mask $M = 0b100100$. Applying the computations above, we obtain $R = 0b101000$. The 6th bit (e.g., MSB of X_2) of R is 1, meaning that X_2 fails the equality test. The 3rd bits (e.g., MSB of X_1) of R is 0, meaning that X_1 passes the equality test.

The algorithm checks whether $x = a$ by examining if $d = x \oplus a = 0$. Let d_{rb} be the remaining bits in d excluding MSB, $d_{rb} = d \,\&\, {\sim}m$, $d \neq 0$ if and only if one of the following is true

- $d_{msb} = 1$
- $d_{rb} \neq 0 \implies d_{rb} + {\sim}m$ generates a carry
  $\implies ((d \mathbin{\&} {\sim}m) + {\sim}m)_{msb} = 1$

Let $r = d \mid ((d \mathbin{\&} {\sim}m) + {\sim}m)$, we see

$$x = a \iff d = 0 \iff r_{msb} = 0$$

The pseudo-code is shown in Algorithm 3.

Algorithm 3 Equal Operator on bit-packed Integer with AVX-512

```
function EQUAL(X, a)
    A = build(a);
    ▷ build will fill each entry with given value
    NM = build((1 ≪ e -1)-1);
    D = _mm512_xor_si512(X, A);
    DAM = _mm512_and_si512(D, NM);
    DAM = mm512_add_epi64(DAM, NM);
    return _mm512_or_si512(D, DAM);
end function
```

Less Operator

The `less` operator takes a SIMD word X and a scalar a, determining whether for each entry $X_i \in X, X_i < a$. We construct M and A in the same way as described above, and compute

$$U = (X \mid M) - (A \mathbin{\&} {\sim}M)$$
$$R = ({\sim}\, A \mathbin{\&} (X \mid U)) \mid (X \mathbin{\&} U)$$

then return R as a sparse bitmap satisfying

$$X_i < a \iff (R_i)_{msb} = 0 \tag{4.2}$$

The algorithm checks whether $x < a$ by examining if one of the following cases happens

- $x_{msb} = 0$ and $a_{msb} = 1$
- $x_{msb} = a_{msb}$ and $x_{rb} - a_{rb}$ causes a carry

In the first case,

$$x_{msb} = 0 \text{ and } a_{msb} = 1 \iff (a \mathbin{\&} {\sim}x)_{msb} = 1 \tag{4.3}$$

In the second case, let $u = (x \mid m) - (a \mathbin{\&} {\sim}m)$

$$x_{msb} = a_{msb} \iff [{\sim}(x \oplus a)]_{msb} = 1 \tag{4.4}$$

$$\begin{aligned} & x_{rb} - a_{rb} \text{ generates a carry} \\ \iff & (x \mathbin{\&} {\sim}m) - (a \mathbin{\&} {\sim}m) \text{ generate a carry} \\ \iff & [(m + x \mathbin{\&} {\sim}m) - (a \mathbin{\&} {\sim}m)]_{msb} = 0 \\ \iff & [(x \mid m) - (a \mathbin{\&} {\sim}m)]_{msb} = 0 \\ \iff & u_{msb} = 0 \end{aligned} \tag{4.5}$$

Combining the equations above we have

$$x < a \iff (\underbrace{(a \mathbin{\&} {\sim}x)}_{\text{Equation (4.3)}} \mid (\underbrace{{\sim}(a \oplus x)}_{\text{Equation (4.4)}} \mathbin{\&} {\sim} \underbrace{u}_{\text{Equation (4.5)}}))_{msb} = 1$$

Using boolean algebra to simplify the formula, we have

$$(a \mathbin{\&} {\sim}x) \mid ({\sim}(a \oplus x) \mathbin{\&} {\sim}u)) = {\sim}({\sim}a \mathbin{\&} (x \mid c)) \mid (x \mathbin{\&} c) = {\sim}r$$

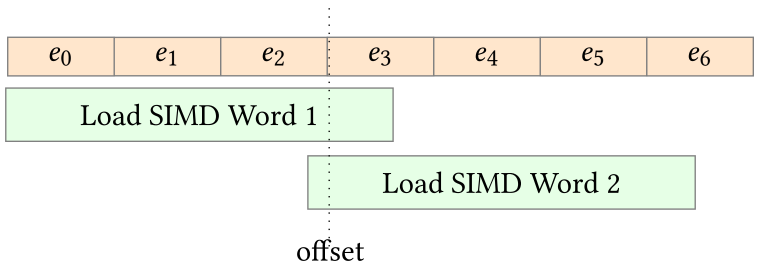


Figure 4.5: Load Entries crossing SIMD Word Boundary

This shows

$$x < a \iff r_{msb} = 0$$

. The pseudo-code is provided in Algorithm 4.

Algorithm 4 Less Operator on bit-packed Integer with AVX-512

```
function LESS(X, a, b)
    M = build(1≪e-1);
    NM = build((1≪e-1)-1);
    A = build(a);
    AORNM = _mm512_and_si512(A, NM);
    NA = _mm512_xor_si512(A, ONE);
    XORM = _mm512_or_si512(X, M);
    U = _mm512_sub_epi64(XORM, AORNM);
    XAU = _mm512_and_si512(X, U);
    XOU = _mm512_or_si512(X, U);
    return _mm512_or_si512(_mm512_and_si512(NA, XOU),
    XAU);
end function
```

Dealing with cross-boundary entries

For entries crossing SIMD word boundary, we use unaligned load instruction to load next SIMD word including that entry. This is demonstrated in Figure 4.5. Entry e_3 crosses SIMD word boundary, so the next SIMD word will start its loading position just before e_3. It should also be noticed that loading address must be byte boundary aligned, so entry e_3 may have a non-zero offset in the second SIMD word, which can be safely handled by our algorithm.

Previous research[122] suggests that unaligned load/store leads to negligible performance

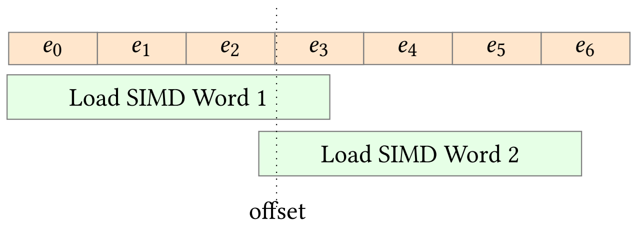


Figure 4.6: Unpack Data as 32 bit Integer

penalties on recent Intel CPUs, and our experiment on latest hardware platform also justify this conclusion.

On platforms where unaligned load/store may lead to unacceptable performance penalties, we propose an alternative solution that simply extract the involved bytes from SIMD register, and use scalar comparison to execute predicates on them. The result is then written back to the corresponding location in the result data stream. Note that we only need to write MSB for the given entry, which can be done with a bitwise operation involving one single byte in memory.

4.2.2 Fast Decoding and Table Scan for Delta Encoded Data

In this section, we introduce our algorithm utilizing Intel's `hadd` instruction to implement a vectorized algorithm to decode delta encoded data. As `hadd` instruction is not yet available on AVX-512, we use AVX2 to implement this algorithm.

Delta encoding store delta between consecutive numbers in tightly bit-packed format. We first use an algorithm similar to what is used in pre-processing step of bit-packed `dataScan` to unpack these numbers as 16-bit or 32-bit integers. This algorithm is also described in Willhalm et. al.'s work[122]. Figure 4.6 illustrates how the unpacking process works. We load bit-packed entries are loaded into SIMD registers, and use byte-level `shuffle` instruction to move each entry to a separate 16-bit or 32-bit lane. As some entries are not aligned with byte boundaries, we next use `shift` instruction to align them to the lane boundary.

With data unpacked as either 16-bit or 32-bit integers in SIMD words, the next step is to compute their cumulative sum in order to obtain original data. We first introduce a `cumsum` function that computes the cumulative sum of each integers in the SIMD register. That is, given SIMD word $B = [B_0, B_1, \ldots, B_n]$, `cumsum` compute A where $A_i = \sum_{k=0}^{i} B_k$. The `cumsum` function for 256 bit SIMD word and 16/32 bit integer is demonstrated in Algorithm 5.

Algorithm 5 Vectorized Cumulative Sum with 256 bit SIMD and 16/32 bit Integer

```
const ZERO = _mm256_set1_epi64(0);
const SHIFT16 = _mm256_set1_epi64x(16);
const MASK16 = _mm256_set1_epi32(0xffff);
const IDX = _mm256_setr_epi32(8,0,1,2,3,4,5,6);
const IDX2 = _mm256_setr_epi32(0,8,2,8,1,4,3,6);
const IDX3 = _mm256_setr_epi32(8,8,8,8,0,1,2,3);
function CUMSUM16(b)
    bp = _mm256_bslli_epi128(current, 2);
    s1 = _mm256_hadd_epi16(current, aligned);
    s2 = _mm256_sllv_epi64(s1, SHIFT16);
    s3 = _mm256_hadd_epi16(s1, s2);
    s4 = _mm256_and_si256(s3, MASK16);
    return _mm256_hadd_epi16(s3, s4);
end function
function CUMSUM32(b)
    bp = _mm256_permutex2var_epi32(b, IDX, ZERO);
    s1 = _mm256_hadd_epi32(b, bp);
    s2 = _mm256_permutex2var_epi32(s1, IDX2, ZERO);
    s3 = _mm256_hadd_epi32(s1, s2);
    s4 = _mm256_permute2x128_si256(s3, IDX3, ZERO);
    return _mm256_add_epi32(s3, s4);
end function
```

We illustrate how the 32-bit algorithm works in Figure 4.7, where we use b_{ij} to denote $\sum_{k=i}^{j} b_k$. We first shift the input to left by 32 bits, shifting in 0. As Intel does not provide 256 bit lane shift instruction, we utilize `permute` instruction to achieve this. The following code shift entire 256 lane in input to the left by 32 bits.

```
ZERO = _mm256_set1_epi64(0);

IDX = _mm256_setr_epi32(8,0,1,2,3,4,5,6);

_mm256_permutex2var_epi32(input, ZERO, IDX);
```

We then use `hadd` on the original input b and the shifted input bp to obtain sum of adjacent number pairs. Reordering the result using `permute` instruction, and use `hadd` one

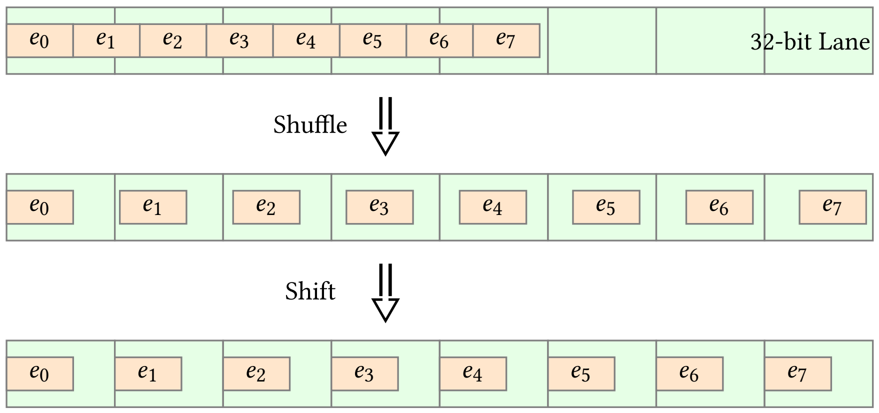


Figure 4.7: Use `hadd` to compute 32-bit Cumulative Sum

more time gives us partial sum of at most 4 consecutive numbers.

Finally, we reorder the `hadd` result and perform a 32-bit add, obtaining cumulative sum for each index. It can be seen that the result is stored in a inverse sequence. We will leave the result as is when performing `dataScan`, and only restore the correct order in `decode`. This saves one `permute` instruction and increase throughput. 16-bit `cumsum` works in a similar fashion and we skip the description here for succinctness.

With `unpack` and `cumsum` function, it is now straight-forward to implement `decode` and `dataScan` function for delta encoded data, which is shown in Algorithm 6 and Algorithm 7. We describe the 32 bit version here, and 16 bit version can be implemented in a similar manner.

When performing `decode`, we first load data into SIMD registers, and unpack them as 32-bit integers, then use `cumsum` to compute the cumulative sum in current register. The variable `sum` tracks the cumulative sum of all numbers that have been scanned so far. Adding `sum` to `cumsum` result then gives us the decoded value, as well as the new value of `sum`. Finally, the decoded number is reordered to correct sequence using `permute` instruction and output to storage.

`dataScan` just use the decode process we described above, and SIMD comparison instruction to execute predicate. The result is a dense bitmap and can efficiently be used in future operations.

Algorithm 6 `decode` for Delta Encoded 32 bit Integer

```
const INV = _mm256_setr_epi32(3, 2, 1, 0, 7, 6, 5, 4);
function DECODE(stream)
    sum = 0;
    while stream.hasNext do
        words = unpack(stream.next);
        for word in words do
            cumsum = CUMSUM(word);
            decoded = _mm256_add_epi32(cumsum, sum);
            sum = _mm256_extract_epi32(decoded, 4);
            inverted = _mm256_permutexvar_epi32(decoded, INV);
            OUTPUT(inverted);
        end for
    end while
end function
```

Algorithm 7 `dataScan` for Delta Encoded 32 bit Integer

```
function DATASCAN(stream, predicate)
    sum = 0;
    while stream.hasNext do
        decoded = DECODE(stream.next);
        if predicate == EQUAL then
            scanRes = _mm256_cmp_epi32_mask(decoded,
            predicate.val, _MM_CMPINT_EQ);
        else if predicate == LESS then
            scanRes = _mm256_cmp_epi32_mask(decoded,
            predicate.val, _MM_CMPINT_LT);
        end if
        OUTPUT(scanRes)
    end while
end function
```

4.2.3 Data Scan for Run-Length encoded Integer

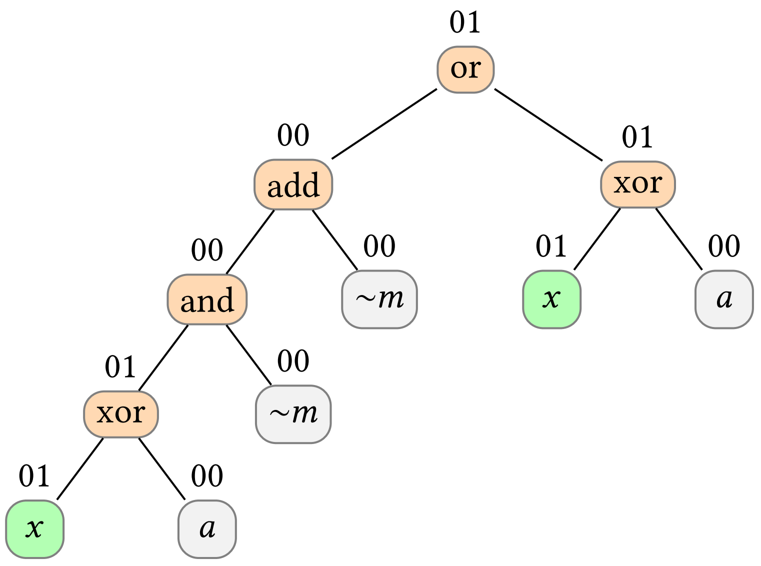
Run-length encoded data comprises of consecutive number pairs `(num, run-length)`. These pairs are then tightly bit packed. `num` and `run-length` can use different bit width, but without loss of generality, we assume that within a storage unit (a page or a block), bit widths for `num` and `run-length` are fixed.

When scanning run-length encoded data, we generate a run-length encoded bitmap. For example, when scanning a run-length encoded data sequence $\{105, 2, 339, 4, 242, 1, 132, 8\}$ to look for numbers less than 200, the output is $\{1, 2, 0, 4, 0, 1, 1, 8\}$. This kind of bitmap had been widely adopted in previous works [49, 57, 125, 84].

We utilize the bit-packed `dataScan` algorithm described in Section 4.2.1 to generate this run-length bitmap. The basic idea is to execute predicates on `num` fields, while leaving `run-length` fields unchanged. We show that by setting bits corresponding to run-length fields to 0 in all parameters used in Equation (4.1) and Equation (4.2), the run-length fields from input will be preserved during the computation of bit-packed `dataScan` algorithm.

In Figure 4.8, we draw the operation tree for `equal` operator in bit-packed dataScan. The numbers above each nodes shows how the bits from input change after each operation. It is clear from the figure that if all parameters (the gray blocks in figure) have their run-length fields set to 0, the run-length values from input will be preserved. In addition, in previous sections we have shown that the `add/sub` operations used in the algorithm does not generate carry bits crossing entry boundary. We conduct the same check for `less` operator. The operation tree is shown in Figure 4.9. Similarly, we notice that the bits in input will be preserved when all parameters have their bits set to 0. This shows that by leaving the run-length fields as 0 in all parameters used in Algorithm 3 and Algorithm 4, we are able to get a run-length bitmap generated by applying the bit-packed dataScan algorithm.

Some complex operators may be affected, though. For example, `range` operator can be obtained by $range(x, a, b) = less(x, a) \oplus less(x, b)$. Per our analysis above, $less(x, i)$ will preserve the run-length fields in input. Thus both operands of `xor` will have the same value



$$((x \oplus a)\ \&\ {\sim}m + {\sim}m)\ |\ (x \oplus a)$$

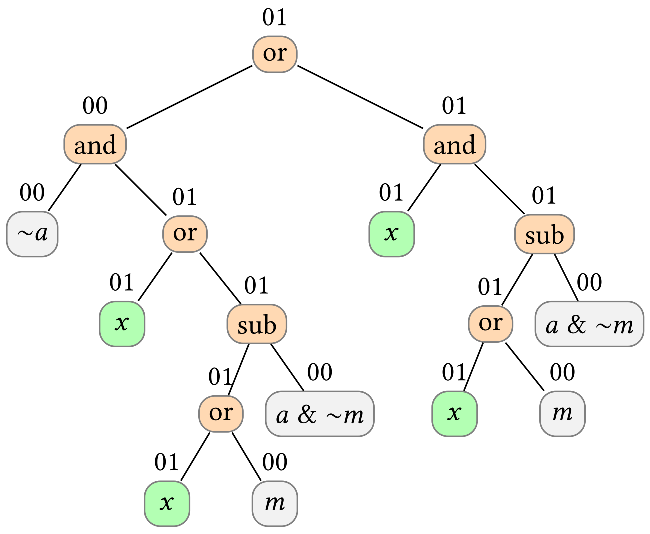
Figure 4.8: Operation Tree for `equal` operator

in their run-length fields, and leads to 0 after the operation. To solve such problem, we simply rewrite $range(x, a, b) = less(x, a) \oplus (less(x, b)\ \&\ \underbrace{11\ldots1}_{\text{value fields}}\ \underbrace{00\ldots0}_{\text{run-length fields}}\ )$. That is, add a mask that erase run-length fields from the right operand. It can be easily seen that this allows run-length fields data to be preserved during *range* opeartor, while comparison result is unaffected. Similar technique can be applied on other operators.

4.2.4 Data Scan for Dictionary Encoded Data

In this section, we describe fast `dataScan` algorithm for dictionary encoded data, again using `dataScan` of bit-packed integers. The basic idea is that by rewriting predicates on dictionary encoded data to predicates on bit-packed data, we covert an predicate on dictionary encoded data into a predicate on bit-packed data, which can be efficiently processed using the algorithm described before.

Formally, if we use dictionary d :`(key)->code` to encode list $[a_i]$ to $[d(a_i)]$, then for any predicate p on a_i, we show that it is always possible to build a predicate p' on $d(a_i)$, satisfying



$$(\sim a \ \& \ (x \mid (x \mid m - a \ \& \sim m))) \mid (x \ \& \ (x \mid m - a \ \& \sim m))$$

Figure 4.9: Operation Tree for `less` operator

$\forall i, p(a_i) = p'(d(a_i))$.

For `equal` operator

$$p(a_i) = \mathbb{I}[a_i = \alpha]$$

, we define

$$p'(b_i) = \mathbb{I}[d(a_i) = d(\alpha)]$$

It is easy to verify p' satisfies the condition above as d is bijective. Non-equality cases can be processed in the same way.

For `less` operator, we use order-preserving dictionary [19]. An order-preserving dictionary d_o makes sure the codes follow the same order as their corresponding keys, e.g., $a_i > a_j \iff d_o(a_i) > d_o(a_j)$. Thus assume d_o is order-preserving, `less` predicate

$$p(a_i) = \mathbb{I}[a_i < floor]$$

can be rewritten as

$$p'(d(a_i)) = \mathbb{I}[d(a_i) < d(floor)]$$

.

In addition, for string types, order-preserving dictionary support a prefix lookup operation `lookup(prefix)-> (mincode, maxcode)`, which allows us to effectively support prefix scan by rewriting it as a range scan.

Now we have shown that all predicates on a dictionary encoded data can be rewritten as predicates on bit-packed data, it is then straight-forward to execute them using bit-packed `dataScan` algorithm.

4.3 Experiment

We use an experiment platform equipped with 2 Intel(R) Xeon(R) Silver 4116 CPU@2.10GHz, and 190G memory. SIMD codes are compiled using GCC 5.4.0, with `-O3` flag. We implement SBoost with Java / Scala, and access the SIMD-based procedures written in C++ using JNI. Software platforms used in the experiment include JDK Version 1.8.0_152, Scala Version 2.12.4, Apache Parquet version 1.9.0.

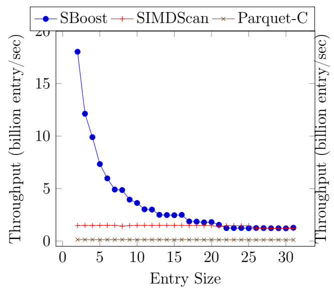
4.3.1 Microbenchmark

In this section, we evaluate SBoost's dataScan/decode algorithm performance on in-memory data, with single thread.

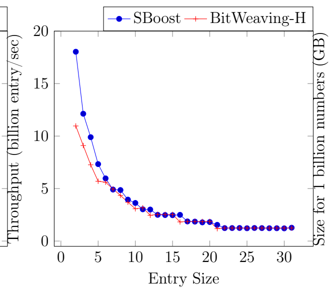
DataScan of Bit-Packed Integer

Figure 4.10 shows the experiment result on dataScan of bit-packed encoded integers. In fig. 4.10a, we compare SBoost with Willhalm's SIMDScan algorithm [122, 83], rewritten in AVX-512, and Parquet's implementation. We can see that both algorithms outperforms Parquet's highly optimized scalar algorithm by over one order of magnitude. Moreover, SBoost outperforms SIMDScan by another one order of magnitude on smaller entry size.

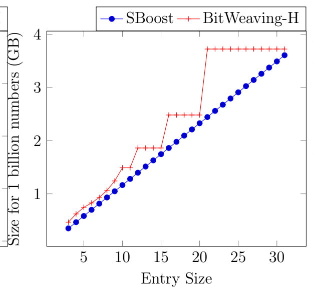
SBoost achieves higher efficiency on smaller entry size primarily due to better parallelization. While SIMDScan uses one 32-bit lane for each bit-packed entry, SBoost can fit



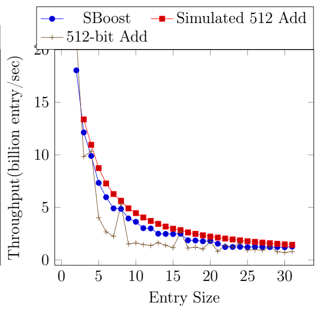
(a) dataScan Performance



(b) Throughput vs. BitWeaving-H



(c) Space Saving vs. BitWeaving



(d) Using 512-bit Arithmetic Operation

Figure 4.10: SBoost Performance on Bit-Packed Data

more than one entry in each 32-bit lane and compare them in parallel, thus achieves higher throughput for smaller entry size. For entry size of 3, SBoost is able to achieve over 9x performance compared to SIMDScan (over 12 billion numbers per second). When entry size increases to over 22 bits, one 64-bit lane can only accommodate at most 2 entries, which is the same as SIMDScan. Consequently, the throughput drops to the same level as SIMDScan.

We also compare SBoost to BitWeaving-H[87], rewritten in AVX-512. BitWeaving does not use tightly bit-packed encoding. Instead, it uses an encoding scheme that trades storage space for efficient processing. Data in BitWeaving-H is stored in 64-bit lane, with one bit left empty between each entry. In Figure 4.10b, we show that SBoost outperforms BitWeaving-H by 10∼25% for small entry size, again due to higher parallelization. In Figure 4.10c, we demonstrate the space usage to storing 1 billion numbers in tightly bit-packed format and in BitWeaving-H format. For smaller entries, SBoost is faster than BitWeaving-H. For larger entries, SBoost achieves similar performance as BitWeaving-H but use much less space (at most 30% space saving).

As a conclusion, we see that SBoost does not only outperforms previous algorithms on tightly bit-packed integers, it also achieves better performance than variation of bit-packed encodings. This shows using SBoost with tightly binary-packed integer is the best choice for both data scan speed and storage efficiency.

Next, we propose a study result that is able to further improve the efficiency of this algorithm, and may inspire future research. As is described in Section 4.2.1, our algorithm need some extra pre-processing step to align data to 64-bit lanes, due to the limitation in arithmetic operation of Intel CPU. This step does not only costs extra CPU cycles, but also limits the number of entry we can process in parallel. For example, with entry size equals 13, we can fit 39 entries into 512-bit lanes, but only 32 entries in eight 64-bit lanes.

To study the impact of this problem, we implements a software AVX-512 add/sub instruction as is described in Section 4.A and test its performance. We also use a hardware simulator to evaluate the throughput if Intel has this 512-bit arithmetic instruction supported and it takes the same cycles as 64-bit arithmetic operation. The result is demonstrated in Figure 4.10d.

We notice that when using our software implementation of 512-bit arithmetic operations, throughput decreases to around 50∼70% of SBoost due to the extra effort we employ to manually handle cross-lane carry bits. However, if this instruction is supported by hardware, we can gain another 15∼20% performance improvement compared to SBoost, which is 20x to SIMDScan, and nearly 2x to BitWeaving-H. This shows that our algorithm still have great potential to further improve throughput and we will explore the possibility of using dedicated hardware for a hardware implementation of this instruction to verify this in the future.

Finally, we conduct performance evaluation on `dataScan` for dictionary encoded data. In previous section, we mention that for `dataScan` on dictionary-bitpacked encoded data, we use a order preserving dictionary and rewrite query to convert the operation into a `dataScan` on bit-packed encoded integers. We omit the result here for succinctness as it is identical to what is shown in Figure 4.10a. As a summary, SBoost is able to achieve nearly two orders of magnitude throughput comparing to Parquet, and can process at most 10 billion bit-packed entries per second.

Decode of Delta Encoded Integer

We report our experiment on SBoost's decode algorithm for Delta encoding. We compare

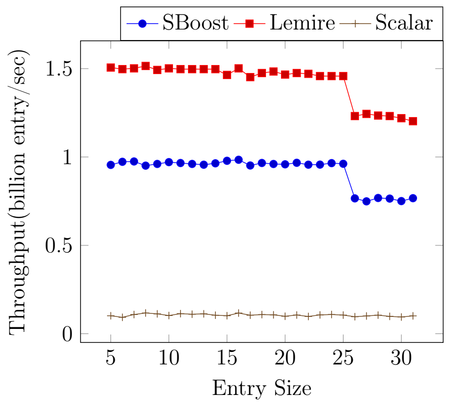


Figure 4.11: SBoost Performance on Delta Encoded Data

our algorithm with the following methods.

- Scalar Decoding algorithm, which extract entries and compute the cumulative sum item by item.
- Lemire's vectorized delta algorithm[83], rewritten using AVX2. Parquet uses a similar encoding format as is used in this algorithm.

Lemire's algorithm chooses not to use standard delta encoding in consideration of decoding efficiency. Instead of computing delta between consecutive numbers, they compute delta between numbers with index difference of i, where i is the number of integers in SIMD word. For example, with Lemire's original implementation using SSE(128-bit SIMD), they compute delta between numbers whose index differ by 4. E.g., a sequence $\{x_0, x_1, \dots, x_7\}$ will be stored as $\{x_0, x_1, x_2, x_3, \delta_0 = x_4 - x_0, \delta_1 = x_5 - x_1, \dots\}$. This allows them to use `_mm_add_epi32` to add all four numbers with one instruction when performing decoding.

The improvement of decoding speed comes at a cost of storage space. Lemire reports that when using a vectorized delta of their vectorized delta is on average four times larger than standard delta, and result in the storage cost up by 2 bits. When migrating to AVX-512, one can compute 16 integers with a single instruction, however, this extra cost also increase to 4 bits.

In Figure 4.11 we compare the throughput of these algorithms. Not surprisingly, Lemire's algorithm performs best as it only execute a single add instruction for each 8 numbers, and reachs a throughput of around 1.5 billion numbers per second. However, SBoost also manages to maintain a performance of 1 billion numbers per second, while they both outperforms the scalar method by one order of magnitude. This shows that if storage size is not a concern, one can choose Lemire's algorithm. If we need to process standard delta encoding or save storage space, SBoost is still the choice to go.

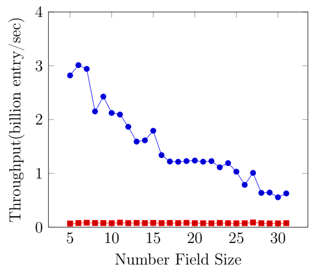
DataScan on Run-Length Encoded Integer

Next, we report our experiment result of SBoost's `dataScan` performance on run-length encoded integer. We vary both number field size and run-length field size, and have the experiment result reported in Figure 4.12. Based on analysis on a real-world dataset collection containing over 15000 columns we have seen that over 99% dataset have an average run-length of less than 2^10, and thus focus our study on small entry sizes. It can be seen that while changing field size makes no difference to Parquet, SBoost again benefits much when dealing with small entries.

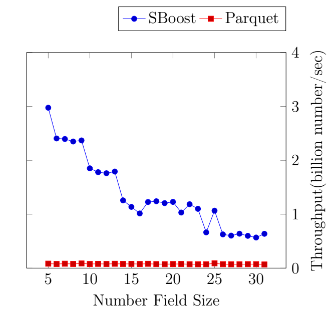
With run-length field size of 5, SBoost achieves in average 20x and at most 40x throughput compared to Parquet, and can process in average 2 billion entries per second. When a larger run-length field size(15) is used, SBoost performance degrades due to less entries can be processed in parallel. Even though, it still achieves an average throughput of 1 billion entries per second.

Even with extremely large run-length field size(26), which means only 8 to 16 entry can fit in a AVX-512 word, SBoost still manages to process 0.5 billion entries per second, which provides a lower bound of the algorithm's throughput.

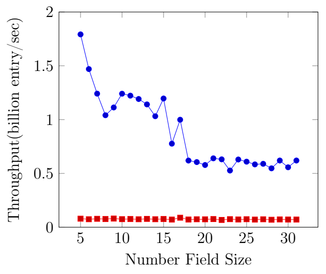
Overall, we have shown that SBoost's algorithms has obvious advantages comparing to both industrial and academic state-of-art competitors, and exhibit great potential in speeding up database queries.



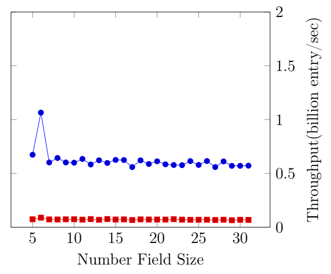
(a) Run-length Field size 5



(b) Run-length Field size 7



(c) Run-length Field size 15



(d) Run-length Field size 26

Figure 4.12: SBoost Performance on Run-Length Encoded Data

4.3.2 TPC-H Performance

In this section, we demonstrate our experiment of using SBoost to speed up TPC-H queries, and compare the performance result with Parquet. As SBoost aims at improving table scanning/decoding speed, we choose Q1 and Q6 from TPC-H queries that only involves select/project operators. We use TPC-H data generator to generate test dataset with scale varied from 1 to 30, and read files from both disk and memory (ramdisk).

The relational algebra of Q1 is (ignoring aggregate function for succinctness)

$$\pi_{\substack{\text{extend\_price}\\ \text{discount}\\ \text{line\_status}\\ \text{quantity}\\ \text{tax}}}(\sigma_{\text{shipdate}<\alpha}(\text{lineitem}))$$

The relational algebra of Q6 is

$$\pi_{\text{extend\_price,discount}}(\sigma_{\substack{\text{quantity}<\alpha\wedge \\ \text{shipdate}\in(\beta_1,\beta_2)\wedge \\ \text{discount}\in(\gamma_1,\gamma_2)\wedge \\ \text{extend\_price}\in(\eta_1,\eta_2)}}(\text{lineitem}))$$

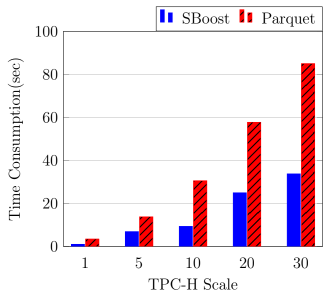
We encode string columns `shipdate, line_status`, and double columns `extend_price, discount, tax` with dictionary-bitpacked encoding using a order-preserving dictionary, and integer column `quantity` with bit-packed encoding. We use SBoost `dataScan` to execute predicate on `shipdate`, and `quantity`, and use `decode` to extract `line_status`. SBoost does not provide algorithms for scanning/decoding double data, so we use Parquet's default implementation for double columns.

The experiment results are shown in Figure 4.13. For Q1, the only predicate is on `shipdate` column, which can be executed efficiently with SBoost. In addition, `quantity` can benefit from SBoost's decode function. Not surprisingly, SBoost achieves over 50% percent performance gain. For Q6, there are four columns involved in predicate execution, of which only two (`quantity` and `shipdate`) can be speed up using SBoost. In addition, the projected columns are all of double type thus do not benefit from SBoost. Even with these limitations, SBoost uses only 55% of Parquet's execution time.

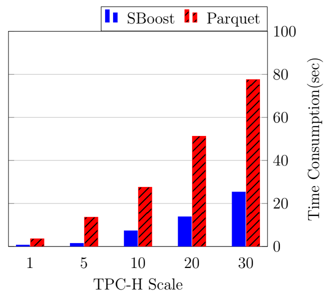
Furthermore, when executing against files stored in ram disk (simulating in-memory database), SBoost has demonstrate even better performance. In the best case, SBoost uses only around 15% time of Parquet to execute a query. We believe this clearly demonstrate SBoost's potential application in both OLAP and in-memory databases.

4.3.3 Scalability

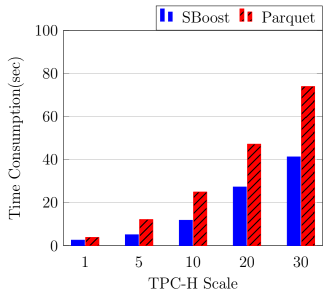
In this section, we study the scalability of SBoost algorithms. It is straight forward to parallelize algorithms we introduced in this paper for bit-packed encoding, run-length encoding and dictionary encoding. We simply split the input/output into multiple slices and process



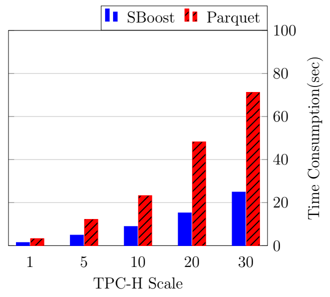
(a) TPC-H Q1 Disk



(b) TPC-H Q1 RAM



(c) TPC-H Q6 Disk



(d) TPC-H Q6 RAM

Figure 4.13: SBoost Speed up TPC-H Queries

each slice with one thread.

For delta encoding, we use a two-pass method. In the first pass, we split input and output into slices as described above, and compute cumulative sum in each slice using the delta-decoding algorithm described before. In the second pass, for each slice, we add to it the sum of last elements from all slices before it. As in this phase, data in each slice has been decoded to 16-bit or 32-bit lanes, the add operation can be done efficiently using `_mm512_add_epi16` and `_mm512_add_epi32`.

Figure 4.14 shows the performance of bit-packed `dataScan` algorithm using multi-threads. Run-length `dataScan` and dictionary `dataScan`, which are based on the same algorithm, exhibit similar patterns.

It can be noticed that multi-threading does benefit the algorithm. Using 16 threads generally brings 4x∼5x throughput comparing to single thread in all cases. However, we also notice that using more than 16 threads does not bring further benefit. For entry size 3, adding more threads causes throughput to drop around 10%. For all other entry sizes,

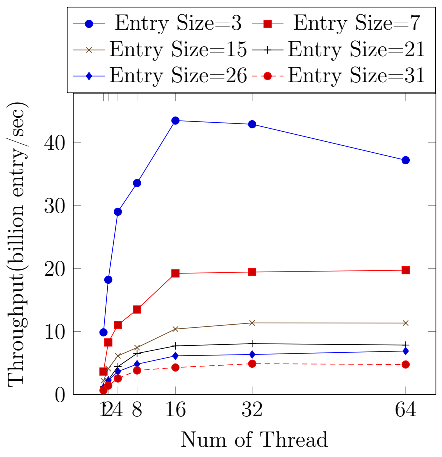


Figure 4.14: Scalability of Bit-packed `dataScan`

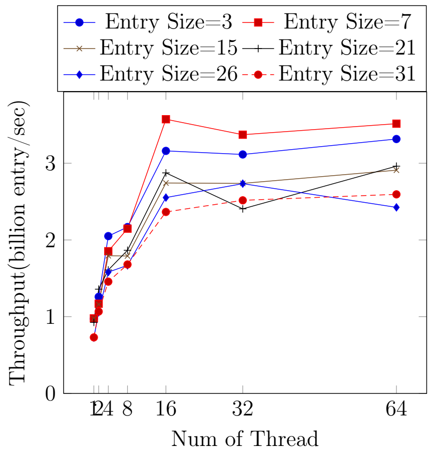


Figure 4.15: Scalability of Delta `decode`

throughput stalls at some plateaus.

The multi-threaded Delta `decode` algorithm, as is demonstrated in Figure 4.15, exhibits a similar pattern. Using more threads helps in the beginning, but no longer has obvious effect after using more than 16 threads. In addition, similar to what we have seen in Section 4.3.1, entry size no longer have obvious impact to throughput.

We have a hypothesis that the performance degrading and plateaus when using more threads is caused by hardware limitation, e.g., available 512-bit registers. AVX-512 defines 32 AVX-512 registers for each CPU, and many instructions involved in our algorithm using 3 registers. With 2 CPU available, we have in total 64 AVX-512 registers. When the number of threads exceeds 21, we cannot guarantee 3 registers for each thread, and some thread need to hang. This hanging caused by resource starvation can also explain the observation that in some cases, performance degrades when using more threads. Due to resource limitation, we leave the proof of this hypothesis to future work.

Nevertheless, this demonstrates that our algorithms scale reasonably well within hardware limit and can make full utilization of latest hardwares.

4.4 Conclusion

Hardware acceleration has always been playing an important role in database research. Among all possible methods, SIMD has exhibited great potential with many advantages such as direct memory access and fused control flow. In this paper, we introduce novel SIMD algorithms for prevalent encoding schemes that supporting predicate execution directly on encoded data. Our algorithms work on standard encodings, requiring no additional storage space or special file format, yet provide lightening processing speed. Our algorithm for bit-packed encoded integer and dictionary-bitpacked encoded integer / string can process over 10 billions numbers per second. Our algorithm for delta encoded integer and run-length encoded integer also achieves a throughput of over 1 billion numbers per second.

We implement these algorithms and build columnar data store SBoost based on Apache Parquet. Our experiment results demonstrate that the new algorithms outperform their counterparts by at least one order of magnitude. It reduces TPC-H query time by over 60% for on-disk queries and over 80% for in-memory queries.

In the future, we have plans to extend this work in several directions. First, we have shown through a simulation result that the performance of our algorithm can further be improved with support from hardware. In addition, we also see that general purpose hardware solutions such as GPU and SIMD can easily be outperformed by dedicated hardware on specific problems. We are interested in building a hardware implementation of our algorithms to further improve efficiency.

Furthermore, our current solution only applies to table scan, leaving joining for future.

While the general case of joining two columns from different tables requires decoding data into memory and making comparison, if the columns to be joined are encoded using same encoding schema, there is possibility that direct comparison between columns can be conducted without decoding.

4.A Implementing 512 bit add/sub operations

Our algorithm use 512 bit arithmetic operations such as add and subtract. However, Intel only provides 64-bit arithmetic instructions. We implement 512-bit arithmetic operations using AVX-512 and describe the detail here. To make the introduction concise, we take add as an example, subtract can be done in a similar fashion.

For a 512 bit number x, we represent the 8 64-bit numbers using $x[i], i \in [0, 7]$. Given two 512 bit number a,b and $r = a + b$, we have the following equations

$$
\begin{aligned}
r[0] &= a[0] + b[0] \\
r[1] &= a[1] + b[1] + r_c[0] \\
r[2] &= a[2] + b[2] + r_c[1] \\
&\vdots \\
r[7] &= a[7] + b[7] + r_c[6]
\end{aligned}
$$

where $r_c[i] \in \{0, 1\}$ represents whether $a[i] + b[i]$ generate a carry, and can be computed by performing unsigned integer comparison between the sum result with either addend.

$$r_c[i] = \mathbb{I}[a[i] + b[i] < a[i]]$$

Noticing that $r[i]$ is either $a[i]+b[i]$ or $a[i]+b[i]+1$, we can precompute both numbers, then selecting from them based on the values of $r_c[i]$. We use `blend` instruction introduced before to optimize the process.

The 512 bit add algorithm is demonstrated in Algorithm 8. In line 1-3 we precompute $nc[i] = a[i] + b[i]$ and $wc[i] = a[i] + b[i] + 1$, where nc mean "no carry" , and wc means "with carry". in line 4-6 we compare nc and wc to a to determine whether a carry bit is generated for each 64 bit add operation. The reason we need to compute carry bits on both nc and wc is as following. If $a[i]+b[i]$ generates a carry, when look at $i+1$ lane, we need to check whether $a[i+1]+b[i+1]+1$ generates a carry, instead of $a[i+1]+b[i+1]$. In line 7, we combine the carry bits as one integer, and use a pre-computed `BLEND_TABLE` to lookup blend instruction. Those magic numbers and details of `BLEND_TABLE` will be described below. Finally, we use blend instruction to select 64 bit integers from nc and wc to construct the result.

Algorithm 8 Optimized 512 bit add

```
1: function ADD_512(a,b)
2:     nc = _mm512_add_epi64(a,b);
3:     one = _mm512_set1_epi64(1);
4:     wc = _mm512_add_epi64(nc, one);
5:     ncval = _mm512_mask_cmp_epu64_mask (0xff, nc,
       a, _MM_CMPINT_LT);
6:     wcval = _mm512_mask_cmp_epu64_mask (0xff, wc,
       a, _MM_CMPINT_LT);
7:     blendIdx = ((wcval & 0x7e) ≪ 6) | (ncval & 0x7f);
8:     blend = BLEND_TABLE[blendIdx];
9:     return _mm512_blend_epi64(nc, wc, blend);
10: end function
```

The blend table stores the correspondance between carry bits and appropriate blend instructions. We use an example to show how this table is computed. Assume there are carries generated at location 0, 2, 3, and 6. We illustrate the situation in Figure 4.16, where "-" means the bit is ignored, and "?" means the bit can be either 0 or 1.

We first notice that the MSB of both *ncval* and *wcval*, corresponding to the highest 64 bit lane can be ignored, as even if there is a carry generated from the lane, no lane will take the carry. Similarly, the LSB of *wcval* can also be ignored as no lower lane can contribute a carry to it. Thus only the lower 7 bit of *ncval* and the middle 6 bit of *wcval* is meaningful.

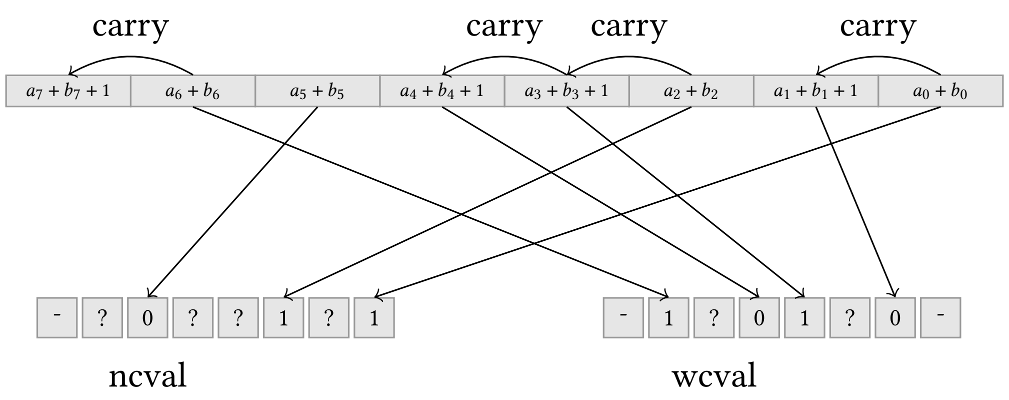


Figure 4.16: Compute blend instruction from carry bits

This gives us the magic number seen in line 7 of Algorithm 8.

It can also be noticed from Figure 4.16 that if a bit is set in *wcval*, the corresponding bit in *ncval* can be ignored and vice versa. So there are in total only 7 effective bits. Instead of go through the bits and determine which one are valid, we concatenate all bits from the two variables as a 13-bit integer $\underbrace{1?01?0}_{\text{from } wcval}\ \underbrace{?0??1?1}_{\text{from } ncval}$. All indices conforming to this pattern will lead to the same value in the blend table.

Finally we need to compute the blend instruction value for this index pattern. From Figure 4.16 it is easy to notice $r[0] = nc[0], r[1] = wc[1], r[2] = nc[2], r[3] = wc[3]$ $r[4] = wc[4], r[5] = nc[5], r[6] = nc[6], r[7] = wc[7]$. The blend instruction corresponding to this index pattern is thus 10011010, where 1 means the value is chosen from wc, and 0 means the value is chosen from nc.

By iterating all possible 2^7 patterns in a similar way, we can compute all 2^{13} entries for the blend table. The code for computing the blend table can be found in Algorithm 9.

Algorithm 9 Compute 512-bit add Blend Table

```
for i = 0 to 8191 do
    wc = (i $\ll$ 6);
    nc = i & 0x7f;
    usenc = true
    result = 0
    for j = 0 to 7 do
        current = usenc? nc:wc;
        if !usenc then
            result |= (1 $\gg$ j)
        end if
        usenc = (current & (1 $\gg$ j)) == 0
    end for
    BLEND_TABLE[i] = result;
end for
```

CHAPTER 5

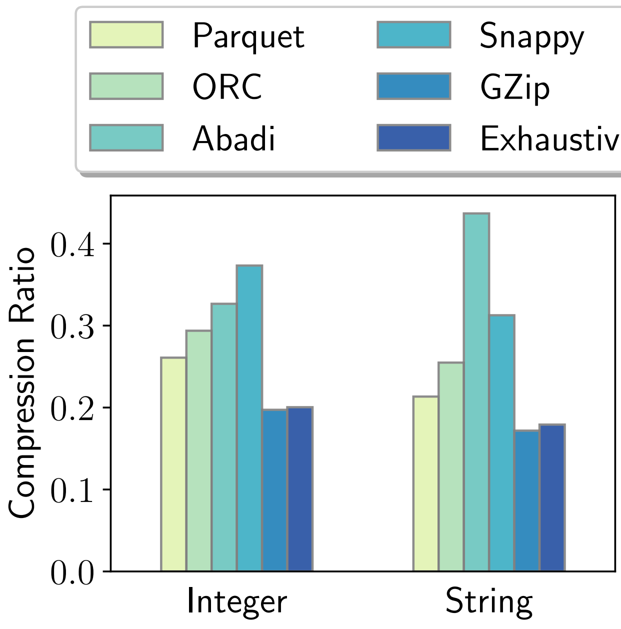
CODECDB: AN ENCODING-AWARE DATABASE

Over the past decade, columnar databases dominate the data analytics market due to their ability to minimize data reading, maximize cache-line efficiency, and perform effective data compression. These advantages lead to orders of magnitude improvement for scan-intensive queries compared to row stores [60, 134]. As a result, academic research [114, 2, 63, 1], open-source communities [13, 12], and large commercial database vendors, such as Microsoft, IBM, and Oracle are embracing columnar architectures.

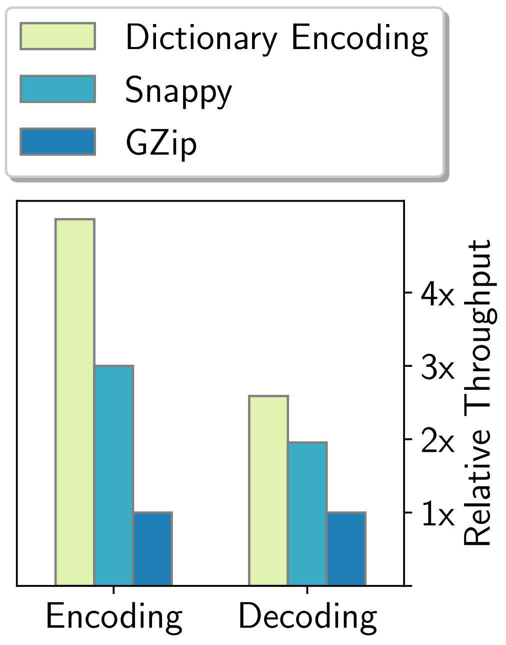
Columnar databases employ compression and encoding algorithms to reduce the data size and improve bandwidth utilization. Both are important for organizations storing data in public clouds. For example, one S&P 500 employee we spoke with disclosed that their monthly cloud costs for storing Parquet files are in the six-figure range. Therefore, encoding data to reduce the storage size makes significant practical sense. Popular encoding schemes include dictionary encoding, run-length encoding, delta encoding, bit-packed encoding, and hybrids. These methods feature a reasonable compression ratio with fast encoding and decoding steps.

Many database systems support the LZ77-based byte-oriented compression algorithms [131], such as Snappy [54] and GZip [51]. Although the decompression step in these algorithms is slow and hinders query performance, people often believe they feature a better compression ratio over encoding schemes. However, this is not always the case. GZip and Snappy are one-size-fits-all compression algorithms, having no preference for the dataset. Encoding schemes are designed for datasets with particular characteristics. For example, dictionary encoding works best on datasets with low cardinalities, and delta encoding works best on sorted datasets. The nature of encoding schemes requires us to choose the encoding scheme correctly for a given dataset, which is not trivial.

To illustrate this point, in Figure 5.1a, we compress a large corpus of real-world datasets using GZip, Snappy, two popular open-source columnar datastores Apache Parquet [13] and



(a) Compression Ratio.



(b) Throughput.

Figure 5.1: Comparison of encoding schemes against an encoding selector that exhaustively evaluates encodings. The exhaustive encoding selection compresses as good as GZip and dictionary encoding is much faster than GZip and Snappy for encoding and decoding data.

Apache ORC [12], and one encoding selection algorithm from previous work [2] implemented on Parquet. We then compress the dataset with all available encoding schemes and choose the one with smallest size (Exhaustive). We see that although GZip yields a better result than Parquet and ORC, the exhaustive encoding selection achieves a similar compression ratio as GZip. In Figure 5.1b, we compare the throughput of dictionary encoding, Snappy, and GZip on a synthetic IPv6 dataset, and observe that this encoding scheme is 3x-4x faster than GZip in both encoding and decoding. This result implies that a good encoding selector allows us to benefit from both good compression ratios[1] and high query performance.

Despite the prevalence of columnar databases and the importance of appropriate encoding selection, limited work exists on selecting encodings for a given dataset. Seminal work by Abadi et al. [2] proposes a rule-based encoding selection approach that relies on the global knowledge of the dataset (e.g., is the dataset sorted) to derive a decision tree for the selec-

1. We define the compression ratio as $\frac{\text{compressed size}}{\text{uncompressedsize}}$ [131].

tion process. Open-source columnar stores such as Parquet, ORC and Carbondata [10], and commercial columnar solutions such as Vertica [92] choose to hard-code encoding selection based on the column data type. Unfortunately, these approaches all have significant limitations. Abadi's rule-based algorithm achieves a sub-optimal compression ratio and requires multiple passes on the original dataset, which becomes prohibitively expensive when dataset size increases. Hard-coded encoding selection, as we already showed in Figure 5.1a, leads to sub-optimal results in practice.

Besides compression, the encoding schemes also facilitate efficient query processing. Most encoding schemes compress each data record individually. This feature allows a carefully designed database iterator to locate the raw bytes corresponding to the records to access and decode only these records, skipping the records in between [2, 1]. A more advanced algorithm makes comparisons on the bit-pack encoded records directly without decoding any byte [67], making the query even faster. A dictionary-encoded column contains a list of distinct values. Operators, such as aggregation, can use this information to speed up execution.

Nevertheless, many open-source columnar databases [44, 9] have encoding-oblivious query engines. These systems separate the query engine and the encoded data file with a decoding layer. When the query engine reads an encoded column, the decoding layer first decodes the column into in-memory data structures, then passes the decoded data to the query engine. The query engine is blind to the encoding scheme used and has no direct access to the encoded data. This design prohibits the query engine from performing optimization towards encoded columns.

Based on these observations, we design CodecDB, a holistic encoding-aware columnar database. CodecDB demonstrates that by tightly coupling the data encoding selection in the database design, we can significantly improve the end-to-end system performance. The contribution of CodecDB is twofold. First, CodecDB provides a data-driven encoding selector to choose encoding with the best compression ratio for a given dataset. CodecDB identifies features that impact datasets' encoding performance and utilizes machine learning techniques to train a series of models to predict compression and query performance. This approach is beneficial because it requires no prior knowledge from end-users of candidate encodings, domain knowledge, or understanding details of the encoding implementation – all of which are inferred from the dataset. Our experiments show that CodecDB only needs to access a small portion of the dataset when making encoding decisions, without requiring global knowledge of the whole dataset.

Second, the query engine in CodecDB leverages advanced open-source SIMD libraries, parallel hash structures, and encoding-aware query operators to optimize access to encoded data. CodecDB achieves a significant improvement in the end-to-end performance evaluation of queries over encoded data in comparison to open-source, research, and commercial competitors. Specifically, CodecDB evaluation includes query operator micro-benchmarks, TPC-H and SSB benchmarks, and a cost breakdown analysis for a better understanding of the improved performance. This result justifies the design of a tight coupling between the query engine and the data encoding schemes. It also quantifies the benefit such a design could bring to the query performance, which is otherwise lost. We believe that this result can have significant implications considering that several recent large-scale analytic frameworks (e.g., Presto [44]) avoid this coupling to provide a simple execution engine.

We build our prototype of CodecDB on top of Apache Parquet columnar format [13], due to its popularity, extensibility for encodings, and open-source nature. Experiments demonstrate that CodecDB's data-driven encoding selection is accurate in selecting the columnar encoding with the best compression ratio and is fast in performing the selection. Specifically, we achieve over 96% accuracy in choosing the best encoding for string types and 87% for integer types in terms of compression ratio. The time overhead of encoding selection is sub-second regardless of dataset size. We evaluate the query performance of CodecDB against the open-source database Presto [44] and a commercial columnar solution, DBMS-X, using the TPC-H benchmark. We also compare against a recent research project MorphStore [32],

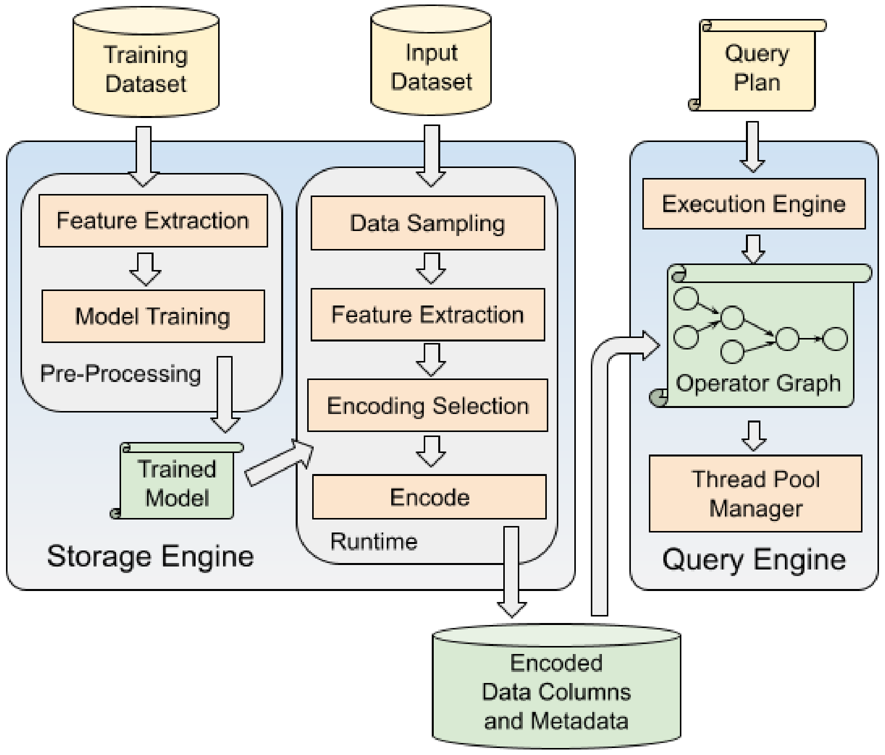


Figure 5.2: CodecDB System Architecture

Presto and DBMS-X using the SSB benchmark. CodecDB is on average an order of magnitude faster than the competitors on TPC-H, and on average 3x faster than the competitors on SSB. CodecDB also has a lower memory footprint in all cases.

We present our main contributions as follows:

- We present CodecDB, an encoding-aware columnar database that achieves both storage and query efficiency (Section 5.1).
- We propose a data-driven method for encoding selection on a given dataset to minimize storage space with high accuracy and efficiency (Section 5.2).
- We implement an encoding-aware query engine to greatly improve query efficiency on encoded columns (Section 5.3).
- We extensively evaluate our ideas (Section 5.4).

Finally, we conclude with a discussion of implications of our work (Section 5.5).

5.1 System Overview

CodecDB consists of a storage engine and a query engine. We demonstrate the system architecture in Figure 5.2.

The storage engine trains a machine learning model for the encoding selection task. When CodecDB runs for the first time, the pre-processing module executes the following tasks. First, a data collection task reads the training dataset prepared by the end-user or a default provided dataset, splits each table into columns, determines the column data type, and encodes each column using all available encoding schemes. The feature extraction task then extracts features from both the raw data columns and the corresponding encoded files. The extracted features are then used to learn how to rank encodings based on the compression ratio. We describe our encoding selection process in more detail in Section 5.2.

When the pre-processing tasks complete, CodecDB gets the runtime module of the storage engine online and is ready to encode input datasets. When the user loads a new data table, the runtime module samples each data column, calculates features for each column, runs the encoding selection model to determine the appropriate encoding scheme for each column and encodes the column. It also records encoding related metadata such as the column's dictionary and bit-width of the encoded record. CodecDB persists the metadata on disk as a plain text file and maintains it in memory as a hashmap. The query engine uses these metadata to optimize access to encoded data. We note that in the pre-processing steps, we scan the entire data column when extracting features for better accuracy. However, in the runtime phase, we only sample from the file for performance consideration. We describe more details on sampling in Section 5.4.2.

The query engine consists of two major components: an execution engine and a thread pool manager. The execution engine is the core component responsible for executing queries. The execution engine reads a query plan and builds an optimized acyclic directed graph of query operators. The query engine associates one worker task with each operator and sends the task group to the thread pool manager for execution. The execution engine returns the results to the end-user when the task group finishes execution. In Section 5.3, we demonstrate more features of the query engine, including lazy evaluation, data skipping, and batch execution.

CodecDB provides support to common operators, including filter (selection), join, aggregation, and sort. We optimize these operators to access encoded data, which is the main reason for CodecDB's performance improvement. We demonstrate the design of these operators in Sections 5.3.3, 5.3.4, and 5.3.5. The current prototype of CodecDB focuses on the execution optimization to encoded data. It does not include a query optimizer and relies on an external component to provide a feasible query plan.

5.2 Learning to Select Encodings

Lightweight encodings are each designed to accompany datasets with specific characteristics. Encoding selection is thus crucial to system performance, and hard-coded encoding selection often fails to achieve a desirable result. In this section, we introduce our data-driven encoding selection solution for optimizing compression ratios in CodecDB. We model the encoding selection as a learning-to-rank problem, identify a series of features affecting the compression ratio of encoding schemes, collect a large amount of data columns from real-world applications, and train a model to estimate the compression ratio of a given encoding scheme on a dataset.

5.2.1 Learning a Ranking Model

To train a ranking model, we consider a set of data columns $C = \{c_1, c_2, ..., c_m\}$, and a set of encoding schemes $E = \{e_1, \ldots, e_n\}$. Each data column c_i is associated with a list of compression scores $S_i = \{s_{i1}, s_{i2}, ..., s_{in}\}$, where s_{ij} corresponds to the relevance of encoding scheme e_j to column c_i. In CodecDB, we let s_{ij} be the compression ratio of e_j on c_i. We then create a feature vector $f_{ij} = F(c_i, e_j)$ for each encoding-column pair (c_i, e_j). The pair of feature vector and score (f_{ij}, s_{ij}) then form an instance in the training set.

The objective is to learn a scoring function $score : (C, E) \to \mathbb{R}$, which takes an input of data column and encoding, and output a score, that minimizes the total loss with respect to the training set:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} loss(score(c_i, e_j), s_{ij}) \tag{5.1}$$

Algorithms based on neural networks have shown great promise in learning such functions for various applications [25, 26, 70, 120]. We exploit a simple neural network that takes the column-encoding pair (c_i, e_j) as input and learns score s indicating the compression ratio e_j can achieve on c_i. We describe our network configuration in Section 5.4.2. An intuitive explanation to this model is that if a new column-encoding instance presents a similar feature set to some known instances, it should also yield a similar compression ratio.

5.2.2 Feature Extraction

The features should reflect the characteristics of a given dataset that affects its compression ratio under the encoding schemes we study. We expect our method to be able to make encoding selection by only accessing the first several blocks of the file, rather than the high overhead of scanning and parsing the entire file. Therefore, these features must be computed on a subset of the records in the dataset. In this section, we describe the features we develop in CodecDB to assist encoding selection, where we use N to denote the number of values in a target column, and $[a_1, a_2, \ldots, a_n]$ to represent the values in the column.

Value Length: We compute the length of each value in the target column as the number of characters in its plain string representation, and compute statistical information including mean, variance, max, and min. The compression ratio of bit-packed encoding is closely related to the length distribution of data records. The shorter the records are, the better it can be compressed.

Cardinality Ratio: Cardinality ratio is the ratio of number of distinct values vs. the number of values in the dataset:

$$f_{cr} = \frac{C_N}{|N|}$$

Where C_N is the cardinality of N. To process datasets with large cardinalities, we adopt a linear probabilistic counting algorithm proposed by Whang et al. in [121]. We maintain a bitmap B, compute a hash value for each record and insert a bit into the corresponding location of the bitmap. Let o be the number of occupied bits in the bitmap, the cardinality then can be estimated as follows:

$$C_N \approx -|B| \log \left(1 - \frac{o}{|B|}\right)$$

Cardinality ratio has a direct relationship with dictionary encoding. A dataset with high cardinality ratio is unlikely to be compressed well by dictionary encoding as there are too many distinct entries.

Sparsity Ratio: Sparsity ratio is the number of non-empty records vs. total number of records:

$$f_{ne} = \frac{|\{i|a_i \text{is not empty}\}|}{|N|}$$

A high sparsity ratio means there are many empty entries in the dataset, and implies a better compression ratio with schemes that looks for repetitions in the dataset, such as dictionary encoding and byte-compression.

Entropy We treat the dataset as a byte stream, and compute its Shannon's entropy:

$$f_e = \sum_{c_j \in C} -p(c_j) \log p(c_j)$$

where $C = \{c_k | \exists i, c_k \in a_i\}$ is the collection of characters in the string, and $p(c_j) = \frac{\sum_{i,k} \mathbb{I}(a_i[k]=c_j)}{\sum_i |a_i|}$ is the frequence of character c_j. We also compute Shannon's entropy separately for each value in the column, then collect the statistical information, including mean, variance, max, and min of the entropy values. Shannon's Entropy provides a lower bound for the theoretical best compression ratio that can be achieved by any encoding / compression schemes. In general, a lower entropy value means less information is included in the dataset, which implies a better compression ratio for dictionary encoding, bit-packed encoding, and byte-compression.

Repetitive Words: Most popular byte-compression algorithms that belong to the LZ77 family work by encoding repetitive occurrences of strings as tuple (*prev_ location*, *msg_ length*, *next_ char*), which basically refer to the nearest previous occurrence of the string. The compression ratio of the LZ77 family can be computed as $ratio = \frac{L_c}{L_s} \sum_{m=1}^{M} K_m$ where L_s is the input data length, L_c is the tuple length, M is the maximal message length, and K_m is the number of messages of length m. As L_c, L_s, and M are all constants, we can explore the efficiency of the compression algorithms by making an approximation on K_m, the number of distinct repetitive words in the dataset.

We use a block-based analysis algorithm similar to what is used in LZ77. We treat the input dataset as a byte stream and read a block of size S from it. Starting from the beginning of the block, we scan the content and record the substring $s(i, j)$ we met so far, where i is the start point of current scan, and j is the current read position. If $\exists\ k < i, s(k, k+j-i) = s(i, j)$ (i.e., $s(i, j)$ occurred before) we restart the scan starting from $j + 1$. When reaching the end of the block, we record the total number of new messages discovered, as well as their length distribution. For efficiency, we represent a string with its Karp-Rabin fingerprint [71]. Given a string $a = a_0a_1a_2 \dots a_n$, a large prime number p and a random $r < p$, the Karp-Rabin fingerprint $krf(a)$ is defined to be

$$krf(a) = (\sum_{i=0}^{n} a_i r^i) \mod p$$

This representation also allows easy substring concatenations as $krf(concat(a, b)) = krf(a)+$

$r^{|a|}krf(b)$. Converting a string to its fingerprint allows our algorithm to make faster string comparisons and requires less space for intermediate result. The probability that two different substrings have the same fingerprint is very low [71] and we ignore such cases.

Sortedness: The sortedness of a dataset evaluates how "in order" a dataset is. Many encodings schemes can achieve better a compression ratio from a well-sorted dataset. For example, run-length encoding can generate longer runs on a sorted dataset than the same dataset with records randomly organized. Delta-bitpacked hybrid encoding can also benefit from the sorted dataset as the delta values between sorted entries tend to be smaller and thus can yield to shorter bit-packed entries. Previous methods (e.g., [2]) use a boolean value to represent whether a dataset is sorted or not. However, we observed that a continuous variable more robustly captures the sortedness property of a dataset, as in practice, many datasets can be "partially" sorted and these partially sorted datasets still benefit from certain encodings. For example, a dataset is 90% sorted will have longer runs than a non-sorted dataset.

We adopt three methods of evaluating the sortedness of a column, f_s, and include all of them in feature sets. Kendall's τ [72] and Spearman's ρ [33] are two classical measures of rank correlation. For our purpose of evaluating the sortedness of a given dataset, Kendall's τ is computed as

$$\tau = 1 - \frac{2\left|\{(a_i, a_j)|i < j, a_i > a_j\}\right|}{n(n-1)/2}$$

and Spearman's ρ is computed as

$$\rho = 1 - \frac{6\sum_{i=1}^{n}(s_i - i)^2}{n(n^2-1)}$$

Both methods generate a real number in $[-1, 1]$. 1 means the dataset is fully sorted, and -1 means the dataset is fully inverted sorted. However, most lightweight encodings will work just as well on a fully inverted sorted dataset if it works well on a fully sorted one. Observing this, we define a variant, called absolute Kendall's τ.

$$\tau_{abs} = 1 - |1 - 2\tau|$$

and τ_{abs} has a value range of $[0, 1]$, and approaches 0 when the dataset is close to either fully sorted or fully reverse sorted.

Computing the sortness features on the entire column have a time complexity of $O(n^2)$, which is prohibitively time-consuming. Therefore, we adopt a sliding window method. We slide a window of size W over the dataset and with probability p perform computation on pairs within that window. There are in total $n - W + 1$ such windows, and for each window, the time complexity is $O(W^2)$. The time complexity will be $p \cdot (n - W + 1) \cdot O(W^2)$. By setting p to $\Theta(\frac{1}{W^2})$, we can perform the computation in $O(n)$.

5.2.3 Dataset Collection

We derive our training set from various structured data collections (e.g., open city data portals, scientific computation cluster logs, machine learning datasets, traffic routes, and data challenge competitions). We describe the dataset in more detail in Section 5.4.1. We develop a dataset collection framework for data preparation. The framework consists of a file reader, a feature extractor, and a data store. The file reader uses file extensions to determine file format and invokes a corresponding parser. We currently support common tabular file formats, including CSV, TSV, JSON, XLS, and XLSX. The file reader splits a file into columns and infers each column type, then extracts features on the generated columns. The framework stores the generated columns as separate files in the file system and metadata and extracted features in a DBMS.

We use the encoding algorithms shipped with Apache Parquet. To determine the best encoding (encoding that compresses a given column with minimal size) for each data column, we apply all viable encodings to every column and compute the corresponding compression ratios. The encoding scheme having the best compression ratio is chosen as the "ground-truth" in the training phase.

We collect data and train the model on a variety of input data, and expect that the trained model can be applied to many users and datasets. However, including new encoding types would require re-training of the model.

5.3 Encoding-Aware Query Engine

We build an encoding-aware columnar query engine in CodecDB. With it, we demonstrate that a holistic system design with a tight coupling between encoded columns and database operators significantly improves the end-to-end query performance. The query engine maintains the encoding information of data columns and applies operators optimized for the corresponding encodings. These operators rely on recent algorithms that provide higher throughput, consume less CPU time, and less memory footprint than previous similar solutions. The query engine also provides features targeting big data analysis, such as lazy evaluation and batch execution. Next, we introduce the operators and these features of the query engine.

5.3.1 In-Memory Data Structures

CodecDB keeps the execution results of operators in in-memory data structures, also known as mem tables. When the operator performs a selection on one input table, we store the result in a bitmap, with each bit marks each row's validity. Our bitmap provides SIMD-based logical operations of bitmaps and a fast iterator allowing users to access all marked positions. As the input tables can be arbitrarily large, CodecDB provides a sectional bitmap consisting of multiple small bitmaps, with each section corresponding to a data block. Individual sections can be cached to an external storage, or be compressed individually to reduce memory footprint. CodecDB supports compressing the bitmap using run-length encoding.

CodecDB provides row-based and columnar mem tables, which can be used to store results from aggregation, join and sort operators. The mem tables support common primitive types, including int32, int64, float and double, and variable-length binary type. CodecDB uses a 'zero-copy' strategy for binary data. Each binary field in mem table is a `struct binary {uint8_t* ptr, uint64_t len}`, where `ptr` is a pointer to the start of the binary record, and `len` the length of the record. When loading a binary column from external data files, CodecDB decodes the binary records into the internal buffer and returns a reference to that record. Subsequent operations that move binary records between mem tables only involve copying the pointer, not moving the data.

5.3.2 Operator Evaluation

CodecDB evaluates all operators in an operator graph in parallel to utilize multi-core platforms. The parallelism happens on two levels, operator level and data block level. When executing a query, CodecDB treats each operator as a task and submits the task group to an operator thread pool. Each task is blocked until all its ancestors finish. Operators not related (e.g., two filters on different columns) can thus run in parallel. Operators also access their input data in parallel by splitting the input as multiple data blocks and use a data processing thread pool to process the blocks. All operators share the same data processing thread pool, and we configure the thread pool size to limit the memory footprint used by each query.

CodecDB utilizes pipelined evaluation of operators and late materialization to reduce the memory footprint used by temporary relations. To group query operators from an operator graph into pipelines, we implement lazy evaluation of operators. Lazy evaluation does not execute an operator immediately when it is invoked. Instead, it maintains a record of operators that have been called, and executes them all together with a pipeline when encountering a blocking operator, such as sort and aggregation. With lazy execution, we convert the operator graph into a directed acyclic graph of pipeline stages. Each of these

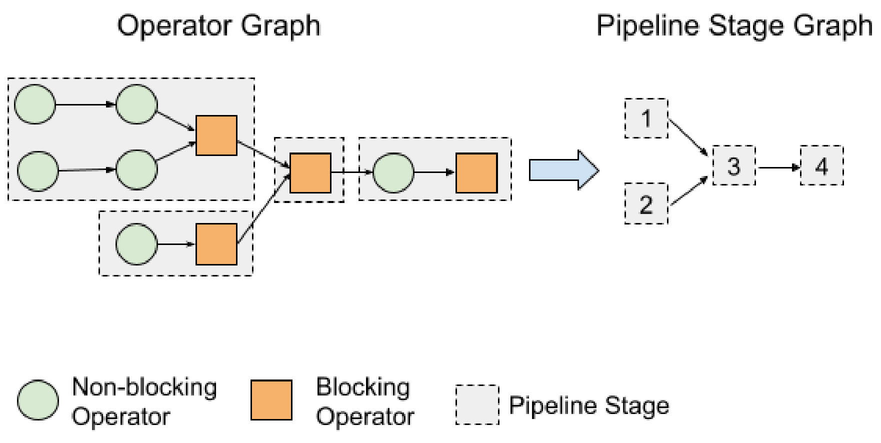


Figure 5.3: Lazy Evaluation groups operators to pipeline stages

stages contains multiple non-blocking operators and one blocking operator. Evaluating a blocking operator will execute the corresponding pipeline stage. We show this process in Figure 5.3. When executing all operators in parallel, the blocking operators separate the operators into four pipeline stages. The first two pipeline stages have no dependencies and execute in parallel. Other pipeline stages start execution when their ancestors finish execution.

CodecDB categorizes an operator as non-blocking if it can execute locally on a single data block, without global information from the entire data file. Filtering and probing a hash table are examples of non-blocking operators. An operator is blocking if it reads multiple data blocks. Building a hash table, aggregation, and sorting are examples of blocking operators.

To execute pipeline stages, CodecDB designs a data stream framework and implements a demand-driven pipeline based on it. A stream of type T, represented by `Stream<T>`, provides two functions: *map(function<S(T)>)* and *foreach(function <void(T)>)*. Users create a pipeline by obtaining a data stream, calling *map* to add operations to the pipeline, and calling *foreach* to execute the pipeline. For example, we show how to build a pipeline to count the positive values in an integer column. We first get a stream of data blocks `Stream<Block>` from the column, and call *map* with a *function<Bitmap(Block)>*. This function scans a block and returns a bitmap marking positions of all positive numbers. We then call *foreach* with a *function<int(Bitmap)>*, which counts the size of each bitmap and sum them up to get the result. The call to *foreach* triggers the pipeline execution and returns the results.

CodecDB supports batch executions of operators accessing the same data column to reduce disk read and improve cache locality. It searches in the execution graph for operators reading the same data column and groups them. When the first operator in the batch group executes, the engine reads disk files, feed it to all operators in the group, and caches the result. When subsequent operators run, the engine directly fetches results from the cache.

After filtering a data table on some columns and obtaining a bitmap, CodecDB uses the bitmap to retrieve data from other columns, known as late materialization [114]. We optimize data retrieval by implementing data skipping in all column readers. Data skipping allows readers to jump to the next valid record marked by the bitmap, skipping all records in the middle without reading them. Data skipping save both disk IO and CPU cost. Without data skipping, column readers need to read all records, decode them, and discard those not required. In CodecDB, we implement data skipping on three levels:

- Data Block Level. When the corresponding bitmap section is empty, CodecDB skips the entire data block. Skipping a data block saves disk I/O.

- Data Page Level. Parquet splits each data block into multiple data pages and compresses each page independently. CodecDB will skip the whole page without decompressing it if the next row to read surpass the boundary of an unread page. Skipping data pages saves decompression effort.

- Row Level. CodecDB computes the number of bytes corresponding to the number of rows to skip, reads those bytes from the input, and discards them. Skipping rows saves decoding effort.

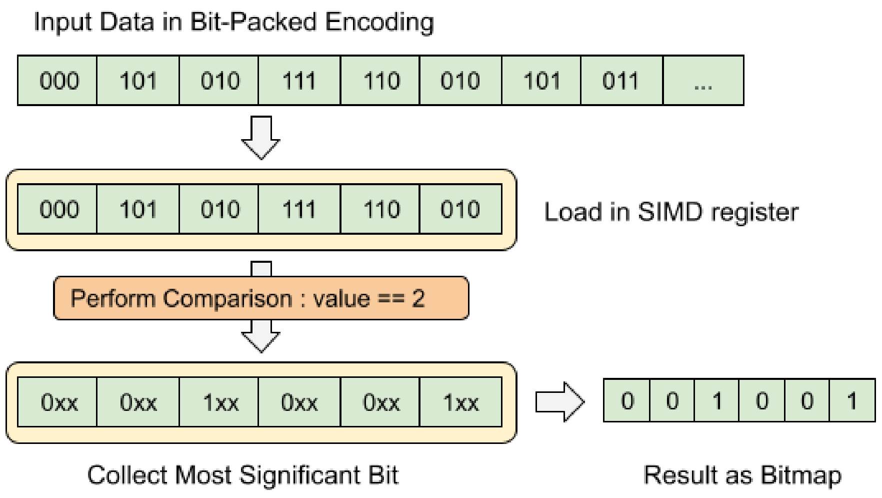


Figure 5.4: SBoost *in-situ* Scan for Bit-packed Data

5.3.3 Filter Operator

CodecDB provides two families of filter operators optimized for dictionary encoding and delta encoding, based on SBoost [67]. SBoost is an open-source library containing fast decoding and in-place search algorithms for lightweight encodings utilizing SIMD instructions. We briefly review how it works here.

One core algorithm in SBoost is in-place scanning of a bit-packed data stream, as is shown in Figure 5.4. The algorithm loads multiple bit-packed data entries into a SIMD register and compares all entries in parallel. It then fetches the most significant bits from each entry as a bitmap, representing the comparison result. SBoost supports all relational operators, including equal, less, greater, and their combinations. The advantage of this algorithm is two-fold: first, the algorithm performs comparison directly on the encoded data and does not decode the data. Skipping the decoding saves a huge computation effort. Second, the algorithm uses SIMD to perform comparisons on multiple data entries in parallel. For example, when scanning a bit-packed stream of size 10 (each entry takes 10 bits), SBoost uses only 8 instructions on average to process 50 entries, achieving over 20x throughput compared to the scalar algorithm that first decodes each entry then performs the comparison.

Dictionary encoding maps data entries to integer keys, and bit-pack the keys. CodecDB provides a single column filter operator on dictionary encoding. The operator uses the data column's dictionary to translate the query value to an integer key, then invokes SBoost to perform an in-place search on the bit-packed keys to find the target. This filter operator also supports greater, less, and range comparisons, if the dictionary is order-preserving. In an order-preserving dictionary, for any two entries $key1 = value1$, and $key2 = value2$, we have $value1 > value2 \iff key1 > key2$. With an order-preserving dictionary, we can rewrite any comparison on the encoded values to comparisons on the keys and invoke SBoost to execute the query.

This single column filter operator does not only support comparison predicate but also `LIKE` and wildcard operations. Examples are *p_type like '%BRASS'* in TPC-H query 2 and *l_shipmode in ('MAIL', 'SHIP')* in query 12. The operator translates these queries as a logical disjunction of multiple equality operators. For example, to execute *p_type like '%BRASS'*, the operator scans the dictionary entries and finds all entries ending with '%BRASS', performs equality predicate for each entry as we described in the previous paragraph, and makes a logical `OR` on the result bitmaps to obtain the final result.

We use a similar idea to build a filter operator to compare multiple data columns using the same order-preserving dictionary. One example is comparing two `DATE` columns,e.g., `o_commitdate < o_receivedate`. Giving the two columns sharing the same order-preserving dictionary, the value of `o_commitdate` is less than `o_receivedate` if and only if the corresponding key of commit date is smaller than that of receive date. We extend the SBoost algorithm to support the comparison between two bit-packed data streams and use the result bitmap as the filter output.

Lastly, CodecDB provides a filter operator optimized for delta encoding. Due to the nature of delta encoding, we need to decode the entire data column before making a comparison. SBoost provides a SIMD algorithm to compute the cumulative sum of 8 integers fast, which we use to speed up the decoding process. CodecDB's delta filter loads a list of delta values into memory, invokes SBoost to compute their cumulative sum, and uses SIMD comparison instruction to compare them against the target value.

5.3.4 Aggregation Operator

CodecDB provides an aggregation operator optimized for data columns with dictionary encoding for the aggregation key. The operator starts by creating an array of aggregation results with the same size as the dictionary. It then reads each row to be aggregated, fetches the integer key from the group by column, and uses it as an index into the array to update the aggregation result. This approach works because the integer key is always a value between 0 and dictionary size. We call this operator array aggregation, for it uses an array to keep the aggregation results.

Compared to the widely used hash aggregation, array aggregation has several advantages. First, hash aggregation needs to compute a hash value from the key. Array aggregation directly uses the stored integer key and does not require additional computation. Second, hash keys can collide, and hash aggregation needs to employ a collision resolution such as open addressing, at a performance cost. Array aggregation has no collisions. Finally, when performing aggregation using multi threads, we first aggregate multiple data blocks in parallel and merge the result. Merging two hash tables is far less efficient than merging two arrays.

Array aggregation executes efficiently for small key spaces, and is only applicable to dictionary encoding. When the key space is large or the column is not dictionary encoded, CodecDB provides a stripe hash aggregation operator, a variation of hash-based aggregation. The operator first splits each data block into stripes by aggregation key. This step guarantees that the same key will always go to stripes with the same index, and tries to spread keys evenly to each stripe. In the current implementation, we compute the stripe index as the key modulo the number of stripes. Next, it performs hash aggregation independently on each stripe in parallel. Finally, it merges stripes with the same index from different blocks together. The major advantage of stripe hash aggregation versus the vanilla version is that it splits a big key space into multiple small ones, and uses several small hashtables instead of a single big one in aggregation. Smaller hashtables facilitate better cache locality, and using several small hash tables allows updates and merges in parallel. These advantages enable stripe hash aggregation to provide better performance than vanilla hash aggregation.

5.3.5 Other Operators

In this section, we briefly introduce other operators CodecDB provides, not optimized for encoded data.

CodecDB provides nested loop join, block nested loop join, and hash join. For hash join, we adopt phase concurrent hashmap(PCH) proposed by Shun et al. [112]. PCH uses a lock-free algorithm allowing operations of the same type to run in parallel. Multiple threads can perform insertion only, search only, or deletion only at the same time with no data races. In CodecDB hash join has two phases. The first one is building the hash table from one table, involving only insertion operations. The second one is searching the hash table (in a typical hash join) or removing entries from it (in hash-based exist join). The two phases do not overlap as the second phase can only start after the hash table is ready. This allows us to use PCH in all hash-based join operators. CodecDB provides an in-memory sort operator and an external merge sort operator. For top-n queries, it offers an in-memory heap-based top-n operator.

5.4 Experiments

In this section, we show the experimental results demonstrating that CodecDBshows significant improvement in both storage and query efficiency. Storage-wise, we show that CodecDB's data-driven encoding selection can accurately identify the encoding with a good compression ratio for various datasets. We also provide an in-depth analysis of the reason our approach excels competitors. Query-wise, we demonstrate CodecDB's query operators

Table 5.1: Datasets Statistics By Category

Category	Table Count	Column Count	Data Size(GB)
Server Logs	166	3836	20.4
Government	256	5126	26.8
Mach. Learning	111	3113	12.5
Social Network	98	1593	23.9
Financial	91	1954	16.8
Traffic	50	2826	22.8
GIS	16	382	5.2
Other	8	428	1.6

outperform their encoding-oblivious competitors. We also show the query engine outperforms open-source query framework, commercial columnar database, and a recent research project in two established benchmarks, in both query time and memory footprint. We further elaborate on how CodecDB achieve such improvement.

5.4.1 Environment Setup

The experiment platform has two Intel(R) Xeon(R) Gold 6126@ 2.60GHz, 192G memory, and 250G SATA SSD. It runs Ubuntu 18.10 with kernel version 4.15.0-101. We build our CodecDB prototype in Java (encoding selection) and C++ (storage engine and query engine). The Java part runs with OpenJDK 1.8.0-252 and Scala 2.12.4. The C++ part is compiled using GCC 7.5.0 with `-O3`.

The datasets we use for data-driven encoding selection come from multiple public data sources, covering a wide variety of domains and application scenarios. Table 5.1 shows the statistical overview of datasets by their categories. These domains generate and store massive amounts of data, facilitating many vital applications.

Columns of string and integer types dominate the dataset (over 76%). Columns of double type also occupy a considerable portion (17%) in the datasets, most of which belong to GIS, machine learning, and financial datasets. However, Parquet only supports Dictionary encoding for double attributes, and double attributes usually have high cardinality, making it unfit for Dictionary encoding. We choose to focus on string and integer types in our experiment.

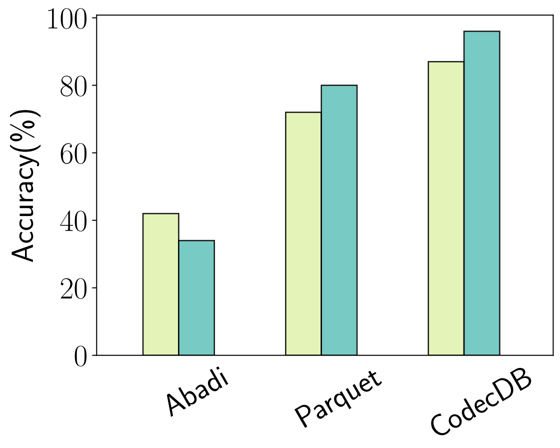
In query evaluation, we compare against Presto version 0.226 and a commercial columnar solution DBMS-X with the latest version, on the TPC-H benchmark. We also compare against MorphStore [32], Presto and DBMS-X on SSB. All systems run on the same hardware platform mentioned above.

5.4.2 Data-Driven Encoding Selection for Compression

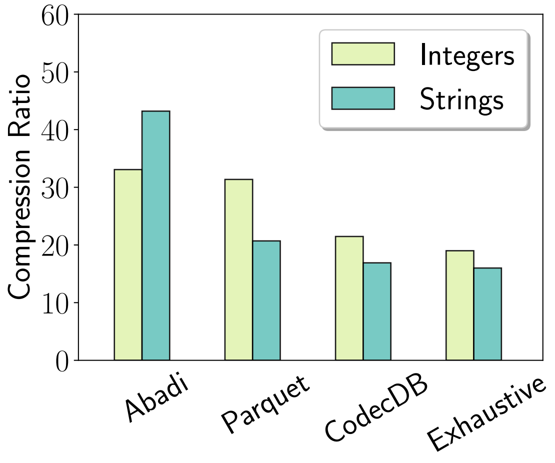
In this section, we evaluate the accuracy of our neural network based data-driven encoding selection method for improving compression ratios. We use a standard MLP neural network for both the classification and the regression task. We construct a two-layer neural network with 1000 neurons in the hidden layer, using tanh as the activation function. We use sigmoid for output, and cross-entropy as the loss function when performing ranking. We train the network with Adam [75] for stochastic gradient descent using default hyper-parameters($\alpha = 0.9, \beta = 0.999$). The step size is 0.01, and decay is 0.99. We use 70% of data columns for training, 15% for dev, and 15% for testing. No noticeable impact is observed when we change the way of partitioning the dataset (e.g., 80/20 for training/testing). Feature `Sortness` has a hyperparameter W for sliding window size. We choose window size to be $50, 100$, and 200, and include all results in the feature set.

We also compare the accuracy of our method to other candidate approaches. Abadi et al. [2] propose an encoding selection method based on a hand-crafted decision tree. They use features that are similar to what we employ in this paper, including cardinality and sortedness (although binary), and empirically setup selection rules. We refer to this decision tree approach as *Abadi* in experiments.

Apache Parquet has a built-in encoding selection mechanism which simply tries Dictionary encoding for all data types. When the attempt fails, it falls back to a default encoding for each supported data type. In practice, we notice that such a failure is primarily caused



(a) Selection Accuracy



(b) Encoded Size

Figure 5.5: Accuracy and Encoded File Size of CodecDB's Encoding Selection

by the dictionary size exceeding a preset threshold, which means the dataset to be encoded has high cardinality. So this can also be viewed as a simplified version of a decision tree. We refer to this rule-based approach as *Parquet* in experiments.

In Figure 5.5a, we show selection accuracy of different approaches, which is the percentage of samples the algorithm successfully choose the encoding with minimal storage size after encoding. For string columns, CodecDB achieves 96% accuracy, a significant improvement from Abadi's decision tree with only 32% accuracy, and Parquet's encoding selection of 80%. For integer columns, CodecDB achieve 87% accuracy, also a substantial gain from Abadi's 40% and Parquet's 72%. Note that we evaluated alternative machine learning models and settled on a neural network as it provides the highest accuracy. Several other models had high accuracy, which also justifies that our features engineering represents critical characteristics of the dataset for encoding. In Figure 5.5b, we show how much storage reduction each algorithm can bring to the entire dataset, where "exhaustive" is the observed best encoding scheme after exhausively testing all valid encoding schemes for an attribute. CodecDB's encoding selection can bring ∼30% size reduction compared to Parquet, and delivers a compression ratio close to the exhaustive result. The compression ratio is also competitive against commercial columnar stores. On TPC-H dataset of scale 20, CodecDB compresses data tables to 9.8G, 10% smaller than DBMS-X, which compresses tables to 11G. Presto uses the same data tables as CodecDB.

Next, we evaluate whether some features play more important roles than others. To verify this, we iteratively remove each feature from the set, retrain the network with the same parameters. The result shows that removing any feature brings a drop in prediction accuracy of 18∼25%, with cardinality and length leading the drop. This result is expected as most features are designed to map to some specific encoding schemes. For example, sortness is important to delta and RLE encoding, cardinality is crucial to dictionary encoding and also has an effect on bit-packed encodings. As a result, removing any of the features will lead to a misprediction on a subset of encoding schemes.

Case Study: Where previous methods fail

We have chosen three typical cases to show where Abadi and Parquet's methods underperform compared to our approach.

Case 1: Abadi Tree for High Cardinality Abadi's approach has the following selection path: if the number of distinct values is greater than 50000, use either LZ compression or no compression based on whether the data exhibits good locality. However, we observe that when the number of distinct values is greater than 50000, there are still over 12% of attributes for which bit-packed encoding achieves better compression than LZ. For these cases, merely removing leading zeros result in better space savings than removing repeating values.

Case 2: Abadi Tree for Run-Length Another selection path in Abadi's approach is that when average run-length is greater than 4, it uses run-length encoding. However, we found that there are over 23% of columns having an average run-length greater than 4, where dictionary encoding performs best. This difference can be a factor of encoded key size compared with the value size, local dictionaries that leverage partially sorted datasets to provide small keys, and bit-packing or run-length dictionary hybrids.

Case 3: Parquet for Bit-Packed Parquet by default always tries to use dictionary encoding. But our data shows that for integer columns, there are only 72% of attributes have dictionary encoding as the ideal encoding. A considerable amount of the remaining integer columns can be compressed well by bit-packed encoding, which Parquet fails to choose.

We find that these methods typically suffer from the following problems that CodecDB's approach addresses:

- Unable to extend to new encoding schemes or encoding scheme variations
- A single property (e.g., run-length, cardinality) cannot distinguish different groups
- Expert knowledge-based parameters can be inaccurate (and expensive to obtain)

CodecDB does not hard-code the decision but relying on the data characteristics to make the decision, allowing it to make a better choice than previous methods.

Encoding Selection on a Partial Dataset

We have demonstrated that a neural network-based data-driven encoding selection method outperforms current state-of-art from academic research and open-source implementations. However, most features we employ require scanning the entire column, which is time-consuming. To mitigate this problem, we read only the first N bytes from the dataset and compute features based on those values. We then use the computed features to make decisions as in the original method, eliminating the correlation between dataset size and time needed for encoding selection, making it possible to make selection decisions in constant time.

To empirically estimate how much accuracy we can achieve with only partial knowledge of the dataset, we vary N to be $10K, 100K$, and $1M$ bytes. This experiment is conducted only on data columns whose size are larger than 10MB to avoid oversampling. Not surprisingly, the prediction accuracy decreases when a smaller M is used. However, we still manage to achieve reasonable accuracy. Our result shows that when $N = 1M$, we have 85% accuracy on integer and 94% accuracy on string. With $N = 10K$, we can get 83% accuracy on integer dataset and 92% accuracy on string dataset. which is still better than state-of-art.

Random sampling [59, 94] is another widely adopted sampling method in previous works.

We also compare the result of random sampling with our approach of head sampling. When applying random sampling, the accuracy of encoding selection drops drastically to 65%, and we noticed that the misprediction primarily occurred on data columns suitable for delta encoding and run-length encoding requirements to data locality. Delta encoding measures the difference between adjacent values, and run-length encoding counts the consecutive repetition of values. As the sampled data from random sampling failed to preserve this locality, prediction using randomly sampled data does not yield a satisfactory result.

Performance Overhead

In this section, we study the performance overhead of the data-driven method, with time consumption only involving feature extraction and model execution. The model training process is conducted off-line and is not included. We test selection time when choosing the first 1M bytes to generate features. The average time for calculating the features on a single data column is 57ms, and that for executing the model is 3ms. We also compare the data-driven method running time against exhaustive encoding selection, which encodes a data column with all encoding candidates and choose the one with the smallest size. Our experiment includes four encoding types for integer data and three encoding types for string data. As the encoding time is proportional to input file size, we execute CodecDB's feature extraction on the entire data column to make a fair comparison. Result shows CodecDB is 2.5x faster compared to exhaustive encoding when scanning then entire column. With our default setting of sampling the first 1M bytes, CodecDB can be three orders of magnitude faster than the exhaustive approach on a 1GB data column. When the selection involves more encoding types, CodecDB will benefit more compared to the exhaustive approach.

5.4.3 Encoding-Aware Query Execution

In this section, we explore CodecDB's query engine performance. We start by showing micro-benchmark results of CodecDB's operators. We then show that CodecDB outperforms

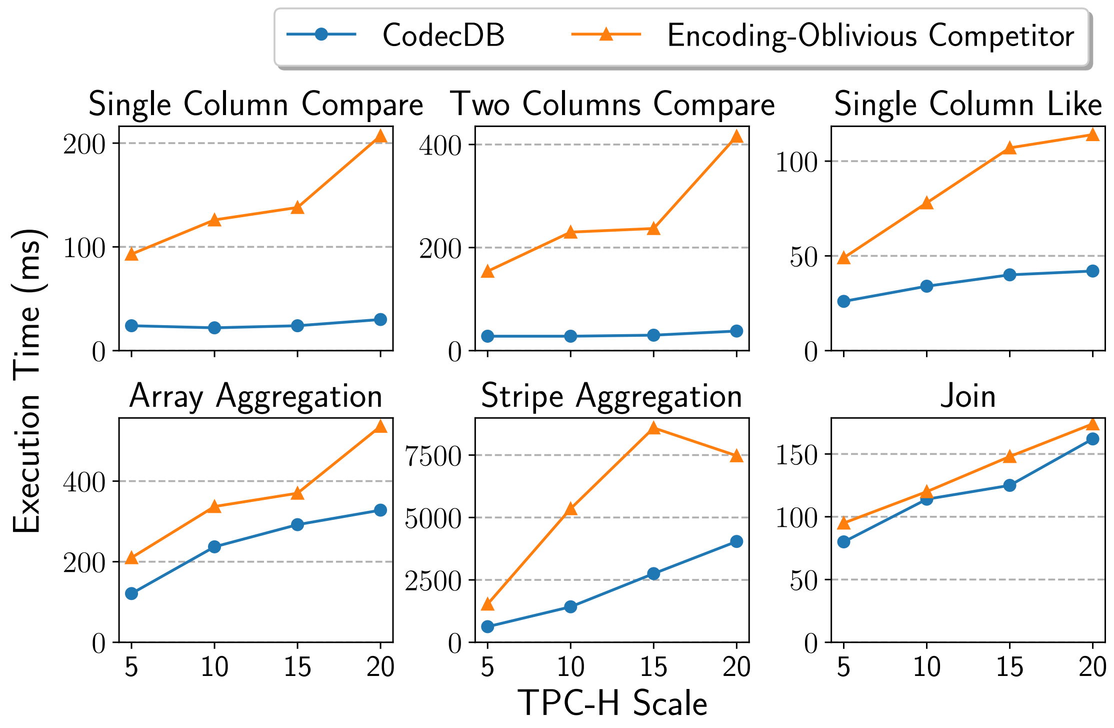


Figure 5.6: CodecDB operators outperform encoding-oblivious operators

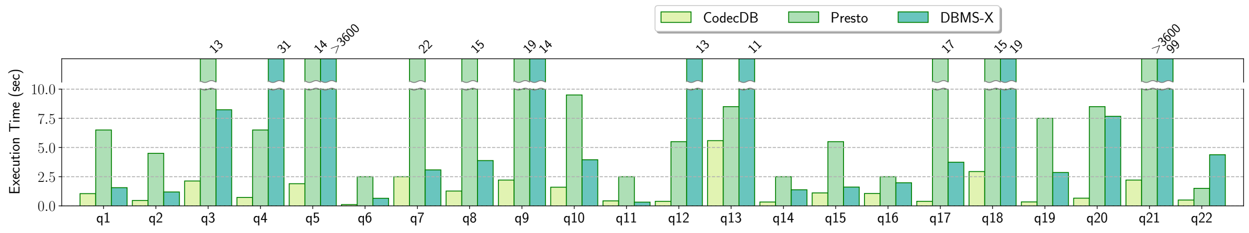


Figure 5.7: CodecDB outperforms encoding-oblivious columnar databases in TPC-H Benchmarks (Scale 20)

competitors on TPC-H and SSB. We also provide breakdown analysis that helps explain how CodecDB achieves such improvement.

In Figure 5.6, we test CodecDB operators on various TPC-H scales and show that CodecDB operators always outperform their encoding-oblivious competitors. We also describe these competitors after describing each operator. We first test filter operators on the dictionary encoded column. The “Single Column Compare” in Figure 5.6 corresponds to the predicate `l_shipdate <= '1998-09-01'`, “Two Columns Compare” corresponds to `l_commitdate < l_receiptdate`, and “Single Column Like” corresponds to `p_container`

`LIKE ‘LG%’`. The competitors in these cases are encoding-oblivious, who decode records from columns and make comparisons. We see that CodecDB's operators bring 5-20x performance boost compared to encoding-oblivious solutions, and has more advantage when dealing with large datasets. While encoding-oblivious solutions' time consumption almost doubles when moving from the TPC-H scale 5 to 20, CodecDB's time consumption only increases by 30%. Next, we test the aggregation operators. In "Array Aggregation", we count `lineitem` group by `l_receiptdate`. CodecDB uses an array of size 2560 to perform aggregation, and the competitor uses the Google sparsehash [55] hash table. In "Stripe Aggregation", we count `orders` group by `o_custkey`. CodecDB splits the input into 32 stripes, and the competitor uses a sparsehash hash table that does not split the input. In both cases, we see a 2-3x performance improvement. The last experiment in the micro-benchmark is our hash join operator based on phase-concurrent hashtable. The competitor uses a sparsehash hash table. We join `orders` with `customer` on the foreign key, with condition `c_mktsegment=‘HOUSEHOLD’`. The results show 10-15% improvement, primarily because we can build a hash table using multiple threads.

Next, we compare CodecDB query engine against popular columnar database solutions on TPC-H benchmarks of scale 20. We choose two candidates, Presto [44] and DBMS-X. Presto is an open-source distributed SQL query engine designed to query large data sets. Presto supports the Parquet storage format and executes queries in an encoding-oblivious way, making it a good candidate to show the advantage of encoding-aware query execution. DBMS-X is a commercial big data analytic system leveraging columnar storage. We encode the tables in Parquet format with column chunk size of 128M, and page size 1K. For all systems, we limit the number of concurrent threads per query to 20.

We setup Presto to use a single node, with the maximal memory per query set to 20G. Presto reads the same Parquet tables as CodecDB does. DBMS-X lacks support for many Parquet encodings, prohibiting it from reading our Parquet tables. Instead, we use its native table format with auto compression. For query efficiency, we load data into DBMS-X's Read-

Only Storage, a highly optimized read-oriented disk storage structure. As CodecDB is not equipped with a query optimizer, we use Presto to generate a query plan for each query, replace the operators with CodecDB's corresponding version, and manually code the query plan into CodecDB. We measure the time Presto and DBMS-X spend on generating query plans, and deduct it from the execution time to ensure a fair comparison.

We run all TPC-H queries with CodecDB, Presto and DBMS-X and show the result in Figure 5.7. We limit the bar graph's height and show the time consumption on the top of the bar. DBMS-X on Query 5 and Presto on Query 21 do not finish after 1 hour, we record these two outliers as ">3600" seconds, and ignore them in subsequent analysis. In general, we see a substantial performance improvement of CodecDB versus competitors. For all 22 queries, CodecDB is, in average 11.43x faster than Presto and 9.81x faster than DBMS-X, excluding the outliers mentioned above. The best result versus Presto is on Query 17, where CodecDB is 46x faster. The one for DBMS-X is Query 20, where CodecDB is 44x faster.

On queries with at least one predicate on the dictionary encoded column, CodecDB performs extremely well. We see that most queries satisfy this situation in practice. 17 out of 22 TPC-H queries (exceptions are Q9, Q11, Q13, Q18, and Q22) contains at least one such predicate. In these queries, we see at least 10x performance improvement compared to competitors.

In Figure 5.8, we make a time consumption breakdown of the first four TPC-H queries to understand better how CodecDB improves query performance. We see that all four queries are CPU-bound. CodecDB's encoding-aware query execution significantly reduces the CPU execution time. Besides, data skipping helps CodecDB to reduces the IO cost. Both contribute to efficient query execution. In Figure 5.9, we compare the memory footprint of the four queries, collected using `/proc/<pid>/stat`. We see that CodecDB saves up to 80% memory footprint compared to DBMS-X. CodecDB can execute directly on encoded data without decoding them into memory, and skip data records not accessed by the query. Both contribute to the improvement. The two experiments demonstrate the benefit that a query

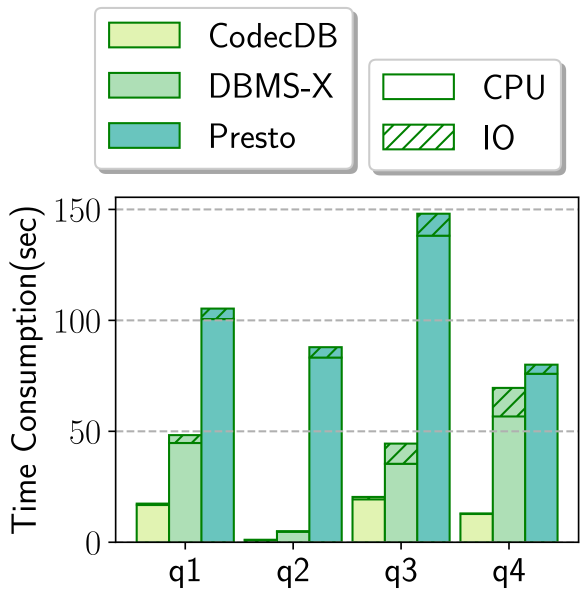


Figure 5.8: Time Breakdown of TPC-H Queries

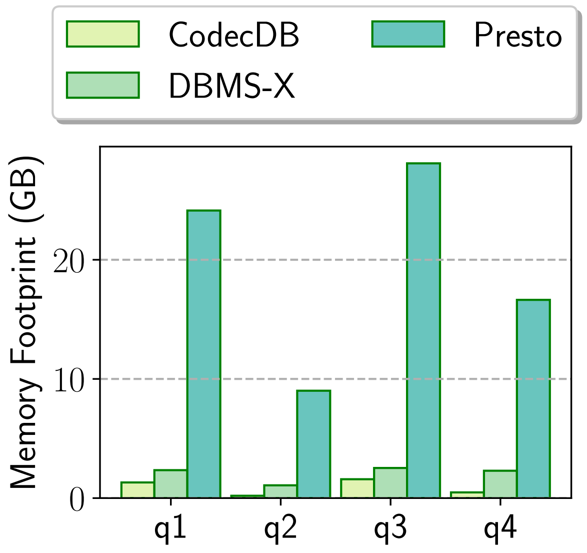


Figure 5.9: Memory Footprint of TPC-H Queries

engine tightly coupled with encoded columns brings.

Finally, we compare CodecDB query engine with MorphStore [32] on the Star-Schema Benchmark(SSB) with scale 10. MorphStore is a columnar database that compresses intermediate results with lightweight encodings to reduce memory footprint and uses SIMD to speed up query on compressed data. It shares many similar design concepts with CodecDB. We use SSB as MorphStore does not support TPC-H benchmark. We compare the two systems on query execution speed and memory footprint of intermediate results. We also execute SSB with DBMS-X and Presto, and include their results for reference. For MorphStore, we obtain the query execution time from its running artifacts and compute the intermediate result memory footprint using the minimal compressed size. For Presto and DBMS-X, we only include the query time, as the systems are not instrumented to measure the size of intermediate results.

We show the result in Figure 5.10. CodecDB is again faster on most queries than all competitors. It runs SSB queries up to 5x and in average 3x faster than MorphStore and consumes less memory on intermediate results. The reason is three-fold. First, CodecDB uses a late materialization execution strategy and generates fewer intermediate results than

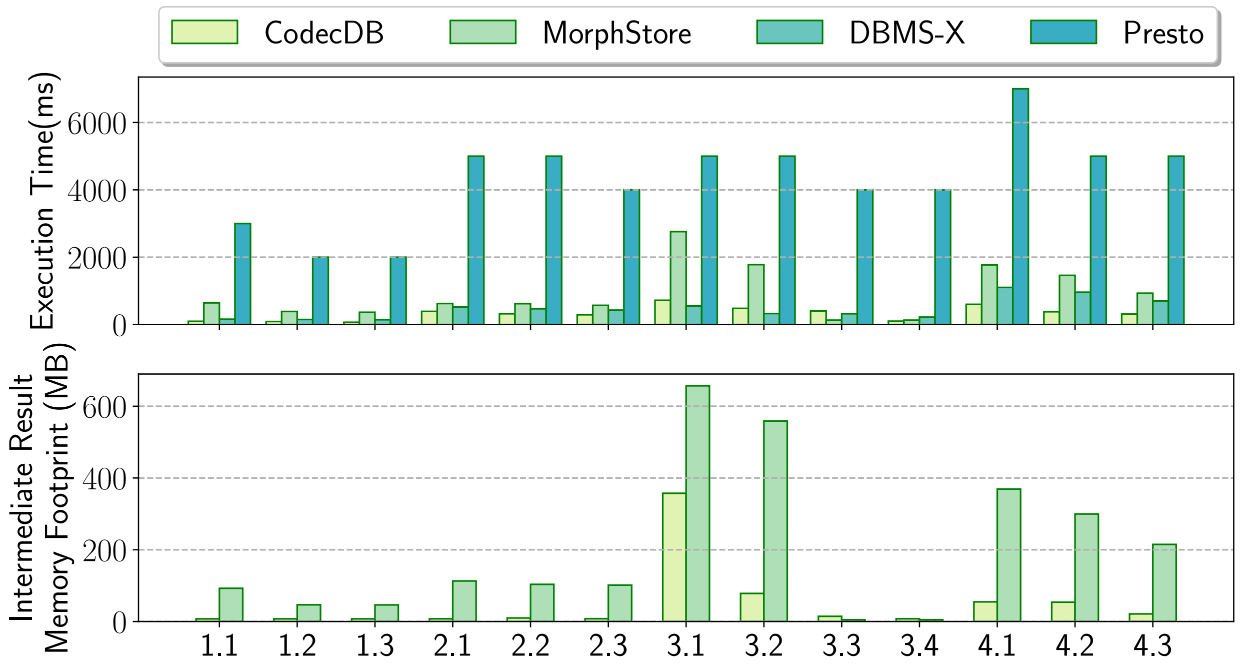


Figure 5.10: CodecDB is faster than MorphStore on SSB, and consumes less memory on Intermediate Results

MorphStore. Second, CodecDB relies heavily on bitmaps as intermediate results. Bitmaps are smaller in size and facilitate faster intersection/union operations. Finally, late materialization allows CodecDB to push down most filtering operations to SBoost, further speeds up the query execution. For example, in Q1.1, to perform a predicate+interesction operation, CodecDB uses 55ms and only generates a single bitmap of 7MB as an intermediate result. MorphStore takes 500ms on the same operation and generates 12 intermediate results sum up to 94MB. These experiments demonstrate that CodecDB is efficient in execution time and provides an alternative solution to reducing query memory footprint.

5.5 Conclusion

We propose CodecDB, an encoding-aware columnar database that exploits a design tightly coupled with encoding schemes. CodecDB combines autonomous data-driven encoding selection and encoding-aware query execution to improve both storage and query efficiency on encoded columnar data. CodecDB's storage engine analyzes data column characteristics to choose the encoding scheme that best fits a given data column, achieving a compression ratio comparable to GZip. CodecDB's query engine utilizes the encoding knowledge of data columns to improve query efficiency on encoded data.

Extensive experimental results show that on both encoding selection and query execution, CodecDB brings substantial improvement compared to prior research, widely-used open-source implementations, and commercial products. Overall, as a system, CodecDB demonstrates the great potential of the system design philosophy of encoding-awareness. In the future, we plan on expanding CodecDB to support query-aware encoding selection, include new encoding schemes, lossy compression, and further explore more encoding-aware algorithms for other database operators, such as joins.

CHAPTER 6

COLOM: COLUMNAR LAYOUT IN KEY-VALUE STORE

A Log Structured Merge Tree (LSM-tree) is an efficient data structure for key-value data persistence and query. The LSM-tree is designed to support write-intensive applications using an out-of-place update policy. Modern LSM-trees implements this by appending updates to an in-memory structure as log entries. When the memory buffer is full, LSM-tree merges the entries and flushes them as a sorted run to the disk. When the number of runs of a similar size exceeds a threshold, multiple runs are merged into a longer run, which is often referred to as a multi-level structure. The level of a run indicates how many merges it has participated in. Such an update policy enables high write throughput with a trade-off of lookup performance and storage space. Modern key-value stores, such as Bigtable [27], HBase [8], LevelDB [53], RocksDB [45], and Apache Cassandra [48] all adopt LSM-tree as their storage layer solution for its superior write performance. Additionally, Facebook built MyRocks [43], an LSM-tree based storage engine for MySQL. LSM-trees are also used to build auxiliary database structures such as inverted indices and spatial indices [123, 73, 104].

The multi-level nature of the LSM-tree brings extra effort to key lookup operations. When inserting or updating a key, an LSM-tree does not check the existence of the key. Instead, it simply writes the key and the new value to the memory buffer. Only during the merge step will an LSM-tree consolidates two entries with the same key. Multiple entries with the same key can exist in multiple levels or in multiple runs of the same level. If an application only needs the latest value of a key, it starts with the lowest level, scans all runs in that layer, and stops as soon as it finds the target key. On the other hand, if an application needs all entries belonging to the same key, it needs to scan all runs in an LSM-tree. In either case, a lookup operation usually needs to scan more than one run for the target key and consolidate the results as the final output. The efficiency of key lookup in runs is thus crucial to LSM-tree lookup performance.

Many LSM-tree implementations [45, 53] store the runs on disk as a data structure

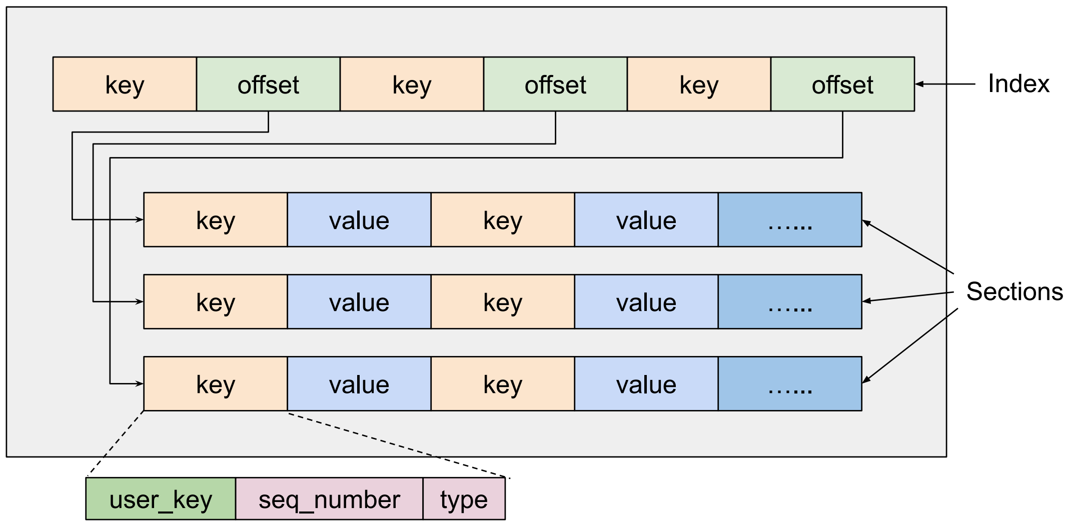


Figure 6.1: SSTable stores keys and values in a row-oriented layout

called sorted-strings table (SSTable). We demonstrates its structure in Figure 6.1. SSTable stores the sorted key-value pairs in an interleaved manner, with each value following its corresponding key. Each key in SSTable consists of three pieces: the user key, a sequence number that monotonically increases with each operation, and a type marking if the entry is an insert or a delete. To facilitate efficient lookup, SSTable splits the data into sections and provides a sparse index referring to the header of each section. When performing a point look up with a specific key, the SSTable first uses the sparse index to locate the section containing the target entry, then executes a sequential search in that section to find the key and the corresponding value.

SSTable stores data in a way similar to the row-wise layout in relational databases, in which fields belong to the same row are stored consecutively. In an SSTable, each row consists of four fields: user key, sequence number, type and value. We notice that while a lookup operation on an SSTable only need to access the user key column, it has to read out the entire row then discard other fields. As an alternative to row-wise storage layout, columnar layout handles this scenario more efficiently. A columnar layout stores fields of the same column consecutively, allowing the query to access each column independently and reduce unnecessary cost.

Storing a column together also allows the system to compress data with various lightweight encoding schemes, reducing the storage size and improving query efficiency [1]. Lightweight encoding is a family of compression algorithms featuring fast compression and decompression speed and low CPU utilization. Typical lightweight compression includes bit-packed encoding, run-length encoding, and delta encoding. Many popular compression algorithms, such as Gzip and Snappy, are blocked-based. These algorithms compress and decompress data in blocks. To access a single compressed record, we need first to decompress a large block in which the record resides, then fetch the record from the decompressed data. Many lightweight encodings are record-based and maintain a one-on-one mapping between a record and its corresponding bits in the compressed result. This mapping allows readers to locate and decompress a target record in a compressed data block, saving the effort of processing adjacent records irrelevant to the query. With advanced algorithms [67], the system can also query the encoded data without decoding them. Thus, queries on lightweight encoded data are efficient with data skipping and direct queries on encoded data.

As we have seen, lookup operations in an SSTable only access the user key fields. Therefore, storing the key in a separated column and applying lightweight encoding on it should help improve the lookup performance of an LSM-tree. However, we notice that although LSM-tree and columnar databases share some similarities, there is still a big difference in their application scenarios. In columnar databases, queries are mostly read-only, and data rarely changes after the initial import. As the encoding operations on data columns is a one-off process, we have a high tolerance to the latency it brings. While in an LSM-tree, data runs are frequently merged during the update step. If we encode the run, each merge will involve decoding and re-encoding operations to the data. Thus, the extra encoding/decoding step can impact the overall performance of an LSM-tree.

Previous research [34, 35, 36] points out that in an LSM-tree, data access pattern is asymmetric among different levels. Most merges happen in lower levels. In data runs at higher levels, lookups dominates the operations. With this finding, we propose a hybrid structure in the runs of an LSM-tree. For runs at lower levels, where merge happens frequently, we use a merge-friendly data structure. For runs at a higher level, we use a lookup optimized columnar structure. This hybrid structure addresses the efficiency of both the lookup and merge operations and leads us to the design of CoLoM (**Co**lumnar **Lo**g Structure **M**erge Tree). It explores using columnar layout and encoding to expand LSM-tree design space and improve query efficiency. CoLoM consists of three parts from the bottom up.

VEST:Columnar Key-Value Storage Format We propose **VE**rtical **S**orted **T**able (VEST), a key-value storage format that persists sorted entries on disk in columnar layout. VEST stores the keys, control data, and values in separated columns ordered by the keys, then encodes each column with lightweight encoding algorithms to reduce storage size and improve lookup efficiency. A lookup operation first performs a binary search on the key column to find its index, then locates the control data and value in the corresponding columns with the index. A merge operation of two VESTs first materializes key-value pairs from the old VESTs, performs a merge sort, then stores the pairs in a new VEST. When searching on integer keys, VEST can be up to 3x more efficient in key lookups compared to the widely used SSTable, while maintaining a similar performance in the merge operations. VEST also supports encoding algorithms on the columns.

LSM-tree Implementation that Supports Varying Storage Format at Each Level We design an LSM-tree that allows users to choose different storage formats at each level. With fewer merge operations and frequent lookup operations at higher levels, one can use a storage format that optimizes for lookup. At lower levels where merge happens frequently, one can choose a storage format that excels in scan and materialization operations.

CoLoM: Adaptively Storage Selection at Each Level We propose CoLoM, a fast and self-tunable LSM-tree-based key-value store. CoLoM uses a cost-based optimizer to decide the storage format to use for each LSM-tree level and makes sure the entire system runs with high efficiency. It models the cost of read/write operation, computes the expected cost regarding a given workload, describes the cost as an optimization problem concerning model choice for each level, and solves the problem to find the optimal assignment.

Our contributions include the following:

- We show that columnar layout and encoding could greatly improve lookup efficiency in disk-based key-value storage.
- We introduce a columnar LSM-tree that supports a hybrid design of various data layouts and encoding on each level.
- We design CoLoM, a key-value store that analytically finds the optimal design of a columnar LSM-tree regarding the application workload.
- We implement the system on LevelDB and provide an extensive experimental evaluation.

In the rest of the chapter, we present the design of VEST in Section 6.1. We introduce the cost model in Section 6.2, and the evaluation results in Section 6.3. We conclude the chapter in Section 6.4.

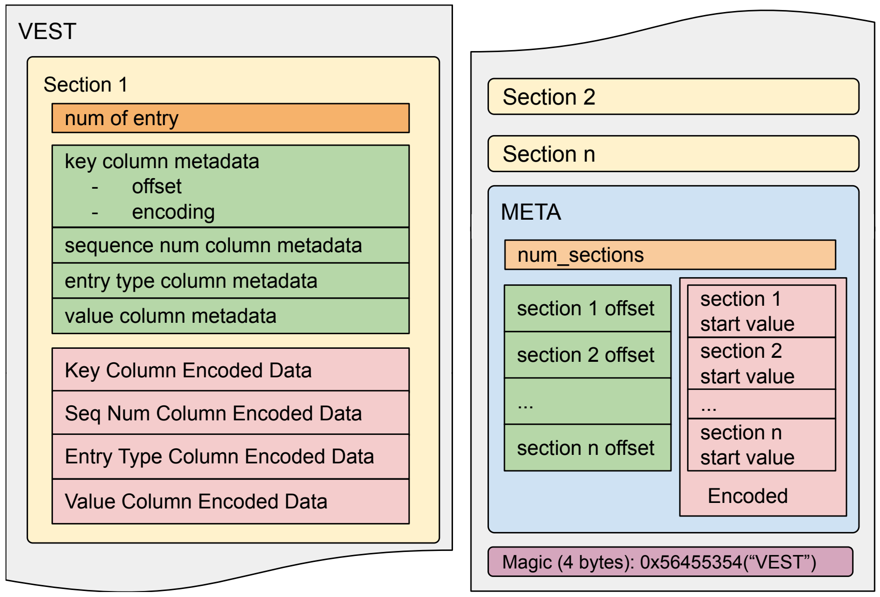


Figure 6.2: VEST Storage Format Layout

6.1 VEST: Sorted Runs with a Columnar Layout

We design VEST, a columnar storage format for LSM-tree. As a prototype, we provide a VEST implementation in LevelDB [53], a popular open-source LSM-tree implementation. It is also straightforward to migrate VEST to other LSM-tree implementations.

We demonstrate the structure of a VEST file in Figure 6.2. A VEST file consists of multiple sections storing key-value pairs, a footer of metadata, and a magic number for applications to recognize the format. Metadata stores the number of sections in this file, as well as the offsets of each section. It also keeps the first key of each section as a sparse index to facilitate faster lookup.

Each section starts with the number of entries it contains, followed by the columns' metadata. VEST includes two columns for keys and values, as well as some additional columns for the control data specific to each LSM-tree system. We develop the prototype of VEST based on LevelDB, in which the control data includes a monotonically increasing sequence number and a entry type flag marking deletion operations. Thus, each section of a VEST file for LevelDB contains four columns: the user key, sequence number, entry type, and value. The metadata of a column records the encoding type this column uses and the data offset relative to the beginning of a section.

VEST chooses a proper encoding for each data column to facilitate better space utilization and efficient data access. As an example, if the dataset only contains keys of integer type, VEST computes the delta between the keys and the smallest value of the section and bit-packs the deltas. The entry type is a binary value, and relatively few records have values of 1 (which marks a deletion operation), VEST uses run-length encoding on it. Users can also manually indicate the encoding type for each column.

A point lookup in VEST first executes a binary search on the start value of sections in the metadata to finds the section containing the key. It then locates the offset of the target section and performs a second binary search on the key column to locate the index of the target key in the section. Finally, it uses the index to read data from all other columns, then materializes the final output. A range lookup follows similar steps to search for the first key in the range and maintains an iterator to read the following entries.

The merge of two VEST files is implemented as a two-level merge sort. The first level of merge happens on sections. We read the sections in sequence from both files and see if they overlap. If there is no overlap between the two sections to be merged, we copy the smaller section into the output, read the next section from the corresponding VEST, and repeat the steps. If the two sections overlap, we go two the second level of merge that happens on entries. In this level, we sequentially materialize all entries from the two overlapping sections, perform a merge sort on them, and write the merged result to the output.

We add the support to VEST into existing LSM-tree solutions to build CoLoM. CoLoM supports using different storage formats at each level of an LSM-tree. It introduces an abstract storage layer, which implements all the storage APIs of the original LSM-tree and uses this layer to replace the original storage layer in the LSM-tree seamlessly. The abstract layer contains readers and writers corresponding to VEST and the original storage format used by the LSM-tree implementation, which we refer to as "base format".

When the LSM-tree reads a data file, the abstract layer recognizes VEST files and base files by looking for the magic number at the end of the file and relays the requests to the VEST reader or the base reader. When CoLoM writes a table at a level, it consults an assignment table to determine what storage format to use at this level and invoke corresponding writers. To build such an assignment table, CoLoM employs a cost model to estimate the average cost of the system and optimize the cost concerning a given workload.

L	The number of levels
T_i	The capacity ratio between level i and $i+1$
M	The boundary of leveling and tiering. For $i >= M$ we use leveling and tiering otherwise
B_i	The number of entries in a block at level i
f_i	The false positive rate of Bloom Filters at level i
$p(x)$	The point lookup time in a block with size x
r	The average sequential scan time of a block
$u(x)$	The merge time of a block with size x

Table 6.1: Symbols Conventions used in Cost Analysis

6.2 Cost Analysis and Optimization

This section describes the cost model used by CoLoM to generate the assignment table. We begin by modeling the cost of each operation in an LSM-tree and estimating the average cost with regards to a given workload as a weighted sum of the operation costs. We then describe the file format assignment at each level as an optimization problem. By solving it, we obtain the assignment table.

6.2.1 Cost Analysis of Operations

We analyze the worst-case write cost, point lookup, and range lookup in an LSM-tree, and derive the average cost of a given workload. We list the symbols used in this section in Table 6.1. In this analysis, we make the following assumptions: bloom filters for all runs reside in memory and have negligible access cost.

Point Lookup The worst case of a lookup occurs when it searches every run of an LSM-tree. This situation happens when the target key is not in the LSM-tree or exists only at the top level. The bloom filter at level i has a probability of f_i to give a false positive, which leads to a wasted lookup in the run, taking time $p(B_i)$. The total cost for processing all runs at a level with tiering policy is then $T_i f_i p(B_i)$, and for one with leveling policy is

$f_i p(B_i)$. The expected cost of a worst-case point lookup operation is

$$P = \sum_{i=0}^{M-1} T_i f_i p(B_i) + \sum_{i=M}^{L} f_i p(B_i) \tag{6.1}$$

Range Lookup The worst case of a range lookup accesses every run of a LSM-tree, when each run contains at least one key within the range. It involves a point lookup of the start key, and a sequential scan until reaching the end key. Here we assume all range lookups are short range lookups, in which all the keys being read fall in the same storage block. The sequential scan on a run thus takes a constant time r. The cost of a range lookup can be represented as follows

$$R = \sum_{i=0}^{M-1} T_i(p(B_i) + r) + \sum_{i=M}^{L-1} (p(B_i) + r) \tag{6.2}$$

Update A single update operation to a LSM-tree not only has immediate effect to the memory layer, but also participates in subsequent merge operation. We thus compute the cost of an update as amortized cost among all levels. In the analysis we assume that an update always targets at a key in the top level and participates in merges L times. This is the worst case and serves as an upper bound of the cost of an update operation. At level i with tiering policy, the entry participates in merge only once, and takes time $u(B_i)$, with each entry contributing $\frac{u(B_i)}{B_i}$. At a level with leveling merge policy, the entry participates in merge T_i times and contributes to the cost $\frac{T_i u(B_i)}{B_i}$. The total amortized cost of one update operation is

$$U = \sum_{i=0}^{M-1} \frac{u(B_i)}{B_i} + \sum_{i=M}^{L} \frac{T_i u(B_i)}{B_i} \tag{6.3}$$

Approximation of Lookup/Merge Time

The point lookup time $p(x)$ and merge time $u(x)$ used in cost estimations are functions of block sizes, and we now discuss how to approximate their output. A point lookup on a sorted data block of size n performs a binary search, with time consumption $O(\log n)$. We approximate it with a log function $p(x) = \epsilon \ln(x) + \eta$. A merge results in a data block of size n has a time consumption of $O(n)$, and we approximate it with a linear function $u(x) = \mu \cdot x + \xi$. The ϵ, η, μ and ξ here are coefficients that vary between different block implementations, and we determine their values using linear regression.

6.2.2 Optimization

We consider a workload W consisting of α point lookup, β range lookup, and γ updates. From the discussion in previous section, we want to choose appropriate data structures for each level to minimize the expect cost $C(W)$ of the workload

$$C(W) = \alpha P + \beta R + \gamma U \tag{6.4}$$

where P, R and U are described in Equations (6.1) to (6.3).

To minimize $C(W)$, we define L binary variables $k_i \in [0, 1]$. $k_i = 1$ means using lookup optimized mode in level i, otherwise using balanced mode. We denote the time of performing a point lookup, range scan, and merging a block in lookup optimized mode as $p_l(x), r_l$ and $u_l(x)$. In balanced mode, they are $p_b(x), r_b$ and $u_b(x)$. For each level i, we have

$$p(x) = k_i p_l(x) + (1 - k_i) p_b(x)$$

Similarly, we represent R_i and U_i with k_i. We can now rewrite Equation (6.4) as an integer linear programming(ILP) with regarding of k_i.

$$\begin{aligned} \min_{k_i} \quad & C(W) \\ \text{s.t.} \quad & k_i \in \{0, 1\} \end{aligned}$$

and solve it with an ILP solver to obtain the values of k_i.

6.3 Evaluation

6.3.1 Implementation and Setup

Implementation We implement a prototype of CoLoM based on LevelDB [53], a popular open-source key-value store. LevelDB is a tiered LSM-tree implementation. When merging two data runs, it uses a table builder to write new data runs to disk. We modify the table builder to take an additional parameter indicating whether the new table should use VEST instead of SSTable. When a merge happens, CoLoM consults an assignment table to determine what data format to use based on the level where the merge operation occurs.

CoLoM stores the assignment table as a disk file and provides an executable to generate the assignment table based on system parameters and workloads described in Section 6.2. CoLoM reads the model file to determine what storage format to use at each level. By default, the cost model uses a capacity ratio of 10 and a Bloom Filter false positive rate of 1%. Both are default values in LevelDB. We show the values of these parameters in Table 6.2.

Experimental Platform The experiment platform has two Intel(R) Xeon(R) Gold 6126@2.60GHz, 192G memory, and 250G SATA SSD. It runs Ubuntu 20.10 with kernel version 5.4.0-77. We implement the CoLoM prototype based on LevelDB version 1.23 compiled using GCC 9.3.0 with `-O3`. The cost model is optimized using IBM ILOG CPLEX [31] version 12.10.0.

We compare CoLoM against competitors using Yahoo! Cloud Serving Benchmark(YCSB, [30]) version 0.17.0. We develop YCSB clients for both CoLoM and LevelDB that access the C++ libraries using JNI. YCSB runs on the same hardware platform with OpenJDK 11.0.11. We use a dataset of 10M records of size 11.2G before compression. We first load the full dataset into the target systems, then run the workloads on them. We use the default workloads setting from YCSB, including mixtures of lookup, insert, update, and delete operations.

L	7	Maximal 7 levels
T_i	10	Capacity Ratio
M	8	Not using Leveling
B_0	15000	Based on File size of 2M and value size of 128 Bytes
f_i	1%	Bloom Filters use 10 bits for each entry

Table 6.2: Parameter Values used in Cost Model Estimation

6.3.2 Operation Microbenchmarks

In this section, we compare the performance of VEST and SSTable with a series of micro-benchmark experiments. All experiments in this section read and write data against a memory buffer and do not involve disk I/O.

Point Lookup We start with the point lookup performance and demonstrate the result in Figure 6.3. We generate sequential integer keys and uniformly pick the lookup target. In the left subfigure, we fix the number of entries to one million and vary the value field length from 16 bytes to 4096 bytes. We see that 1) VEST supports 2-3x faster point lookup than SSTable and 2) VEST has a stable lookup performance with larger values, while SSTable's lookup performance deteriorates with larger value lengths. In the right subfigure, we keep the value field size to 128 bytes and vary the number of entries from one million to four million. We see that both VEST and SSTable become slower with more entries, but the lookup time consumption of VEST increases much slower than SSTable.

The advantage of VEST over SSTable on point lookups primarily comes from the columnar layout and encoding VEST employs to store data. A Lookup operation in the LSM-tree only involves key comparisons. SSTable interleaves the keys and values into the same memory region, forcing a lookup to load both key and values into CPU caches, but needs to eliminate the value fields when performing key comparisons. In contrast, VEST uses a columnar layout that separates keys and values into different memory regions and applied encoding to reduce their size. The lookup on VEST only needs to access the key column and achieves better cache efficiency as it loads more entries into caches for comparison. For example,

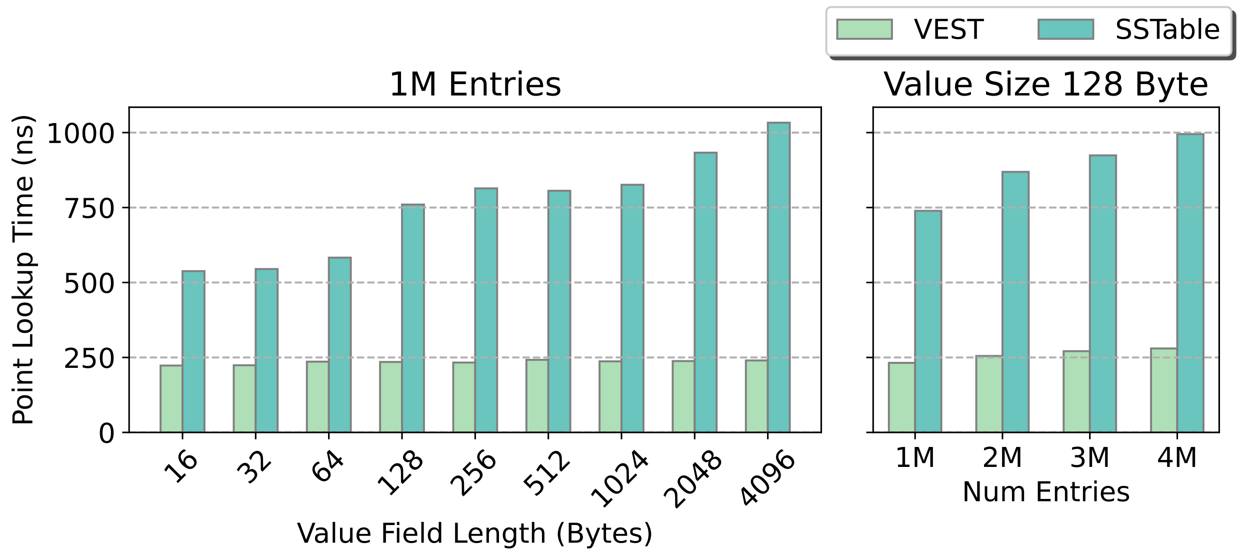


Figure 6.3: Point Lookup Performance of VEST and SSTable. VEST is 2x-3x faster than SSTable, and more stable with larger value sizes.

on a dataset with integer keys and a value size of 400 bytes, VEST can load an order of magnitude more entries into cache compared to SSTable.

When the lookup obtains the target index from the key column, it uses the index to perform data skipping on the value columns to fetch the value. The time consumption on data skipping is relatively independent of the value field size. VEST thus maintains a more stable performance compared to SSTable when dealing with large value fields.

Range Lookup We next compare the range lookup performance on VEST and SSTable. We adopt a similar scenario in the point lookup experiment: load the data table with sequential integer keys and uniformly choose the lookup target. We then scan a range of at most 100 keys starting from the target key. We show the result in Figure 6.4. In the left subfigure, we fix the number of entries to one million and vary the value field length. In the right subfigure, we fix the value size to 128 bytes and vary the number of entries. As the range lookup contains a point lookup and a range scan, we subtract the result of Figure 6.3 from the result of Figure 6.4, and see that the average time consumption of scanning a single entry in VEST is 11.5 ns, while that for SSTable is 14.5 ns. These numbers are relatively stable across different parameter settings and comply with previous studies on the benefit of

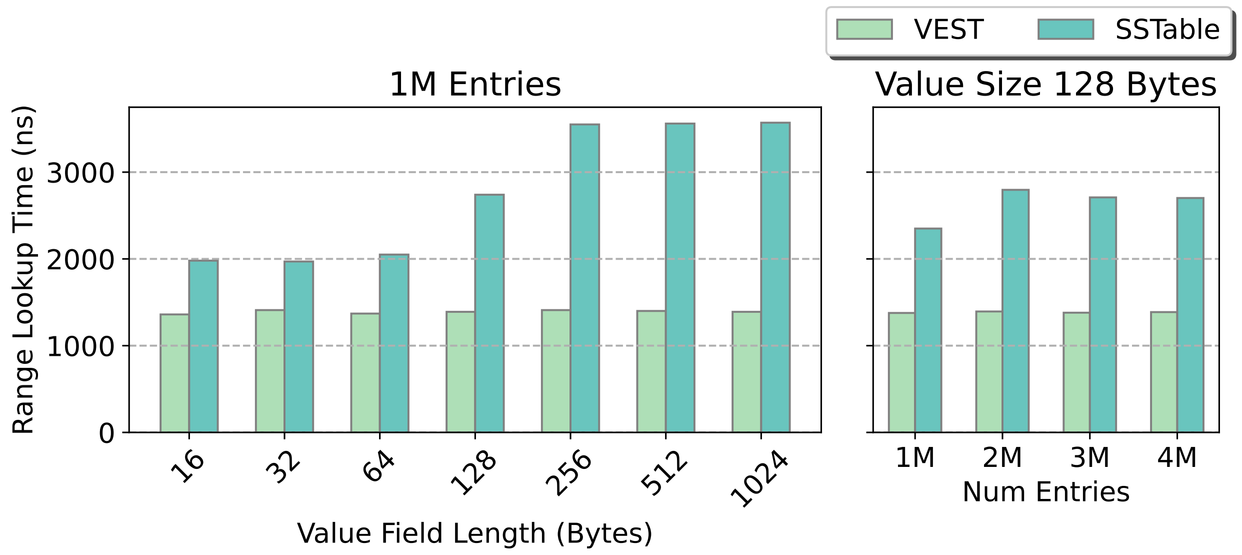


Figure 6.4: Range Lookup Performance of VEST and SSTable. VEST is 1.5-2x faster than SSTable, and more stable with larger value sizes.

the columnar layout [114]. As a result, we see an overall performance improvement of 1.5 2x of VEST over SSTable on range lookups.

Merge We then compare the merge performance between VEST and SSTable. We prepare two data runs of the same number of entries, with their key ranges not overlapping. We perform a sort merge on the two input data runs and write the result to a new data run in the same storage format. We present the result in Figure 6.5. In the first subfigure, we have each input data run containing 35K entries and vary the data run size by varying the value field length. We see that VEST consumes about 50% time comparing to SSTable when performing a merge operation. In the second subfigure, we fix the value field size to 128 bytes and vary the number of entries in each input data run. In the third subfigure, we fix each input data run to 2MB and vary the value field size. In all scenarios, we see that VEST has a 5%-20% performance improvement comparing to SSTable.

We notice that in the second subfigure, while VEST's merge performance is linear to the number of entries as expected, SSTable's merge performance is close to 2x of that of VEST when the number of entries <= 30K, and is ~1.1x when the number of entries is larger than 30K. A further evaluation demonstrates that with a value size of 128 bytes, when the

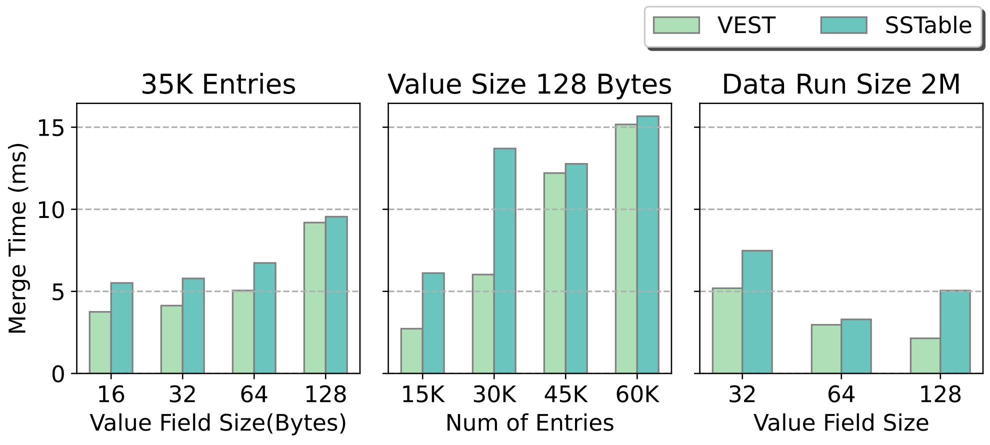


Figure 6.5: Merge Performance of VEST and SSTable. VEST is 5%-20% faster than SSTable when perform merging.

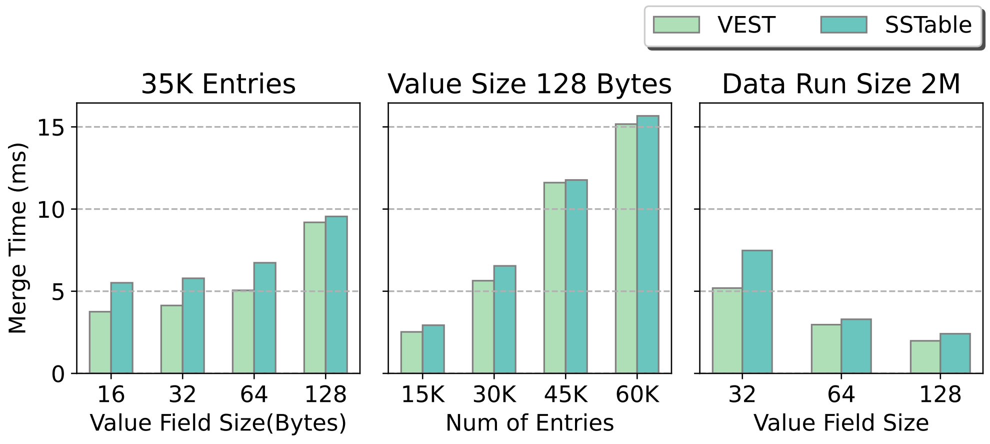


Figure 6.6: Merge Performance of VEST and SSTable, only considering user CPU time.

number of entries is no larger than 32410, SSTable takes 2x time of VEST, and otherwise only slightly slower than VEST. We profile the program under the two inputs, and notice that with the number of entries 32,410, there are ∼270,000 page faults, and about half of the CPU time is spent in kernel space. With the number of entries 32,411, the number of page faults drops to ∼25,000, and the kernel CPU time drops to close to zero. In Figure 6.6, we redraw the figures of merge time consumption, including only user space CPU time. We can see that the merge time of both SSTable and VEST is linear to number of entries. We are still investigating the root cause of this observation.

Table Size and Compression SSTable supports applying Snappy [54] compression al-

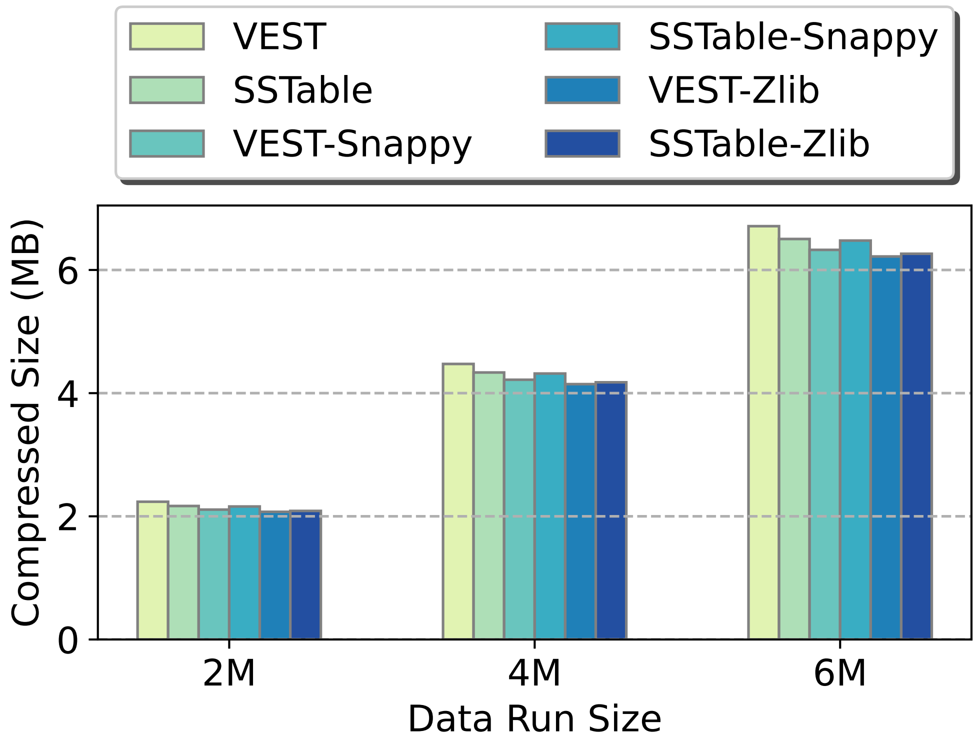


Figure 6.7: Comparison of the disk file size of VEST and SSTable, with and without Snappy. VEST is slightly larger than SSTable, and the compression approaches has little impact on both formats.

gorithm on the data block. We also implement Snappy support in VEST. As an extension, we further implement Zlib compression in both formats and study how compression impacts both formats' merge performance. The dataset contains values of a fixed size of 128 bytes, filled with random characters. We vary the number of entries and show the result in Figure 6.7. We see that VEST has a slightly larger file size compared to SSTable as we do not apply any encoding on the sequence number column. Applying bit-packed encoding on sequence numbers could reduce the file size of VEST to be slightly smaller than SSTable, while the difference is still insignificant. Nevertheless, Snappy and Zlib have negligible impact on the file size on both table formats, primarily due to the incompressible random strings in the value fields. Considering the non-negligible compression and decompression operation overhead, we avoid using compression in all our experiments.

	VEST	SSTable
ϵ	34.98	179.46
η	-251.67	-1740.2
μ	409.41	429.87
ξ	-7.99×10^6	-5.01×10^6

Table 6.3: Parameter Values from Linear Regression

6.3.3 Running YCSB with LevelDB and CoLoM

YCSB is a popular benchmark for key-value databases, and we use it to compare the performance of LevelDB and CoLoM under different workload scenarios. YCSB supports running against different DBMS using YCSB clients. A typical client connects to the DBMS server through the network and relays the operation commands from YCSB to the DBMS. However, LevelDB and CoLoM do not have a client-server infrastructure. As a result, we implement YCSB clients for LevelDB and CoLoM using JNI. First, we build shared libraries from LevelDB and CoLoM, then load the libraries into the clients upon initialization and create a DB instance. During the benchmark execution, each YCSB operation is passed to this DB instance through JNI for execution.

YCSB assumes the target DBMS supports multiple value fields for a key, while LevelDB and CoLoM only supports a single value field for each key. To bridge the gap, we concatenate the `field_name:value` pairs into a single string and prefix the field names and values with their lengths, allowing us to extract the entries from the encoded string.

We run the core workloads in YCSB with 10M records and 1M requests and report the results in Figure 6.8. CoLoM outperforms LevelDB on all workloads.

6.3.4 Cost Model Estimation

We use linear regression to estimate the parameters and obtain the value shown in Table 6.3 of both SSTable and VEST using 4-byte integer key, 128-byte value as an example.

We use these numbers to update the cost model and predict the system throughput

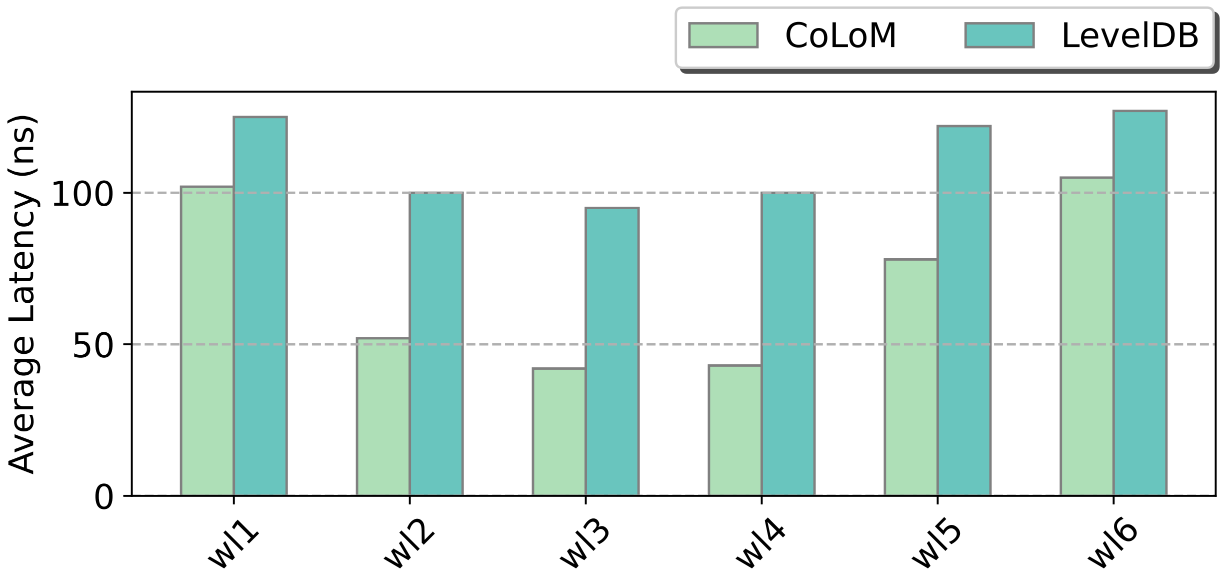


Figure 6.8: Running YCSB with CoLoM and LevelDB.

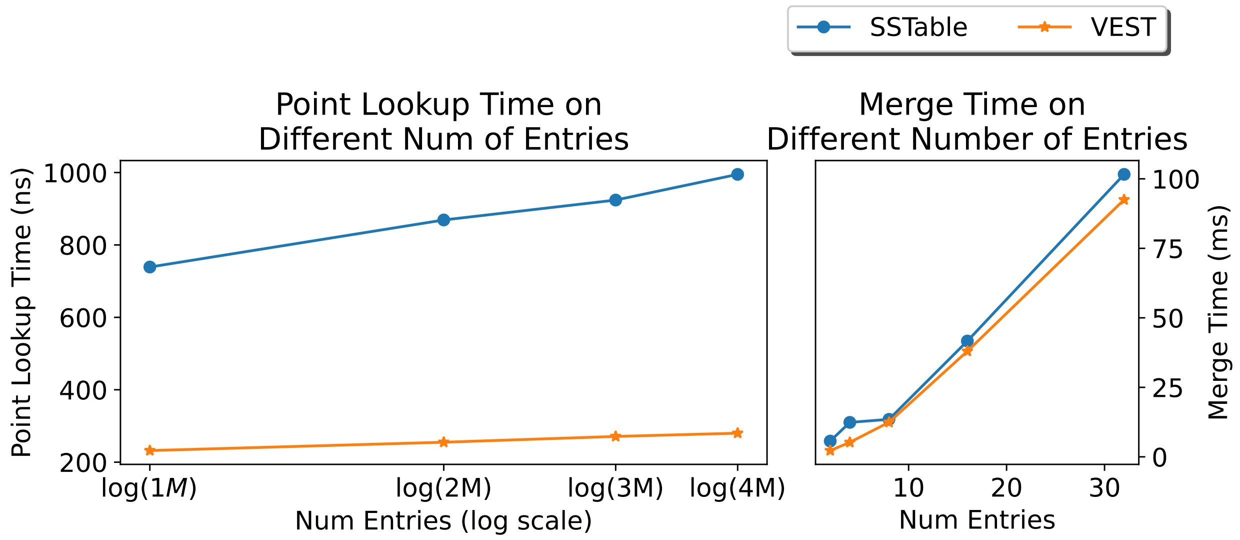


Figure 6.9: Use Linear Regression for the Model Parameters

when assigning different storage models to each LSM-tree level. VEST excels SSTable in all operations. Not surprisingly, the cost model chooses VEST for all levels.

We further study the boundary conditions that make the CoLoM cost model favor one storage format over the other. In other words, how much should we slow down VEST before the cost model favors SSTable? For example, we use a workload with 50% point lookups and half updates and assume all the values have 128 bytes. All the other parameters are the same as shown in Table 6.2.

We simulate two situations. In the first one, we keep stable the point lookup performance of VEST, and alter VEST's merge performance by introducing a coefficient f, and setting $\mu_{VEST} = f \cdot \mu_{SSTable}$, $\xi_{VEST} = f \cdot \xi_{SSTable}$. By varying the value of f, we can simulate the situations that VEST performs merge operations faster or slower compared to SSTable. This experiment can simulate the case when VEST uses more advanced string compression techniques such as RePAIR-Front Coding [80] on the value column to trade materialization/encoding time with space efficiency. Next, we keep the merge performance of VEST stable and vary its point lookup performance by setting $\epsilon_{VEST} = f \cdot \epsilon_{SSTable}$, $\eta_{VEST} = f \cdot \eta_{SSTable}$. This situation simulates a storage format that supports efficient compression and decompression, but need to decompress the whole data block every time it performs a key lookup.

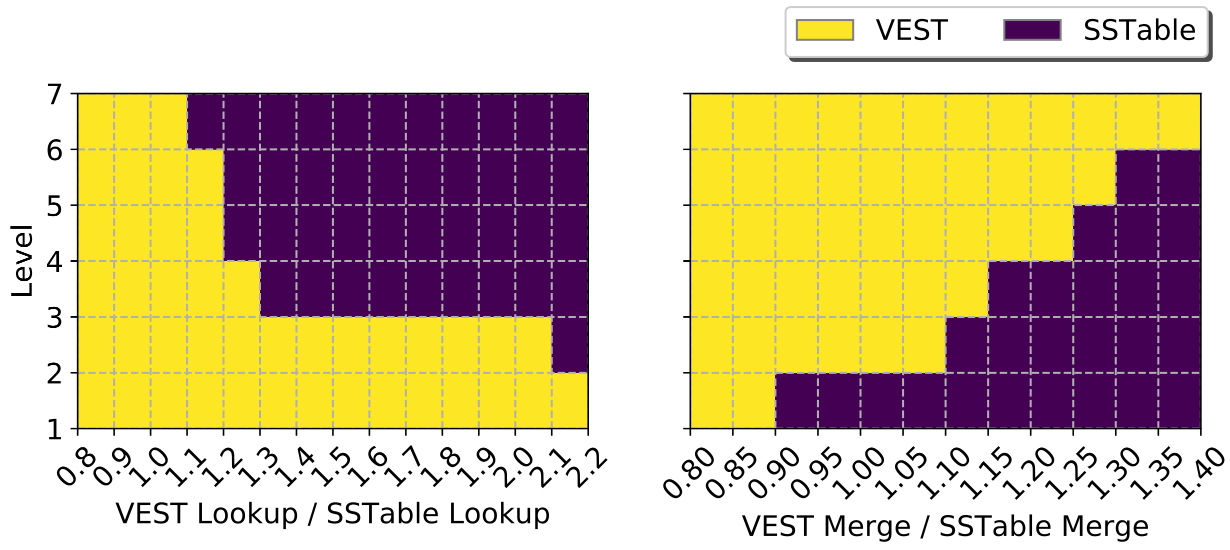


Figure 6.10: Use Cost Model to simulate CoLoM behavior with different performance characteristics of VEST

6.4 Conclusion

LSM-tree is a storage solution widely used to back many key-value data stores. Popular LSM-tree implementations use SSTable, a row-wise storage to persist the key-value entries on disk. In this paper, we propose VEST, a columnar storage solution that outperforms SSTable in both lookup and merge operations. We build CoLoM, an LSM-tree implementation based on LevelDB. CoLoM supports both storage formats of VEST and SSTable. It uses an adaptive storage selection to choose appropriate storage formats for each storage level. Experiments show that CoLoM achieves better efficiency compared to LevelDB.

CHAPTER 7
RELATED WORK

7.1 Compression and Encoding

Lightweight encoding, such as dictionary encoding, and LZ77 [131] are two popular families of compression algorithms adopted in columnar stores. Lightweight encoding features high throughput and *in situ* data filtering. LZ77 features higher compression ratios but requires decompression before evaluation. By extracting sub-attributes and applying a lightweight encoding, we achi-eve a compression ratio comparable to LZ77 compression, while preserving all the benefits of lightweight encoding.

Popular byte-level compression techniques, such as Gzip and Snappy, both belong to the LZ77 family, which utilizes a sliding window on an input data stream, looking for strings that contain a recurring prefix, and encodes each string as a reference to the previous occurrence. Gzip applies Huffman encoding to the reference stream for a better compression ratio [39] and Snappy skips that step for higher throughput. LZ77 [131] has a theoretical guarantee for a good compression ratio. However, it is a sequential algorithm and needs to decompress an entire data block before the original data can be accessed. This brings a high latency for accessing the compressed data.

Lightweight encoding is a family of entry-level compression techniques. It transforms each data entry (an integer, a line of text, etc.) in the input to a shorter representation. Popular lightweight encoding schemes include dictionary encoding, bit-packed encoding, delta encoding, run-length encoding, and their hybrids [114, 22, 132, 78]. Lightweight encoding has very low CPU consumption as the operations are usually simple and can often be performed in parallel [67]. Lightweight encoding maintains entry boundaries during compression, allowing access of compressed entries without decoding the entire data block, and direct predicate execution on compressed data, skipping decompression [67].

Another related research area is string dictionary compression [94, 79]. Research here applies dictionary encoding to the data, then compresses the dictionary entries using methods such as prefix-coding and delta encoding. Such approaches also address repetitions of substrings in a string attribute, but they help little for improving query execution. Additionally, these methods rely on sorting the string dictionary for prefix-coding.

7.2 Encoding in Databases

Application of encoding techniques has been studied for various components in databases, including data table [64, 28, 106, 77], data column [114, 98], index [18, 74], data dictionary [94], database duplication [124], and search trees [126].

Selecting the proper encoding scheme for a database system is a trade-off between size reduction and CPU overhead in the encoding/decoding process. Classical byte-oriented encoding techniques such as GZip and Snappy have been widely supported in various DBMS systems [64, 28, 114, 13, 27]. However, studies [132, 1] suggest that these schemes come with notable CPU overhead from decompression and may significantly impact system performance. Lightweight, attribute-level compression, such as run-length and bit-packed encoding, can be beneficial to query intensive systems.

Columnar databases, such as C-Store [114] and MonetDB [60], physically consecutively persist attributes, and allow a higher performance of lightweight encoding. Previous works [83, 21] also show that for specific data sets, lightweight encoding achieves a comparable compression ratio with far lower CPU time than GZip.

Reasonable size reduction, significant low CPU overhead, and *in-situ* query execution make lightweight encoding algorithms more favorable than byte-oriented compression in columnar data stores [2].

7.3 Query Execution on Encoded Data

Compression-aware database design is necessary for query execution on compressed data. Chen et al. [28] design a cost-based optimizer for compressed database tables, and Kimura et al. [74] propose an algorithm for selecting compressed indices for a set of queries under a limited space budget.

Hardware acceleration also demonstrates high potential in speeding up the decoding operation. Willhalm et al. [122] uses SIMD instruction to speed up the decoding process for bit-packed data. Jiang et al. [67] proposes a SIMD-based algorithm for decoding delta-encoded data. Rozenberg et al.[109] develops Giddy, a library for executing fast decoding algorithms using GPUs. Fang et al. introduce UDP [46], a co-processor for data extraction and transformation tasks in columnar encoding and compression. Variations of encoding formats that are optimized for hardware execution, such as VarInt [37, 113], Horizontal Bit-Interleaving [87, 103], and SIMD-Delta [83] are also proposed.

Lightweight encoding has an advantage over byte-oriented compression algorithms that they preserve attribute boundary during encoding [2, 1], and allow algorithms [3, 67] to query on encoded data without decoding, significantly reduce the CPU overhead and enable more efficient query execution. Damme et al. utilize these features of lightweight encoding to design MorphStore [32]. This in-memory query engine compresses intermediate results with lightweight encoding, reducing memory footprint and improving query efficiency.

7.4 Cost Estimation and Encoding Selection

We can estimate an encoding scheme's efficiency from both space and time aspects, e.g., the compression ratio and encoding/decoding overhead. Popular approaches for estimating compression ratio include mathematical modeling [28, 94], regression on statistical features [18], and random sampling [59, 74]. Kimura et al. [74] propose using a bipartite graph between column and index to deduce compressed index size. For en/decoding overhead, most previous work [28, 74] model it as a weight in addition to normal access cost.

Encoding selection tackles a relevant problem of choosing an efficient encoding scheme for a given a data table or column. Abadi et al. [2] introduce a hand-crafted decision tree for encoding selection on a given dataset based on experience and global knowledge of a dataset (i.e., cardinality and if a column is sorted). Lemire et al. [83] focus on integer data and propose rules to choose between PFOR and bit-packed encoding.

In practice, many implementations solve the problem by hard-coding a “not too bad” default encoding per data type. Apache Parquet [13] and CarbonData [10] uses dictionary encoding for all data types and will fall back to some default encoding if the dictionary size exceeds a preconfigured limit. Apache ORC [12] uses RLE for integer and Dictionary-RLE for string types. Apache Kudu [11] uses a dictionary for string type and bit shuffle for all other data types.

VEST proposes using a neural network-based learning approach to rank and select from various encoding schemes. Learning to Rank is a widely adopted approach in data mining and information retrieval, in which learning algorithms are applied to datasets of labeled documents [25], document lists [26], and labeled features [40], to learn a utility function evaluating the importance of each target document. Neural network-based learning to rank approach [25, 129] has demonstrate great potential in various domain applications such as e-commerce search [70], image annotation [120], and behavior analysis [100]. VEST’s approach can be analogous to the learning to rank problem.

7.5 Pattern Inference and Data Extraction

In general, inferring patterns from samples is a program synthesis technique known as Programming-by-Example (PBE). A PBE algorithm automatically analyzes existing examples and generates programs that can be applied to new examples. PBE has many applications in data processing tasks that involve large amounts of input data with indeterminate formats, such as in structured data extraction [14, 85, 47, 82, 107, 50], table transformation [56, 17, 68], and entity augmentation [118, 127].

In PBE tasks, users often provide both input and output example pairs [82, 17, 68]. By treating input and output examples as states in a search space and by defining data transformation operations as transitions between the states, such a problem can efficiently be converted to a search problem. As the number of available states is usually exponentially large, pruning techniques [82] and heuristics [68] are usually employed to facilitate the search process.

For a large input dataset, as is in our case, it is often impractical for a user to provide output examples corresponding to each input. The algorithms introduced by PADS [47] and Datamaran [50] both follow a search-rank-prune pattern process to automate pattern inference. Similarly to the input-output example case, input examples are mapped to a search space as the source state and transitions are defined between states. However, instead of searching for a given target, the algorithm computes all reachable states from the source, ranks them with a custom scoring function, and prunes the states with the lowest scores. This process is repeated until a reasonably good target state is discovered.

Unfortunately, such search-rank-prune processes are often time-consuming due to a large number of potential states. Instead of searching all possible states, in PIDS, we use a greedy search approach where, with the use of a heuristic function, we evaluate all possible transitions from the current state, choose the transition with the maximum gain, and ignore the rest. By carefully designing the transition rules and heuristic function, our approach becomes very efficient while achieving good accuracy performance. While previous algorithms extract structures from ad-hoc unstructured log data, they have to make many assumptions on the input data. Instead, PIDS targets data from the same attribute in a relational database. Besides, PIDS employs a classifier to filter out input data that is unlikely to contain a structure. This allows PIDS to make fewer assumptions on the input data and extract structures that are omitted by the previous methods.

Table 7.1 compares the assumptions made by PADS, Datamaran, and PIDS. We briefly

Table 7.1: Assumptions made by Extraction Algorithms.

Assumption	**PADS**	**Datamaran**	**PIDS**
Coverage Threshold	✘	✔	✘
Non-overlapping	✔	✔	✘
Structural Form	✔	✔	✔
Boundary	✔	✘	✔
Tokenization	✔	✘	✘

introduce the assumptions here; the detailed definitions can be found in Datamaran [50]. *Coverage threshold* assumes that the generated pattern mat-ches at least a certain percentage of samples. PIDS does not force a coverage threshold as it will generate a pattern that always covers the entire sample set. *Non-overlapping* assumes that the alphabets used in the pattern and field values are not overlapping. In other words, a character c is either part of a pattern or part of the extracted data, but not both. In PIDS, any word can be either part of the pattern or the extracted data. *Structural Form* assumes that the pattern is a tree-like structure with each node as a sub-pattern. This assumption is shared by all approaches. *Boundary* assumes that the boundary of records can be easily identified beforehand. As PIDS processes input data from an attribute, the input is well-bounded. *Tokenization* assumes that each character can be tokenized as a part of the pattern or the extracted data beforehand. PIDS does not make this assumption and determines the role of each character during inference.

7.6 SIMD in Compression and Database Acceleration

Database systems involves extremely intensive IO operations. Compression techniques greatly reduce the amount of data to be transferred at the cost of CPU occupancy upon decompression. Various studies [64, 28, 106] have been done on the impact of compression to database performance.

Columnar data stores save data from the same column in consecutive manner, allowing efficient application of encoding techniques mentioned in this paper. Encodings achieves high compression ratios with relative low CPU consumption. They also allows in-situ query execution without decoding the entire data block [2]. These advantages make them more favorable than generic compression algorithm such as GZip and Snappy in database systems.

As decoding process generally involves independent simple operations on multiple data entries, SIMD seems like a perfect solution to the problem. Willhalm et. al. [122] describe a SIMD-based algorithm for decoding and scan bit-packed data, which deals with similar problem as we do in this work. More work has been focus on designing encoding variation that work well with SIMD. Stepanov et. al. [113] introduce a SIMD version of varint-G8IU [37]. Limire et. al. propose SIMD-PFOR [83], a SIMD variation for PFOR [133] and Li et. al. demonstrate BitWeaving [87], a bit-packed encoding variation.

SIMD has many advantages comparing to other hardware acceleration alternatives. Most importantly, SIMD is built in CPU and has direct access to CPU databus and cache, avoiding data movement between different device memories. SIMD also has instruction level interoperability with control flow codes, allowing fine-grain transition between parallel and scalar mode.

SIMD based algorithms have been proposed for almost every aspects of database execution. Zhou et. al.[130] describes the general idea of using SIMD for various database operators including scan, aggregation, index scan and join. Chhugan et. al.[29] use SIMD to implement a bitonic merge network for merge sort. Ross et. al.[108] proposes to speed up hash join by optimizing Cuckoo hashtable[42] with SIMD. Jha et. al.[66] experimentally explores the hardware oblivious and hardware conscious joins on Xeon Phi platform with SIMD optimization. Other applications including vectorized bloom filter[102] and bitmap counting[93].

7.7 Performance Tuning of LSM-trees

LSM-trees have been widely used to support key-value stores [53, 45, 48, 5], secondary-indices [123, 73, 104], filters [6], and materialized views [41]. Due to its popularity, researchers invest significant effort in improving the performance of LSM-trees.

Tiering merge policy [65] is often more prevalent than leveling [96] in systems with high write pressure as tiering triggers fewer merges. However, tiering generates more disk runs than leveling, bringing slower lookups and merges. As one way to compensate for the problem, researchers propose partitioning the disk runs by key ranges. Partitioning allows lookups and merges to access only a subset of runs and improve efficiency. WriteBuffer [7] uses a vertical partitioning that groups partitions with the same key range but from different levels together. It uses a hash function to distribute keys to partitions and manage the partitions with a B+-tree. PebbleDB [105] selects the partition key range probabilistic based on the inserted keys. SifrDB [90] performs a horizontal partitioning that split each data run independently.

Some other works focus on improving merge efficiency. VT-tree [111] presents a stitching operation that detects and skips SSTables that have no key overlapping with other SSTables during merges. Zhang et al. [128] proposed a pipelined merge implementation. This approach splits a merge into three phases: read, merge-sort, and write, then builds them as a pipeline. The merge-sort phase is CPU heavy while the other two are IO heavy, allowing the pipelined merge to achieve better IO and CPU utilization. bLSM-tree [110] improves the leveling merge policy by adding a dedicated component at each level, allowing merges on different levels to execute in parallel. It also designs a merge scheduler that eliminates write stalls by limiting the memory component write throughput. dCompaction [97] introduces virtual partitions and virtual merges to defer the merge operations and executes multiple merges in batches.

After a merge occurs, the LSM-tree may suffer significant cache miss for subsequent lookups as the new merged run is not cached. Ahmad et al. [4] propose to address the issue by offloading the merge operation to remote servers and re-route the lookup to the newly merged runs chunk-by-chunk gradually to enable a cache warmup. LSbM-Tree [115, 116] keeps the post-merge old data-runs in a buffer, temporarily uses them for lookups, and gradually deletes them based on access frequency.

People also explore the opportunities of integrating LSM-trees with the recent hardware platform. FloDB [16] present a two-layer memory component to accommodate large memory. The top layer is a hash table for fast writing, and the bottom layer is a skip-list for lookup. Accordion [23] extends the idea of having layers in memory and moves the lower level disk-runs in LSM-trees into memory. cLSM [52] proposes novel concurrent algorithms for LSM-trees operations in multi-core environments. FD-Tree[86] and FD+-Tree [117] explores reducing the random writes of LSM-trees on SSD. MaSM [15] uses SSD as an internal buffer to store disk runs before moving them to slower disks. NoveLSM [69] add a layer designed for NVME storage between the memory component and disk-runs.

The performance of LSM-trees is affected by many parameters, including merging policy, bloom Filter false positive rate, and capacity ratios. The optimal values of these parameters can be different for each level and different under the various workload. Researchers propose systems that unsupervised fine-tune these parameters for each level based on given workloads. For example, monkey [34] sets different false-positive rates for bloom filters at each level. Dostoevsky [35] uses leveling merge policy at the top level and tiering merge policy for all the other levels. LSM-Bush [36] uses a different capacity ratio for each level. ElasticBF [88] dynamic adjusts the bloom filter size for each data-run based on the data access frequency.

CHAPTER 8
CONCLUSION

In this thesis, we have addressed challenges in applying lossless compression in columnar databases and key-value stores. We demonstrate that lossless compression can significantly reduce compressed file size and improve the query efficiency of data stores. First, we present PIDS, a new compression algorithm that decomposes sub-attributes from string columns to achieve a smaller compressed size and support faster queries. Next, we describe SBoost, a C++ library using SIMD instructions to filter encoded data without performing decoding. We then introduce CodecDB, a columnar database with a data-driven encoding selection algorithm and an encoding-aware query engine. Finally, we propose CoLoM, a key-value store that extends columnar layout and compression to improve LSM-trees efficiency. This work addresses storage space and access speed, the two most important factors for all data stores. With the rapid increase in data size, these two factors continue to significantly impact the data store design and development. Thus, our work will continue to play an essential role in data systems.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[3] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, Oakland, CA, may 2015. USENIX Association.

[4] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, April 2015.

[5] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. *Proc. VLDB Endow.*, 7(14):1905–1916, October 2014.

[6] Sattam Alsubaiee, Michael J. Carey, and Chen Li. Lsm-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data*, GeoRich'15, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Hrishikesh Amur, D. Andersen, M. Kaminsky, and K. Schwan. Design of a write-optimized data store. In *CERCS Technical Reports*, 2013.

[8] Apache. HBase. https://hbase.apache.org/.

[9] Apache Foundation. Apache Arrow. https://arrow.apache.org/.

[10] Apache Foundation. Apache CarbonData. https://carbondata.apache.org/.

[11] Apache Foundation. Apache Kudu. https://kudu.apache.org.

[12] Apache Foundation. Apache ORC. https://orc.apache.org.

[13] Apache Foundation. Apache Parquet. https://parquet.apache.org/.

[14] Arvind Arasu and Hector Garcia-Molina. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 337–348, New York, NY, USA, 2003. ACM.

[15] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. Masm: Efficient online updates in data warehouses. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 865–876, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 80–94, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 218–228, New York, NY, USA, 2015. ACM.

[18] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George Mihaila, Kenneth Ross, Sherman Lau, Cathy McArthur, Zoltan Toth, and Reza Sherkat. Efficient Index Compression in DB2 LUW. *Proc. VLDB Endow.*, 2(2):1462–1473, aug 2009.

[19] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 283–296, New York, NY, USA, 2009. ACM.

[20] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[21] Peter Boncz, Thomas Neumann, and Viktor Leis. Fsst: Fast random access string compression. *Proc. VLDB Endow.*, 13(12):2649–2661, jul 2020.

[22] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.

[23] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better memory organization for lsm key-value stores. *Proc. VLDB Endow.*, 11(12):1863–1875, August 2018.

[24] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[25] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to Rank Using Gradient Descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[26] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to Rank: From Pairwise Approach to Listwise Approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 129–136, New York, NY, USA, 2007. ACM.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, jun 2008.

[28] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization in Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282, New York, NY, USA, 2001. ACM.

[29] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, aug 2008.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[31] IBM ILOG CPLEX. V12.10.0: User's Manual for CPLEX. *International Business Machines Corporation*, 2021.

[32] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.*, 13(11):2396–2410, jul 2020.

[33] Wayne W. Daniel. Spearman rank correlation coefficient. In *Applied Nonparametric Statistics*. Boston: PWS-Kent, 2 edition, 1990.

[34] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Niv Dayan and Stratos Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In *ACM SIGMOD International Conference on Management of Data*, 2019.

[37] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[38] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[39] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.

[40] Fernando Diaz. Learning to Rank with Labeled Features. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, ICTIR '16, pages 41–44, New York, NY, USA, 2016. ACM.

[41] Huichao Duan, Huiqi Hu, Weining Qian, Haixin Ma, Xiaoling Wang, and Aoying Zhou. Incremental Materialized View Maintenance on Distributed Log-Structured Merge-Tree. In *Database Systems for Advanced Applications*, pages 682–700. Springer International Publishing, 2018.

[42] Erlingsson, Ulfar and Manasse, Mark and McSherry, Frank. A cool and practical alternative to traditional hash tables. In *7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, January 2006.

[43] Facebook. MyRocks. http://myrocks.io.

[44] Facebook. Presto. http://prestodb.github.io/.

[45] Facebook. Rocksdb. https://rocksdb.org/.

[46] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. UDP: A Programmable Accelerator for Extract-transform-load Workloads and More. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 55–68, New York, NY, USA, 2017. ACM.

[47] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. *SIGPLAN Not.*, 43(1):421–434, jan 2008.

[48] Apache Foundation. Cassandra. http://cassandra.apache.org.

[49] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.*, 3(1-2):1382–1393, sep 2010.

[50] Yihan Gao, Silu Huang, and Aditya Parameswaran. Navigating the Data Lake with Datamaran: Automatically Extracting Structure from Log Datasets. *arXiv preprint arXiv:1708.08905*, 2017.

[51] GNU project. GNU GZip. https://www.gnu.org/software/gzip/.

[52] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[53] Google. Leveldb. https://github.com/google/leveldb.

[54] Google. Snappy. http://google.github.io/snappy/.

[55] Google. sparsehash. https://github.com/sparsehash/sparsehash/.

[56] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[57] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *2014 IEEE 30th International Conference on Data Engineering*, pages 484–495, March 2014.

[58] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 487–498. VLDB Endowment, 2006.

[59] S. Idreos, R. Kaushik, V. Narasayya, and R. Ramamurthy. Estimating the compression fraction of an index using sampling. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 441–444, March 2010.

[60] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[61] A. Ilyas, J. M. F. da Trindade, R. Castro Fernandez, and S. Madden. Extracting syntactical patterns from databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 41–52, April 2018.

[62] Intel. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2017.

[63] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM.

[64] Balakrishna R. Iyer and David Wilhite. Data Compression Support in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 695–704, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[65] H. V. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, 1997.

[66] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proc. VLDB Endow.*, 8(6):642–653, feb 2015.

[67] Hao Jiang and Aaron J. Elmore. Boosting Data Filtering on Columnar Encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 6:1–6:10, New York, NY, USA, 2018. ACM.

[68] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming Data By Example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 683–698, New York, NY, USA, 2017. ACM.

[69] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 993–1005, USA, 2018. USENIX Association.

[70] Shubhra Kanti Karmaker Santu, Parikshit Sondhi, and ChengXiang Zhai. On Application of Learning to Rank for E-Commerce Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 475–484, New York, NY, USA, 2017. ACM.

[71] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[72] M. G. Kendall. A New Measure Of Rank Correlation. *Biometrika*, 30(1-2):81, 1938.

[73] Y. Kim, T. Kim, M. J. Carey, and C. Li. A comparative study of log-structured merge-tree-based spatial indexes for big data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 147–150, 2017.

[74] Hideaki Kimura, Vivek Narasayya, and Manoj Syamala. Compression Aware Physical Database Design. *Proc. VLDB Endow.*, 4(10):657–668, jul 2011.

[75] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[76] G. Kollios, D. Gunopulos, N. Koudas, and S. Berchtold. Efficient biased sampling for approximate clustering and outlier detection in large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1170–1187, Sep. 2003.

[77] 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. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 311–326, New York, NY, USA, 2016. ACM.

[78] N. Jesper Larsson and Alistair Moffat. Offline Dictionary-Based Compression. In *Proceedings of the Conference on Data Compression*, DCC '99, pages 296–, Washington, DC, USA, 1999. IEEE Computer Society.

[79] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Suleyman S. Demirsoy, and Kai-Uwe Sattler. Fast & strong: The case of compressed string dictionaries on modern cpus. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, pages 4:1–4:10, New York, NY, USA, 2019. ACM.

[80] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Suleyman S. Demirsoy, and Kai-Uwe Sattler. Fast i& strong: The case of compressed string dictionaries on modern cpus. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[81] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[82] Vu Le and Sumit Gulwani. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM.

[83] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.*, 45(1):1–29, jan 2015.

[84] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, nov 2016.

[85] Kristina Lerman, Lise Getoor, Steven Minton, and Craig Knoblock. Using the Structure of Web Sites for Automatic Segmentation of Tables. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 119–130, New York, NY, USA, 2004. ACM.

[86] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1–2):1195–1206, September 2010.

[87] Yinan Li and Jignesh M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 289–300, New York, NY, USA, 2013. ACM.

[88] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 739–752, USA, 2019. USENIX Association.

[89] Bernard Marr. How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. *Forbes*, May 2018.

[90] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 477–489, New York, NY, USA, 2018. Association for Computing Machinery.

[91] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *PVLDB*, 3(1-2):330–339, 2010.

[92] Micro Focus International plc. Vertica Encoding Types. https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/SQLReferenceManual/Statements/encoding-type.htm.

[93] Wojciech Mula, Nathan Kurz, and Daniel Lemire. Faster Population Counts using AVX2 Instructions. *CoRR*, abs/1611.07612, 2016.

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

[95] Raghunath Othayoth Nambiar and Meikel Poess. The Making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[97] Feng-Feng Pan, Yin-Liang Yue, and Jin Xiong. dCompaction: Speeding up Compaction of the LSM-Tree via Delayed Compaction. *Journal of Computer Science and Technology*, 32(1):41–54, Jan 2017.

[98] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, and Jens Krueger. How to Juggle Columns: An Entropy-based Approach for Table Compression. In *Proceedings of the Fourteenth International Database Engineering; Applications Symposium*, IDEAS '10, pages 205–215, New York, NY, USA, 2010. ACM.

[99] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[100] Diego Perna and Andrea Tagarelli. An Evaluation of Learning-to-Rank Methods for Lurking Behavior Analysis. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, UMAP '17, pages 381–382, New York, NY, USA, 2017. ACM.

[101] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[102] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom Filters for Advanced SIMD Processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 6:1–6:6, New York, NY, USA, 2014. ACM.

[103] Orestis Polychroniou and Kenneth A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 9:1–9:6, New York, NY, USA, 2015. ACM.

[104] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 551–566, New York, NY, USA, 2018. Association for Computing Machinery.

[105] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[106] Gautam Ray, Jayant R. Haritsa, and S Seshadri. Database Compression: A Performance Enhancement Tool. 09 2004.

[107] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 882–890. AAAI Press, 2017.

[108] K. A. Ross. Efficient hash probes on modern processors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301, April 2007.

[109] Eyal Rozenberg and Peter Boncz. Faster Across the PCIe Bus: A GPU Library for Lightweight Decompression: Including Support for Patched Compression Schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, pages 8:1–8:5, New York, NY, USA, 2017. ACM.

[110] Russell Sears and Raghu Ramakrishnan. Blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 217–228, New York, NY, USA, 2012. Association for Computing Machinery.

[111] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 17–30, USA, 2013. USENIX Association.

[112] Julian Shun and Guy E. Blelloch. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 96–107, New York, NY, USA, 2014. Association for Computing Machinery.

[113] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. SIMD-based Decoding of Posting Lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.

[114] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[115] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang. Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79, 2017.

[116] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. *ACM Trans. Storage*, 14(2), April 2018.

[117] Risi Thonangi, Shivnath Babu, and Jun Yang. A practical concurrent index for solid-state drives. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 1332–1341, New York, NY, USA, 2012. Association for Computing Machinery.

[118] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. Recovering Semantics of Tables on the Web. *PVLDB*, 4(9):528–538, 2011.

[119] Henry S. Warren. Integer division by constants. In *Hackers delight*, chapter 10, pages 190–192. Addison-Wesley, 2 edition, 2013.

[120] "Zhang Weifeng, Hua Hu, and Haiyang" Hu. Neural ranking for automatic image annotation. *"Multimedia Tools and Applications"*, "77"("17"):"22385–22406", "Sep" "2018".

[121] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Trans. Database Syst.*, 15(2):208–229, jun 1990.

[122] 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. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009.

[123] Xiaokui Xiao, Yabo Xu, Lingkun Wu, and Wenqing Lin. Lsii: An indexing structure for exact real-time search on microblogs. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 482–493, USA, 2013. IEEE Computer Society.

[124] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online Deduplication for Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1355–1368, New York, NY, USA, 2017. ACM.

[125] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[126] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1601–1615, New York, NY, USA, 2020. Association for Computing Machinery.

[127] Meihui Zhang and Kaushik Chakrabarti. InfoGather+: Semantic Matching and Annotation of Numeric and Time-varying Attributes in Web Tables. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 145–156, New York, NY, USA, 2013. ACM.

[128] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun. Pipelined compaction for the lsm-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 777–786, 2014.

[129] P. Zhao, O. Wu, L. Guo, W. Hu, and J. Yang. Deep learning-based learning to rank with ties for image re-ranking. In *2016 IEEE International Conference on Digital Signal Processing (DSP)*, pages 452–456, Oct 2016.

[130] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[131] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[132] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

[133] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22Nd International Conference on*

Data Engineering, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

[134] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A Vectorized Analytical DBMS. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1349–1350, Washington, DC, USA, 2012. IEEE Computer Society.