OPTIMIZING LIGHTWEIGHT ENCODING IN COLUMNAR STORE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE DIVISION
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
HAO JIANG

CHICAGO, ILLINOIS
GRADUATION DATE
# TABLE OF CONTENTS

LIST OF FIGURES ................................................. v
LIST OF TABLES .................................................. vi
ACKNOWLEDGMENTS ............................................... vii
ABSTRACT ......................................................... viii

1 INTRODUCTION ................................................. 1
  1.1 Encoding Selection for Columnar Store ..................... 1
  1.2 Speed Up Data Filtering on Encoded Data with SIMD ...... 5

2 BACKGROUND ................................................... 8
  2.1 Lightweight Encoding ................................... 8
  2.2 Parquet File Structure .................................. 10
  2.3 SIMD Instructions ...................................... 12

3 RELATED WORKS ............................................... 14
  3.1 Encoding and Compression in Columnar Store ............ 14
  3.2 Hardware Acceleration in Database ....................... 16

4 DATA-DRIVEN ENCODING SELECTION ......................... 18
  4.1 Dataset Collection ...................................... 18
  4.2 Feature Engineering .................................... 21
  4.3 Experiments ............................................. 23
    4.3.1 Selecting Encoding with Best Compression .......... 24
    4.3.2 Encoding Selection Based on Partial Dataset ...... 26
    4.3.3 Encoding Impact on Query Performance ............. 27
    4.3.4 Encoding and Byte-Oriented Compression .......... 31

5 SUB-ATTRIBUTE EXTRACTION AND ENCODING ............... 37
  5.1 Algorithm .............................................. 37
  5.2 Experiment ............................................. 40

6 SPEED UP DATA FILTERING ON ENCODED DATA WITH SIMD ... 44
  6.1 System Architecture ................................... 44
    6.1.1 Operator for Data Filtering ....................... 45
  6.2 SIMD Algorithms ....................................... 46
    6.2.1 Data Filtering for Bit-Packed Encoded Integers ... 46
    6.2.2 Data Filtering for Run-Length Encoded Integers .... 51
    6.2.3 Fast Decoding and Filtering for Delta Encoded Data 52
  6.3 Experiments ............................................. 56
6.3.1 Microbenchmarks ................................................. 57
6.3.2 Boosting JVM-based Columnar Stores ....................... 60
6.3.3 Scalability ........................................................... 63

7 CONCLUSION ............................................................. 67

A THE CORRECTNESS OF DATA FILTERING ALGORITHM ON BIT-PACKED
DATA ................................................................. 69
A.1 Proof of Equality Test on bit-packed data ....................... 69
A.2 Proof of Range Test on bit-packed data ......................... 70

B IMPLEMENTING 512 BIT ADD/SUB OPERATIONS ................. 72

REFERENCES ............................................................. 76
# LIST OF FIGURES

1.1 Poor Encoding Selection Leads to Sub-Optimal Compression ............................................. 2
2.1 Parquet Columnar Store Format .......................................................................................... 11
2.2 How hadd works ................................................................................................................. 13
4.1 Distribution of Column Size .............................................................................................. 19
4.2 Accuracy and Impact of Encoding Selection ...................................................................... 25
4.3 Encoding Selection on Partial Dataset ............................................................................... 27
4.4 Scan Performance on Encoded Data .................................................................................. 29
4.5 Encoding Impact on TPC-H Queries ................................................................................. 30
4.6 Performance Comparison Between Encoding and Compression (compression ratio as encoded size/original size) ................................................................. 32
4.7 Performance of Compression over Encoded Columns (compression ratio as encoded size/original size) ................................................................. 34
5.1 Example of Columns containing Sub-Attributes ................................................................. 37
5.2 Comparing the aggregate file size ratios of sub-attributes with ideal encoding and ideal encoding with filtering candidate attributes based on a classifier. Left Y-Axis shows a histogram, and right Y-Axis and red line shows a CDF. 42
5.3 Sub-Attribute Extraction and Compression ...................................................................... 43
6.1 SBoost System Architecture ............................................................................................. 45
6.2 Operation Tree for equal operator .................................................................................... 52
6.3 Operation Tree for less operator ....................................................................................... 53
6.4 Use hadd to compute 32-bit Cumulative Sum .................................................................. 55
6.5 SBoost Performance on Bit-Packed Data ......................................................................... 57
6.6 SBoost Performance on Run-Length Encoded Data ............................................................ 61
6.7 SBoost Performance on Delta Encoded Data ..................................................................... 62
6.8 Accelerating Queries in Parquet ....................................................................................... 64
6.9 Scalability of Bit-packed filter .......................................................................................... 65
6.10 Scalability of Delta decode .............................................................................................. 66
B.1 Compute blend instruction from carry bits ......................................................................... 74
LIST OF TABLES

2.1 Popular Encodings Supported by Non-Commercial Columnar Systems . . . . . . 11
4.1 Distribution of Data Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.2 Encodings Supported by Apache Parquet . . . . . . . . . . . . . . . . . . . . . 20
4.3 Datasets Statistics By Category . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
ACKNOWLEDGMENTS

This work is supported by the CERES Center for Unstoppable Computation and gifts from FutureWei.
ABSTRACT

In columnar databases, data is generally stored in an encoded format to save storage space and reduce I/O. Columnar encoding is a family of encoding methods that reduce storage size of an attribute, while still enabling efficient in situ data processing. Popular encoding schemes include dictionary encoding, delta encoding, run-length encoding, and bit-packed encoding. In this thesis, we propose methods to optimize columnar encoding for both space and time efficiency.

The selection of right encoding for an attribute is critical for ensuring good compression, however prior work and open-source systems rely on static rules based global knowledge of the dataset or simplistic rules based on the data types. We evaluate the impact and selection of encoding by studying a popular open-source columnar storage framework, Parquet. We highlight how encoding implementation differences leads to challenges in selecting the ideal encoding, explore a data-driven method to select encoding schemes for a given dataset, and evaluate various encoding schemes on a large corpus of public datasets. We also examine decomposing attributes into sub-attributes to enable better compression. This evaluation highlights shortcomings with existing techniques and shows promising directions for efficient columnar storage systems.

In many columnar data store implementations, performing queries on encoded data requires the data to be first decoded to memory, which is time-consuming. We design several novel SIMD-based algorithms to speed up query execution on encoded data. Our algorithms use SIMD to vectorize the execution and skip unnecessary decoding for higher efficiency, achieving a throughput of filtering up to 18 billion numbers per second with single thread. We build SBoost, a columnar data store utilizing these algorithms to speed up filtering on encoded data, thus improving query efficiency. SBoost is written in Java and invokes the SIMD algorithms using JNI, making it readily available for Java-based query platforms, which are dominant in open-source data analytic systems. SBoost demonstrates great po-
potential in speeding up query efficiency in both disk-based analytic queries and in-memory queries by reducing query time by up to 90% compared to Apache Parquet.
CHAPTER 1
INTRODUCTION

Over the past decade, columnar databases have come to dominate the analytical market due to their ability to minimize read data, maximize cache-line efficiency, and provide effective compression. The physical compactness of data also enables faster sequential access and higher memory bandwidth utilization. These advantages lead to ‘orders of magnitude’ levels of improvement for scan intensive queries [25, 63]. As a result, academic research [50, 2, 27, 1], open source communities [8, 7], and large database vendors such as Microsoft SQLServer, IBM, and Oracle all are embracing this architecture.

Columnar stores allow efficient encoding techniques to be adopted. Abadi et al. show that in addition to space saving, executing queries on encoded data also exhibits great potential in improving query efficiency[2]. In practice, lightweight encoding algorithms, which trade compression ratio for much faster decompression operations, are 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. In this thesis, we describe our work of improving encoding efficiency in columnar store.

1.1 Encoding Selection for Columnar Store

To support efficient storage and query processing, many columnar databases support columnar encoding. Popular columnar encoding schemes include dictionary encoding, run-length encoding, delta encoding and bit-packed encoding. These encoding schemes differ from byte-oriented compressions schemes, such as Snappy [22] and GZip [21], in that custom database iterators can directly process encoded data[2, 1] without needing to decompress a segment of data first. While many database systems provide support for byte-oriented compression [23],
For optimal compression rates and efficient query performance, it is crucial to choose proper encoding for each column. For example, choosing dictionary encoding for a column of which its average value size is small and cardinality is large, is unlikely to exhibit either good compression (e.g. due to wasting storage for keys with no space saving for attribute size) or good query performance (e.g. loss of range predicates and value translation overhead).

Despite the prevalence of columnar systems and the importance of proper encoding selection, limited work exists on how to properly choose encoding for a given dataset. Furthermore, previous seminal work [2] on encoding selection relies on global knowledge of the dataset, such as whether it is sorted. This often requires multiple passes on the original dataset to generate the encodings, which is prohibitively time-consuming when dataset size increases. We refer to these rules for columnar encoding selection as Abadi throughout this paper [2].

As a consequence, open-source columnar systems (i.e. Parquet and Carbondata) choose to hard-code encoding selection based solely on the data type or leave the encoding se-
lection to the user. Some frameworks actually require code modifications to support user-driven encoding selection. Unfortunately, making such choices not only requires extensive knowledge on the database implementation, but also expensive analysis on target datasets to determine features on attributes, such as distribution, cardinality, and sortedness. The selection of columnar encoding and use of byte-oriented compression impacts three critical dimensions: the size of the encoded files, time to generate the encoded data, and overhead (or benefits) to operate on the encoded data. This often exceeds users’ capability and leads to sub-optimal decisions in practice. In Figure 1.1, we show that either Abadi’s method or Parquet’s encoding selection failed to achieve best compression on a substantial number of datasets, and may even generate encoded files that are larger than the original ones. Here we compare these methods using Parquet, with the optimal encoding as the empirical best encoding.

To address some of the aforementioned problems, we propose a data-driven method for encoding selection in columnar storages. We utilize machine learning techniques to learn the impact of encoding given a particular implementation and a large corpus of datasets, and results in a method that is capable of selecting most efficient encoding for a given dataset. This approach is beneficial in that it requires no prior knowledge of candidate encodings, domain knowledge input from user, or understanding details of the encoding implementation, which can have a significant impact on the encoding efficiency.

Due to its popularity, extensibility for encodings, and open-source nature, we build our experiment platform on Apache Parquet columnar format [8]. With experimental results on Apache Parquet, we demonstrate that our data-driven method is both accurate in selecting the columnar encoding with the best compression and is fast for selecting the encoding. On Apache Parquet, we have achieved over 96% accuracy in choosing the best encoding for string types and 87% for integer types. The time overhead of making such a choice is sub-second regardless of dataset size.
In addition to encoding attributes, many columnar systems allow users to apply byte-oriented compression (i.e. gzip) on encoded data to further reduce storage size. The granularity at which this compression occurs varies from an entire column to a page. Any use of byte-oriented compression results in a blocking decompression step before any applying any operators. In this paper we analyze the benefits and overhead of applying compression in the presence of intelligent encoding selection. We also present a framework to guide user to choose proper configuration.

In analyzing the impact of byte-oriented compression and columnar encoding, we found aggressive compressions schemes are still able to significantly compress many attributes, which implies the entropy of the encoded data is amenable to compression. Therefore, we investigate opportunities to further improve encoding efficiency to reduce storage size and develop an algorithm for extracting sub-attributes from string columns that allow different encodings to be applied to each sub-attribute. Our results show that enabling such a compression reduces the compression efficiency gap between byte-oriented compression and columnar encoding.

We believe that this detailed analysis of columnar encoding and byte-oriented compression for a popular open-source framework provides the following contributions:

- An evaluation of how prior research and open-source systems do not encode to minimize storage size.
- A detailed study on the impact of columnar encoding and byte-oriented compression on storage size, file generation time, and read time.
- A lightweight data driven encoding selection method to pick an ideal encoding with minimal overhead.
- An analysis of decomposing attributes into sub-attributes to close the gap between columnar encoding and byte-oriented compression.
1.2 Speed Up Data Filtering on Encoded Data with SIMD

Many previous researches focus on using new hardware features, such as single-instruction-multiple-data (SIMD) instructions, to improve query performance on encoded data. Willhalm et al. [54] demonstrates a new algorithm using 128 bit SIMD instructions to decode 4 bit-packed integers in parallel. Polychroniou et al. [42] propose using SIMD to speed up selection scan, sort, and join operations. Variations of encoding schemes further explore the potential of SIMD processors. BitWeaving [37] and BP-128 [34] are variations of bit-packed encoding. SIMD-PFOR [34] is a variation of patched encoding. These variations all exhibit significant better performance comparing to corresponding scalar version and demonstrate that using SIMD to speed up encoding/decoding operations in database systems has great potential.

However, most of these algorithms work only on customized variation of encoding schemes that either need extra space in the storage format or requires data to be re-organized in a special order, making them space-inefficient and incompatible with standard encoding specifications. For example, one of the variation formats BitWeaving proposes, BWH, requires a separator bit between entries and entries residing within 64-bit lanes that can lead to a space waste of up to 30%. Another variation, BWV packs data tightly, yet requires data to be stored vertically instead of horizontally, e.g, adjacent bits in same entry are separated into adjacent words. In addition to wasting space, converting existing datasets 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 filtering / decoding data stored in standard encoding formats, including bit-packed encoding, run-length encoding, and dictionary encoding. Our data filtering algorithms works directly on encoded data, efficiently skipping a decoding process, saving both CPU effort and memory space. Comparing to previous methods, our algorithms can process more numbers in parallel and
achieves a throughput of filtering up to 18 billion numbers per second with a single thread.

We implement these algorithms in SBoost, a columnar data store based on Apache Parquet’s storage format. SBoost is implemented in Java, and invokes SIMD algorithms through JNI to speed up data filtering on Parquet tables. SBoost works on widely used standard encoding schemes, making it readily available for existing data stores, and outperforms existing solutions by at least an order of magnitude. By improving query times for both on-disk and in-memory queries in Parquet, SBoost demonstrates great potential in speeding up query efficiency for Java-based query platforms.

The contributions include

- **Fast Table Filtering on Bit-packed encoded data.** During a selection scan/filtering, predicates such as equality and range search will be applied on data to obtain a comparison result. Many previous systems require 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 to skip decoding process, and having more numbers processed in parallel to improve throughput, thus achieving ultra-fast data filtering.

- **Fast Table Filtering for Run-length and Dictionary encoded data.** Using query rewriter to convert queries on the encoded data to predicates on underlying bit-packed data and utilizing our fast bit-packed scan algorithm, we propose fast SIMD-based table filtering algorithms for both run-length and dictionary encoded data.

- **Fast Decoding and Table Filtering 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 filtering on the decoded data.

- **Speeding up Java-based Query Platforms with SIMD + JNI.** We build SBoost
to demonstrate that our algorithms are able to speed up both OLAP and in-memory queries for Apache Parquet. It also provides JNI interfaces and can be easily migrated to other Java-based query platforms. Our experiments shows this architecture has potential to improve query efficiency for other Java-based query platforms such as Spark, ORC, and CarbonData.
CHAPTER 2
BACKGROUND

Here we review columnar encoding, differences in implementations, and discuss our platform's format in detail.

2.1 Lightweight Encoding

In this section, we briefly introduce common columnar encoding schemes.

**Bit-Packed Encoding:** Bit-Packed Encoding stores the number using as few bits as possible. Given a list of non-negative numbers \([a_0, a_1, \ldots, a_n]\), bit-packed encoding find a \(w\) satisfying \(a_i < 2^w\), and represents each number using \(w\)-bit loselessly. The bits are then concatenated in sequence as encoding output. Note that bit-packing requires knowledge of the largest observed max to generate the encoding. Our feature extraction process can estimate a max value, but if wrong it can require recoding the entire dataset. *Null suppression* [2] shares the same idea but uses two bits to indicate the byte length for the encoded values, thus values could be encoded by only using as many bytes necessary to represent the data.

**Delta Encoding:** Delta Encoding stores the delta between consecutive values, most commonly on numbers. Given a list of values \([a_0, a_1, a_2, \ldots, a_n]\), delta encoding encodes it as a list \(b_i\) with \(b_0 = a_0, b_1 = a_1 - a_0, b_2 = a_2 - a_1, \ldots, b_n = a_n - a_{n-1}\). The results can then be bit-packed. As the delta between numbers are generally smaller than the numbers themselves, bit packing the delta generally allows higher compression ratio than bit packing the original data. *FOR* and *PFOR* share similar idea, but store all values as offsets from a reference value rather than previous value, which is fixed or page level respectively.

**Run-length Encoding (RLE):** Run-length Encoding encodes a consecutive run of repeating numbers as a pair \((num, run-length)\). The list \([a_0, a_0, a_1, a_2, a_2, a_2, a_3, a_3, a_3, a_3]\) will be encoded as
\([a_0, 2, a_1, 1, a_2, 3, a_3, 4]\). The result may then bit packed. The combination of bit-packing and RLE is set by default in Parquet’s RLE implementation.

**Dictionary Encoding:** Dictionary Encoding uses a bijective mapping (a dictionary) to map input values of variable length to compact integer codes. The dictionary used in the encoding process is prefixed or attached to the encoded data. Dictionary allows conversion from data of arbitrary types to integer codes, further enabling more efficient encoding through hybrid schemes, such as bit packed or RLE. In some application contexts, several local dictionaries could be used to substitute one single global dictionary, which is actually the case in Parquet.

**Bit Vector Encoding:** Bit Vector Encoding stores values using bit vectors, each distinct value corresponds to one bit vector which shows the its distribution over all positions. It is useful where the data cardinality is very low. The list shown in the RLE example would be encoded as four bit vectors \(a_0 : [1100000000], a_1 : [0010000000], a_2 : [0001110000], a_3 : [00000001111]\).

**Hybrid Encoding:** To enable higher compression ratios, Parquet supports a set of hybrid encoding as well. In Parquet’s implementation, bit-packed encoding is applied to delta encoding by default, which is adapted from binary packing [34]. A combination encoding of bit-packing and RLE is supported to store repeated values more efficiently. After dictionary encoding, the values are stored as integers by using Parquet RLE encoding.

Table 2.1 shows the difference of mainstream encodings supported by state of the art column-stores and file formats. We merge some similar encodings together and omit several encodings which are rarely supported or used in specialized context. Both Parquet and C-Store support a broad varieties of encodings compared with their counterparts. However, Parquet file is organized for more Hadoop-like environments. Parquet does not need to periodical data moving from a write-store to a read-store as C-Store does, but the input can be streamed to a parquet writer directly, typical for distributed and append only environments. Parquet does not replicate column groups into projections with distinct sorting,
which provides a more succinct layout compare with C-Store, but lacks the ability to control sorting. As Parquet is designed to be stored on a distributed file system, Parquet files can be processed in parallel with row groups as the atomic processing unit for reading and writing. It easily supports new encoding schemes and can be used in a variety of engines like Hive, Impala, Pig, and Spark. For these reasons, combined with its popularity and open-source nature we focus on Parquet, but believe that it’s architecture is found in many modern frameworks (i.e. Dremel, Carbondata, and ORC).

2.2 Parquet File Structure

Parquet is an open-source column-oriented file format for distributed analytic frameworks, such as Spark and Impala [58, 32]. It provides efficient data compression and encoding schemes with the ability to handle complex nested data. In the Parquet file format, values from each column are logically organized to be adjacent and physically stored in contiguous memory locations for improved compression and query I/O. Along with benefits from column-oriented storage, Parquet provides extensibility of storing auxiliary structures (e.g. indices, statistics, dictionaries) in the columnar format to facilitate efficient read operations, and distributed write capabilities by storing metadata at the end of the file.

As shown in Figure 2.1, a Parquet file is made up by several row groups, which are indexed by block metadata saved in a file footer. A row group consists of several column chunks with metadata also in the file footer. In each column chunk, there are data pages and a dictionary page if dictionary encoding enabled. Columns are aligned in row group level, which means all data for a given row is organized in the same row group. In the file footer metadata is organized about column chunk metadata, zone maps, encoding, and compression information.
Figure 2.1: Parquet Columnar Store Format

<table>
<thead>
<tr>
<th></th>
<th>RLE</th>
<th>Dict</th>
<th>Delta/ FOR/PFOR</th>
<th>BitVector</th>
<th>BitPacked/ Null Suppression</th>
<th>Dict-RLE/BP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Store</td>
<td>✓</td>
<td>✓ (global)</td>
<td>✓ (prior)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Parquet</td>
<td>✓</td>
<td>✓ (local)</td>
<td>✓ (fixed)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Carbondata</td>
<td>✓</td>
<td>✓ (global)</td>
<td>✓ (fixed)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ORC</td>
<td>✓</td>
<td>✓ (local)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MonetDB</td>
<td></td>
<td>✓ (global)</td>
<td>✓ (fixed)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kudu</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Popular Encodings Supported by Non-Commercial Columnar Systems
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 [26].

- **horizontal add (hadd)** hadd instruction allows multiple adjacent integers (16 bit or 32 bit) in a SIMD word to be added simultaneously. Figure 2.2 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. $c = \text{permutex2var}(a, b, i)$ satisfies

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

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

- **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.
Our algorithms also require arithmetic operations on entire SIMD words, which lacks support in neither AVX2 nor AVX-512. We implement add and sub operations for 256/512-bit SIMD words. This is described in Appendix B.
CHAPTER 3
RELATED WORKS

While a large body of research exists on compression and columnar databases, we limit our focus here to columnar encoding, encoding selection, and querying over encoded data. A recent survey covers fundamentals of columnar database systems \[1\].

3.1 Encoding and Compression in Columnar Store

Data Store and Encoding As analytic database systems require intensive I/O operations, previous studies focus on storage size reduction and thus reduced I/O for scan intensive workloads \[28, 12, 45\]. These projects demonstrate that a large CPU overhead common with decompression can limit its application.

Comparing to traditional compression techniques, columnar encoding algorithms look for a trade-off between data size reduction and CPU overhead on decompression. Lemire and Boytsov \[34\] show that for certain data sets, encoding achieves comparable compression ratio with far lower CPU consumption compared to compression algorithms, such as GZip. Besides columnar data stores, lightweight encodings are also found applicable to traditional row-based data store. Xu et al.\[55\] propose a similarity-based deduplication approach to group records in row-store, and use delta encoding for compression.

Columnar databases, such as C-Store \[50\] and Monet DB \[25\], physically persist attributes consecutively on disk, allowing lightweight columnar encoding techniques, such as run-length and bit-packed, to be applied. Reasonable size reduction, significant low CPU overhead, and in-situ query execution make encoding algorithms more favorable than byte-oriented compression (i.e. GZip, Snappy) in columnar data stores \[2\]. Many database systems additionally utilize some form of byte-oriented compression that is applied at the page \[8\], record-group \[11\], or column level \[50\] – both with or without attribute encoding.
applied.

**Speeding up Queries on Encoded Data** Columnar encoding has an advantage over block-oriented compression algorithms that they allow information to be retrieved and executed on before decoding data, which allows more efficient query execution [2, 1]. Several novel algorithms [54, 44] allows direct table scans on encoded binary data, thus skipping entire decoding operations and reducing CPU overhead. Bian et al. [9] proposes a cost-model for evaluating I/O overhead in columnar stores, allowing data to be accessed more efficiently.

Using hardware to speed up the decoding and query processes also demonstrates significant benefits. Variations of encoding algorithms that are SIMD-optimized allow for more efficient encoding and decoding [15, 49, 37, 34]. Lang et al. [33] propose a new columnar storage format, allowing SIMD-optimized predicate evaluation. Researchers also pay attention to GPU and dedicated hardware. Rozenberg et al. [48] develops Giddy, a library for executing fast decoding algorithms using GPUs. Fang et al. introduce UDP [17], a co-processor for data extraction and transformation tasks that are common in columnar encoding and compression.

**Encoding Selection** Encoding selection assists database users and administrators in deciding the best encoding scheme for a given dataset. Most prior work [12, 61, 2] on encoding performance reports their evaluation based on the TPC-H [52] workload and dataset. As part of this work we demonstrate how prior methods work with diverse datasets that has different distribution characteristics.

Abadi et al. [2] in their paper on encoding and query execution, introduce a handcrafted 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. [34] 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 [8] uses dictionary encoding for all data types.

Our work of data-driven encoding selection makes choices based on statistical results from collection of datasets, and thus provides a more accurate and reliable result. This method can easily be extended to evaluate the performance of new encodings. In this sense, our work attempts to bridge between academic research and real-world applications.

### 3.2 Hardware Acceleration in Database

**Database Encoding and SIMD** Database systems involve 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 [28, 12, 45] explore the impact of compression on 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 achieve high compression ratios with relatively low CPU consumption. They also allow in-situ query execution without decoding the entire data block [2]. These advantages make them more favorable than generic compression algorithms, such as GZip and Snappy, in database systems.

As decoding processes generally involve independent simple operations on multiple data entries, SIMD seems like a perfect solution to the problem. Willhalm et al. [54] describe a SIMD-based algorithm for decoding and filtering tightly bit-packed data. While Willhalm’s algorithm uses one 32-bit lane to filter each entry, and can process at most 16 entries in parallel using AVX-512, our algorithm fits as many entries as possible into a 64-bit lane, and
can process up to 256 entries for entry size of 2, or 168 entries for entry size of 3 in parallel. In the experiments, our algorithm is able to achieve up to 12x throughput in filtering speed compared to Willhalm’s algorithm and allowing us to filter up to 18 billion values per second.

Others focus on designing encoding variations that work well with SIMD. Stepanov et al. [49] introduce a SIMD version of varint-G8IU [15]. Lemire et al. propose SIMD-FastPFOR [34], a SIMD variation of PFOR [62] that pads both binary-packed data and exception arrays to be aligned with SIMD word boundaries. Li et al. demonstrate BWH and BWV, variations of bit-packed encoding that supports SIMD based fast filtering and early-pruning [37].

**SIMD Acceleration in other Database Operations** SIMD has many advantages compared 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 inter-operability 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. [60] describe the general idea of using SIMD for various database operators including scan, aggregation, index scan and join. Chhugan et al. [13] use SIMD to implement a bitonic merge network for merge sort. Ross et al. [46] propose to speed up hash join by optimizing Cuckoo hashtable [16] with SIMD. Jha et al. [29] experimentally explores the hardware oblivious and hardware conscious joins on Xeon Phi platform with SIMD optimization. Other applications including vectorized bloom filter [43] and bitmap counting [39].
CHAPTER 4
DATA-DRIVEN ENCODING SELECTION

In this chapter, we detail our method of using a simple neural network to build a data-driven encoding selection (DDES) solution. We build a dataset collection framework to collect, parse, convert data to a columnar format, and extract features from these columns as input for DDES. We evaluate a variety of models to predict the accuracy of selecting columnar encoding that has optimal compression. While several models, including k-Nearest Neighbor (KNN, [4]) and decision tree all give high accuracy, we settle on a simple neural network for DDES as it leads to highest accuracy. Details of our model construction and training are in Section 4.3.1.

4.1 Dataset Collection

Our initial training set is derived from The datasets we use in this paper covers a wide variety of domains and application scenarios that have needs for storing large structured datasets, including open city data portals, scientific computation cluster logs, machine learning datasets, and data challenge competitions. Detailed description and links to download data files can be found in the project repo. Table 4.3 shows the statistical overview of datasets by their categories. These domains generates and store gigantic amount of data, facilitating many important applications. Studies on server log [18] has served as foundation of many research including data loading [3], query optimization [51] and data partitioning [41, 56]. Machine learning, especially recently surged deep learning relies heavily on enomorous amount of training data to function properly. With millions of active user, social network systems (SNS) such as Facebook, Yelp and Twitter can generate tons of data everyday. Government facilities generally have regulations requiring data records to be kept for a long period (3-7 years). Proper encodings help relieving storage bottleneck and thus serves these scenarios well.
Table 4.1: Distribution of Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Column Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>9435</td>
</tr>
<tr>
<td>INTEGER</td>
<td>5183</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>3218</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>922</td>
</tr>
<tr>
<td>LONG</td>
<td>482</td>
</tr>
<tr>
<td>FLOAT</td>
<td>18</td>
</tr>
</tbody>
</table>

Figure 4.1: Distribution of Column Size

Table 4.1 shows the distribution of columns by their types. We believe this is a good estimation of data type distributions for many real-world scenarios. String and Integer types dominate the dataset (over 76%). We also notice that columns of double type occupies a considerable portion (17%) in the dataset. Examining these attributes, we find that most of them belongs to GIS, machine learning training, and financial datasets. Parquet only supports Dictionary encoding for double attributes, and on 25% of double columns dictionary encoded file size is larger than original size. However, lossy encoding allows a much larger space of choices. Research [47, 36, 59] on application scenarios explore specific lossy compression for double data. We believe mixing lossy and lossless compression for double attributes is a promising future research direction of our work.

Figure 4.1 demonstrates the distribution of column size, and the red curve is the result of fitting data points to normal distribution. It can be noticed that 90% of the columns have between 10K and 1 million values and has a near-normal distribution. This discovery is also helpful for determining hyperparameter values while designing a data store.

To aid in data collection, we develop an automatic collection framework. The framework consists of a file reader, a feature extractor, and a data store. File reader uses file extensions to determine file format and invokes a corresponding parser. We currently support common file formats, including CSV, TSV, JSON, XLS and XLSX. The file reader splits a file into columns and infers each column type. The framework extracts features on the generated columns, which we detail in next section. We store the generated columns as separate files.
### Table 4.2: Encodings Supported by Apache Parquet

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Encoding Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Delta-Length Byte Array</td>
</tr>
<tr>
<td></td>
<td>Delta Byte Array</td>
</tr>
<tr>
<td></td>
<td>Dictionary</td>
</tr>
<tr>
<td>Integer</td>
<td>BitPacking</td>
</tr>
<tr>
<td></td>
<td>Delta-Encoding Binary Packing</td>
</tr>
<tr>
<td></td>
<td>Dictionary</td>
</tr>
<tr>
<td></td>
<td>Run-Length BitPacking Hybrid</td>
</tr>
<tr>
<td>Double</td>
<td>Dictionary</td>
</tr>
</tbody>
</table>

### Table 4.3: Datasets Statistics By Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Table Count</th>
<th>Column Count</th>
<th>Data Size(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Logs</td>
<td>166</td>
<td>3836</td>
<td>20.4</td>
</tr>
<tr>
<td>Government Records</td>
<td>256</td>
<td>5126</td>
<td>26.8</td>
</tr>
<tr>
<td>Machine Learning Datasets</td>
<td>111</td>
<td>3113</td>
<td>12.5</td>
</tr>
<tr>
<td>Social Network Datasets</td>
<td>98</td>
<td>1593</td>
<td>23.9</td>
</tr>
<tr>
<td>Financial Records</td>
<td>91</td>
<td>1954</td>
<td>16.8</td>
</tr>
<tr>
<td>Traffic Records</td>
<td>50</td>
<td>2826</td>
<td>22.8</td>
</tr>
<tr>
<td>GIS Data</td>
<td>16</td>
<td>382</td>
<td>5.2</td>
</tr>
<tr>
<td>Other</td>
<td>8</td>
<td>428</td>
<td>1.6</td>
</tr>
</tbody>
</table>

We use the encoding algorithms shipped with Apache Parquet [8]. Table 4.2 lists the encoding algorithms supported by Parquet for integer and string type. Other types are ignored as Parquet only has limited encoding support for them. E.g., double type can only be encoded in dictionary encoding so it does not quite make sense to build a selector for it. To determine the best encoding (or ideal encoding) for each data column, we apply all applicable encodings on every column, and compare the size of resulting disk file in Parquet’s format, which includes both data content and necessary metadata for decoding. The encoding algorithm generating minimal size of disk file is chosen as the “ground-truth” in the later training phase.

Overall, this dataset collection has a good coverage over common data application scenarios and has a balanced distribution among data types and data sizes. We believe that this...
dataset collection serves as a fair estimate of real-world data distributions and thus serves as a solid foundation of our conclusion in this paper.

4.2 Feature Engineering

The choice of features is crucial for the accuracy of a data-driven approach. Ideally, such a method should be able to select ideal encoding by only accessing first several blocks of the file, rather than the high overhead of scanning and parsing entire file. Therefore, these features should all be computable on a partial subset of the dataset. In this section we describe our features for encoding selection. We use $N$ for number of records in target column, and $[x_1, x_2, \ldots, x_n]$ to represent the values in a column.

**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 [53]. 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 following

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

**Sortedness:** The sortedness of a dataset evaluates how much “in order” a dataset is. Previous methods [2] use a boolean value to represent whether a dataset is sorted or not. However, we observed that compared to discrete variables, a continuous variable better captures the sortedness property of a dataset. We adopt three methods of evaluating the sortedness of a column, $f_s$, and include all of them in feature sets.
Kendall’s $\tau$ [30] and Spearman’s $\rho$ [14] are two classical measures of rank correlation. For our purpose of evaluating the sortedness of a given dataset, Kendall’s $\tau$ is computed as

$$\tau = 1 - \frac{2|\{(x_i, x_j)|i < j, x_i > x_j\}|}{n(n - 1)/2}$$

and Spearman’s $\rho$ 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 variation of Kendall’s $\tau$, called absolute Kendall’s $\tau$.

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

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

Computing these features on entire column have a time complexity of $O(n^2)$, which is prohibitively time-consuming. In practice 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 $\Theta\left(\frac{1}{W^2}\right)$, we can perform the computation in $O(n)$.

**Record Length:** We compute the length of each value in target column as the number of
characters in its plain string representation, and compute statistical information including mean, variance, max and min of the length.

**Entropy of Entire Column** We generate plain string representation of each value in target column, concatenate them into a single string and compute Shannon’s entropy.

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

where \( C = \{c_k|\exists i, c_k \in x_i\} \) is the collection of characters in the string, and \( p(c_j) = \frac{\sum_{i,k} I(x_i[k]=c_j)}{\sum_i |x_i|} \) is the frequency of character \( c_j \).

**Mean, variance, max, min of Per-Line Entropy:** We compute Shannon’s entropy in the same way as described above, but separately for each value in target column. This gives us \( n \) entropy values for a column containing \( n \) values. We then collect the statistical information, including mean, variance, max and min of the values.

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

\[
    f_{ne} = \frac{|\{i|x_i \text{ is not empty}\}|}{|N|}
\]

### 4.3 Experiments

The goal of our experimental evaluation is to understand the impact of an ideal encoding selection that empirically results in the highest compression ratio (e.g. smallest size). In particular we evaluate the size reduction benefits of ideal encoding, the accuracy of various approaches to select the ideal encoding, the impact of byte-oriented compression (i.e. gzip) with an ideal encoding, the impact of encoding and compression on reads, the overhead of generating and reading encoded and compressed formats.

All experiments are conducted on a workstation equipped with 4 Intel Core i7-5557U CPUS @ 3.10GHz, 16GB memory and SAS-1T disk. The system runs Linux Mint 17.2
Rafaela with kernel version 4.4.0-109-generic x86_64. We implement the selection and evaluation system with Java (Oracle 1.8.0_151) and Scala (2.12.4). Other software platform used in evaluation includes Apache Parquet version 1.9.0, Google Tensorflow 1.4.0, and Apache Hadoop 2.9.0. Source code is available for download at https://github.com/UCHI-DB/enc-selector.

### 4.3.1 Selecting Encoding with Best Compression

In this section, we evaluate the performance of DDES, our neural network based data-driven encoding selection method. We use a standard MLP neural network for the classification task. We construct a two-layer neural network with 400 neurons in the hidden layer, using Tanh as activation function, Sigmoid for output, and cross entropy as loss function. We train the network with Adam\[31\] for stochastic gradient descent using default hyper-parameters($\alpha = 0.9, \beta = 0.999$). The step size is 0.01, decay is 0.99. 70% of the dataset is used for training, 15% records for dev, and 15% records for testing. In each training process, we run at most 200 epochs, and early stop when the dev loss start increasing. 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 feature set.

We also compare the accuracy of our method to other candidate processes. 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, and empirically setup selection rules.

Apache Parquet has a built-in encoding selection mechanism which simply tries Dictionary encoding for all data types. Only when the attempt fails, it falls back to a default encoding for each supported data type. In practice, we notice that such failure is primarily caused by 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 decision
Figure 4.2: Accuracy and Impact of Encoding Selection

The experiment result is shown in Figure 4.2, where DDES stands for “Data-Driven Encoding Selector”. In fig. 4.2a, we show selection accuracy of different approaches, which is the percentage of samples the algorithm successfully choose encoding with minimal storage size after encoding. For string columns, DDES achieve 96% accuracy, a big improvement from Abadi’s decision tree with only 32% accuracy and Parquet’s encoding selection of 80%. For integer columns, DDES achieve 87% accuracy, also a substantial gain from Abadi’s 40% and Parquet’s 72%.

We also notice that although our neural network based DDES achieves best performance in both cases, KNN and decision tree also have a similar performance and exceed previous approaches. This fact from another perspective justify that we have chosen correct features to represent the characteristics of dataset.

In fig. 4.2b, we show how much storage reduction each algorithm can bring to the entire dataset. It can be seen that all machine learning algorithm works equally well, and can save 10~15% additional space on integer type. For string type column, there are also 5% improvement.

It is also crucial to make an encoding selection in a timely manner for certain environ-
ments. We study the time consumption of a 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 in these results. We test selection time when choosing first 1M bytes to generate features. The average time consumption is 436ms and in over 95% cases, the computation can be done within 1 second. We believe this time is negligible comparing to the time of loading and encoding data files, and support the feasibility of using our model in production environment.

We also see that when computing features on entire column, there is a strong correlation (0.9690) between column size and time consumption. This shows time consumption of encoding selection is linear to column size. If users want to gain higher accuracy in encoding selection at the cost of longer computation time, we can compute the linear regression coefficient and the expected time consumption. For reference, on our experiment platform this coefficient is 137.61, which means to compute features on a 100MB file, the time consumption is around 13.7 seconds.

4.3.2 Encoding Selection Based on Partial Dataset

We demonstrate that a neural network based data-driven encoding selection method outperform current state-of-art from both academic research and industry implementation. However, all the features we employ need to scan entire column, which is time-consuming. To mitigate this problem, we read only first $M$ bytes from the dataset and compute features based on those values. The computed features are then used to make decision as in original method. This effectively eliminates the correlation between dataset size and time needed for encoding selection, making it possible to make selection decision in constant time.

To empirically validate how much accuracy we can achieve with only partial knowledge of the dataset, we vary $M$ to be first 10K, 100K, and 1M bytes from each dataset, computing the features and make prediction based on them, and the results are shown in Figure 4.3.
Not surprisingly, the prediction accuracy decreases when a smaller $M$ is used. However, we still manage to achieve a reasonable accuracy. Our result shows that with $M = 100K$, we can get 83% accuracy on integer dataset and 92% accuracy on string dataset. When $M = 1M$, we have 85% accuracy on integer and 94% accuracy on string, which is still a better result than state-of-art.

4.3.3 Encoding Impact on Query Performance

Micro Benchmark

In this section, we evaluate how different encoding schemes affect time needed to access column data. Here we perform full-table scan on each column with all possible encodings for all columns in our dataset where we decode data but do not materialize it to memory.

First, we notice that regardless of encoding type and data type, scan time maintains a strong correlation with original column size. For each encoding of integer types, correlations between scanning time and original file size are always $\geq 0.98$. For string types, this correla-
tion is slightly lower, but the minimal value, which is for scan time on delta-binary-packing encoding, is above 0.85, which still implies a strong correlation. In contrast, scan time has a very low correlation with encoded file size. For example, scan times on integers with bit-packed encoding has a correlation of 0.9984 with original column size, but only 0.4694 with encoded column size. We observe a similar effect for other encodings and data types. Overall, this means time needed for scanning a encoded column is determined by the raw column size, but not related to the encoded file size.

The high correlation between scan time and original file size allows us to use a linear regression to compare query performance of different encodings. The result is shown in Figure 4.4 for both integer and string types. We draw fitting curves for each encoding type, where a larger slopes means slower scanning speed.

It can be noticed that for integer type columns, scan speed on different encodings only have slight difference. Bit-packed encoding has the best performance, but is only $\sim 5\%$ better than dictionary encoding/RLE encoding. For string type columns, the difference between encodings is more obvious. Dictionary encoding performs best and is around 20% faster than delta/delta-length encoding.

This micro benchmark experiment shows that for integers, encoding data always have positive effect to query performance, while differences between encodings is small. This allows users to always choose the encoding scheme having best compression ratio, without worrying about impact to query efficiency, which justify our work in encoding selection that use encoded file size as ground truth.

For strings, as performance difference between encodings become more obvious, we can no longer simply choose encoding schemes with best compression ratio, as this may lead to unacceptable query performance deterioration. In this case, we need more thorough analysis using the performance model proposed in Section 4.3.4. However, if storage size is not a concern, then one can always choose dictionary encoding for best query performance.
TPC-H Query Performance

In this section, we evaluate the impact of different encoding on query performance using TPC-H benchmark. Our evaluation involves two TPC-H queries, Q6 is a single table scan and Q14 is a join involving two tables. For each columns involved in queries (projection, selection or joins), we vary their encoding with three settings, a) plain encoding for all columns, b) encoding chosen by DDES and c) encoding chosen by Parquet. Columns not involved in queries are encoded with Parquet’s default encoding throughout the entire experiment. Each query is repeated on TPC-H datasets of scale 1 to 30. We implement a simple query framework on top of Parquet to execute these queries.

The relational algebra of Q6 is

\[ \pi_{\text{extend \_ price}, \text{discount}} (\sigma_{\text{quantity} < a \land \text{shipdate} \in (b,c) \land \text{extend \_ price} \in (d,e)} (\text{lineitem})) \]

This involves four columns: extend \_ price and discount are double columns, shipdate is a
Figure 4.5: Encoding Impact on TPC-H Queries

For this table, DDES chooses binary-packed encoding for quantity column, and dictionary for all other columns. Parquet chooses dictionary encoding for all four columns. These two settings have almost identical encoded file size (0.2% difference), and are 77% smaller than plain encoding. The time for executing query is shown is shown in Figure 4.5a. We notice that while DDES and Parquet settings benefit from great storage reduction, this does not comes at the cost of query time overhead. The difference between query times under three settings have a relatively small difference (less than 2%).

The relational algebra for Q14 is

$$\pi_{type, extend\_price, discount} (\sigma_{shipdate \in (a,b)} (part \bowtie_{partkey} lineitem))$$

where partkey is a integer column, part.type and lineitem.shipdate are string columns, lineitem.extend_price, and lineitem.discount are double columns.
We perform a hash join, building a hashtable on the part table, and use lineitem table for probing. Parquet again encodes all columns using dictionary encoding. DDES use delta encoding for part_key column in lineitem table, and binary-packed encoding for the key in part table. Files encoded by Parquet are 70% smaller than the plain setting, and DDES files are 5% smaller than Parquet’s encoding. The experiment result is shown in Figure 4.5b. Again it can be noticed that at all scales, the difference of running time between different settings is negligible despite the large difference in storage size. From these experiments, it suggests choosing an encoding with the best compression ratio has minimal negative impact on query performance. We leave a more thorough study of the impact of encoding to a wider variety of queries for future research.

4.3.4 Encoding and Byte-Oriented Compression

In this section we evaluate the efficiency and interaction of columnar encoding and byte-oriented compression, including GZip [21], LZO [40] and Google Snappy [22]. Specifically we would like to address for the following question: If we ideally choose encoding, how much does it help to further apply compression on encoded data?

Previous work [34] shows that for certain datasets with high cardinalities and low fluctuation, bit-packed encoding and delta encoding can have a better compression ratio than GZip and Snappy. However, it is not clear whether the conclusion still holds when being extended to a more general space of datasets and encodings. Our study makes effort to fill this gap.

As described in Section 2.2, Parquet splits a dataset into row groups. Each row group uses columnar storage and columns are broken into pages, where the encoding occurs and relevant metadata lives (i.e. dictionaries). Compression algorithms are then further applied independently on each encoded page. This means no cross-column shared data is observed by the compression algorithm and allows us to study the effect of compression at per-column
First, we study if we choose encoding scheme wisely, is it possible to achieve similar performance as state-of-art compression algorithms. We encode each columns with encoding scheme chosen by our encoding selector (the “DDES” entry in figures), then compress the same column with GZip, LZO and Snappy separately. In practice, we notice Snappy and LZO always have identical performance, therefore to make figures clear, we just show result from LZO.

We evaluate and report encoded/compressed column size, as well as time consumption for compression/decompression. A full-table scan is performed on all output files, to evaluate time required for decompressing or decoding. The results given in Figure 4.6, with showing file size(fig. 4.6a), compression time (fig. 4.6b), and decompression time (fig. 4.6c) for integer values. Figures 4.6d to 4.6f shows the same for string values respectively.
In Figure 4.6a we show a cumulative sum histogram to compare how much each algorithm can compress files. The x-axis shows regions of compression ratio (output file size vs. original size), and y-axis shows the percentage of samples falling in each ratio region. It can be noticed that our encoding schemes works almost as well as GZip on Integer columns. Both can compress over 50% columns to less than 1/4 of original size, and can compress almost all columns to at most 3/4 of original size. We can also see LZO/Snappy is inferior to the first two in almost all ratio regions, and even have around 20% columns enlarged after compression.

Figures 4.6b and 4.6c shows compression and decompression time for integer columns. One interested thing we notice during experiment is that in all cases, both compress time and decompress time is highly correlated to original file size (with > 0.9 correlation). We use linear regression to fit the points and show the curves in figures. Curves with smaller slope means faster execution time. It is not surprising to see that Encoding always run faster than compression algorithms, by 20 ∼ 30%. We also notice while LZO/Snappy claim themselves to be much faster than GZip, this is only true for compression. Upon decompression, LZO/Snappy is only slightly (> 5%) better than GZip.

In Figure 4.6d, we see compression algorithms all perform better than encoding, however with small advantage. While GZip is able to compress almost all columns to less than half size, encoding can also achieve that for over 85% of the columns.

When looking at Figures 4.6e and 4.6f, we realize GZip achieves such good result at cost of great CPU overhead. GZip consumes around 3x time at compression compared to the other two, and 2x at decompression compared to encoding.

Overall, we can see that encoding schemes can achieve similar size reduction as compression algorithms, at a less CPU overhead. Does this result mean we can get rid of compressions from data store systems? To answer this question, we conduct the next experiment to see whether compression algorithms can further reduce storage size on already well-encoded
Figure 4.7: Performance of Compression over Encoded Columns (compression ratio as encoded size/original size)

columns. We again encode each column using the best encoding from encoding selector, then apply different compressions on top of that. Output file size and time consumptions are recorded as above. The result is shown in Figure 4.7.

In Figure 4.7a, we again use a cumulative histogram to show how much compression algorithm is able to further reduce an already well-encoded column size. Surprisingly, GZip can further reduce at least 1/4 the size on 50% columns, and only enlarge file size in 10% cases. LZO/Snappy is able to reduce the size for half of the columns, but for the other half it enlarges the size. In Figures 4.7b and 4.7c, we show time consumption comparison between encoding+compresssion vs. just encoding. Interestingly, we can see that applying compression on encoded column is more efficient than that on raw column, and has a comparable performance to the just encoding case. Similar cases happen to strings as can be seen in Figures 4.7d to 4.7f. GZip/LZO can further reduce encoded column size with a comparable
performance to encoding.

We propose a hypothesis to explain this situation. After encoding, columnar data becomes more organized and physically adjacent, and allows compression algorithms to work more efficiently. For example, dictionary encoding on a string column will collect all string data together into dictionary page, allowing a compression algorithm to easily observe compressible data structures both in the strings and the integer values for the keys. Additionally, encoded data reduces size and allows compression to operate on more data in a limited window. We leave the proof of this hypothesis to future work.

Based on these experimental result, using intelligent encoding and GZip compression together provides a good balance between size reduction and execution time, compared with relying on compression alone, which often is the case for column family systems [11]. However, this combination can only guarantee best compression ratio, as it is determined by dataset and algorithm, and is independent to hardware platform. It does not guarantee acceptable generation time, as time consumption can vary between hardware platforms. Thus instead of proposing a simple guideline, we propose a framework to determine best configuration for a given platform.

For any configuration $C$, encoding time $t_e^{(C)}$ and query time $t_q^{(C)}$ on a encoded/compressed column can be written as linear functions of original column size $s$. We have shown that the times are highly correlated with $s$ above.

$$
t_e^{(C)} = a_e^{(C)} \cdot s + b_e^{(C)}
$$

$$
t_q^{(C)} = a_q^{(C)} \cdot s + b_q^{(C)}
$$

The actual number of coefficients $a_e^{(C)}, a_q^{(C)}$, and bias $b_e^{(C)}, b_q^{(C)}$ are platform-dependent and can be obtained by running a small number of calibrations on target platform. We assume the percentage of read operation in expected workload is $r$. 

...
To find a configuration that minimizes storage size and access performance, we simply propose encoding and GZip.

To find a configuration that minimize average access time is thus equivalent to minimize 
\[
\text{Cost}_C = r \cdot t_q^{(C)} + (1 - r) \cdot t_e^{(C)}
\]
in this model. As both \( t_q^{(C)} \) and \( t_e^{(C)} \) are linear functions of \( s \), the cost is also a linear function of \( s \). And the configuration with best average latency is just \( \arg\min_C \text{Cost}_C \), which can be easily obtained by simply iterating all possible \( C \).

To find a configuration that cover both needs (e.g. get best size reduction while ensuring average access time is no more than a defined threshold), we can order configurations by their size reduction in descending order, and use the formula above to verify whether the configuration can meet the constraints.
CHAPTER 5

SUB-ATTRIBUTE EXTRACTION AND ENCODING

In practice, we often notice string columns that can be described by a common pattern. Figure 5.1 shows excerpts from 2 columns. Machine Partition attributes have high cardinalities, but when described as combinations of columns of smaller numbers, the cardinality drops drastically, allowing either bit-packed encoding or dictionary encoding to work well on them. Geom column contains long values of 50 characters. However, all the records have an identical 17 character header, a common “52C0” in the middle, and a common tail “40”. Observing this leaves us only 27 characters to encode, which efficiently cut half of redundant data. We call these columns “extractable” as they follow a general pattern and can be split into child columns.

5.1 Algorithm

We define a columns to be extractable if a common pattern can be observed, and values in such columns can be split into child columns that we refer to as sub-attributes of the column. Splitting a single string attribute into children attributes allows each child to be encoded independently, and can result in a better compression ratio. Additionally, generating a pattern that summarizes information from a column also allows query optimizers to efficiently filter and skip queries not fit for the column, just as zone maps for integer columns (i.e. (a) Machine Partition (b) The Geom

Figure 5.1: Example of Columns containing Sub-Attributes
Algorithm 1 Sub-Attribute Extraction

```java
function EXTRACT(column, n, p)
    records = column.getLines().take(n).filter(rand() < p)
    pattern = new Union(records.map(parseToken))
    while true do
        for rule in Rules do
            if rule.rewrite(pattern) then
                continue;
            end if
        end for
        break;
    end while
    regex = GENREGEX(pattern)
    subColGroups = TOPLEVELGROUP(regex)
    subColumns = Array(Columns, subColGroups.length)
    unmatchColumn = new Column()
    for record in column.getLines() do
        groups = regex.match(record)
        if null == groups then
            unmatchColumn.write(record)
        else
            for i in subColGroups do
                subColumns(i).write(group(i))
            end for
        end if
    end for
end function
```

by checking a pattern for Gnom, we know a query “Gnom = 42422A” does not match anything and can be skipped without any data access). In this section, we introduce our algorithm of decomposing an string attribute into sub-attributes. We then independently predict and evaluate optimal encodings on the new sub-attributes. In Section 5.2 we evaluate the compression efficiency for this approach.

Previous work on pattern extraction primarily focuses on ad-hoc unstructured data, such as inferring a schema from text or logs [18, 20]. Our algorithm follows a similar and makes optimizations for columnar data. Algorithm 1 shows our algorithm to extract sub-attributes from a given column.

The algorithm randomly samples a small number of values from the head of the column, and parses each value into three types of tokens: word, num, and symbol. It then tries to execute a series of rules. Each rule scans the current pattern and tries to make changes on
it (e.g. a rewrite). This process repeats until no rule can make any change to the pattern. The algorithm then uses the generated pattern to create a regular expression, which is then applied to each values for extracting sub-attributes.

The basic idea of algorithm is divide-and-conquer. We first look for common structures (e.g. symbols, words, etc.) in values from a target column (i.e. “-” (dash) in fig. 5.1a, and “52C0” in fig. 5.1b). Once found, such common structures will be used to divide records into sub-groups. We then treat each sub-group as a new column and further look for patterns in them.

Our algorithm guarantees that generated pattern match all records in the samples, yet it is possible that some unseen records cannot be matched. We put these records into a separate column called “unmatched”, together with its position in the original column, and we resample the column if the “unmatch rate” (e.g. number of records in unmatched column vs. total number of records) is too high.

We use a plug-in approach to manage rules, allowing new rules to be added easily. Currently the following rules are adopted.

**CommonSymbolRule**: looks for common symbols from the sequences and use them to divide records into sub-groups. If no common symbol is shared by all lines, the algorithm falls back to find major symbols that appear in most of the sequences, e.g. a symbol that appears in 70% (configurable) of all samples.

**SameLengthRecordRule**: deals with text sequences with exactly the same length, as seen in Figure 5.1b. Characters at same index from all sequences are scanned to decide the proper type (pure number, pure letter, or mixture of both) at this index.

**CommonSeqRule**: works similarly to **CommonSymbolRule**, but looks at all types of tokens. Tokens with the same types (word/number) will be considered equal.

**MatchAnyWordRule**: replaces exact matches with fuzzy matches, e.g., ”MIR” will be replaced by \w{3} This allows us to generate a more universal pattern, matching not only the samples
we are working on, but also the records we have not seen.

**FlatStructureRule**: removes unnecessary nested structures from generated patterns. E.g., Unions of “\w+” and a word token can be replaced by “\w+”.

We use fig. 5.1a to quickly show how the algorithm works. First, **CommonSymbolRule** finds that all records contain three dashes, and use them to divide records into four sub-groups. The first group contains a single word ”MIR”, the second and third group are both unions of of letter/number mixture, and the last group is a union of 4 distinct numbers. **SameLengthRecordRule** then finds group 2 and group 3 both have same length. By checking characters at same indices, it finds out the letters in these groups are actually hexadecimal numbers, and these two groups are rewritten as union of numbers. Finally, **UseAnyWordRule** rewrite group 1 from “MIR” to \w\{3\}, group 2 and 3 to [A-Fa-f0-9]\{5\}, and group 4 to \d+. This leaves us a pattern

\[(\w\{3\})-([A-Fa-f0-9]\{5\})-([A-Fa-f0-9]\{5\})-(\d+)\]

### 5.2 Experiment

In this section, we show our experimental results of mining patterns from columns and using these pattern to extract sub-attributes. Our results demonstrate that this approach indeed enables more efficient encoding on a substantial number of columns. We also propose a simple yet effective classifier to predict whether encoding sub-attributes of a column separately will get better result than directly encoding the column.

We apply sub-attribute extraction to our collection of 9435 string columns, and ignore those either contain only one sub-attribute or too many (currently we set the threshold to be 20) sub-attributes. For each column, we extract first 5000 records and sample 10% of them, leaving only ~500 records for pattern extraction. This is to make sure we cover enough samples from the file while not affecting performance due to IO overhead. The results shows 4596 (~50%) columns have a valid pattern and we are able to maintain the unmatched value
rate to be less than 5%.

We further apply encoding selection algorithm to the children columns generated from sub-attributes and encode them individually. We then compare the aggregate size of these child columns (including the unmatched values) with the size of original column, both as plain and encoded. Figure 5.2a shows the size ratio between encoded sub-columns compared to encoded original column (using the ideal encoding), both as a histogram (left Y-axis) and CDF (right Y-axis). Here a smaller X-axis value means the sub-attributes extraction can result in a better size reduction than original single column. Not surprisingly, we notice that most of the columns encode well after being decomposed into sub-attributes. 45% of the columns are able to be further compressed even when compared to the best encoding on the original column, which is a substantial improvement.

However, we notice the existence of considerable amount of outliers from Figure 5.2a. For around 50% columns, there is a size increase after decomposition, and in the worst case, the decomposed result can be more than 8x larger than origin. Examining these outliers shows majority have well-defined patterns, but very low cardinalities. Consider an extreme example that a column only contains duplicate of two distinct records. Dictionary encoding can handle this case well by translating the data into a stream of integers (0 or 1 in this case), and hybrid the dictionary with either bit-packed or run-length encoding. If we split the data into $n$ child columns and encode each one independently. For each child column, we will generate a separate dictionary, and exactly the same integer stream as that for the original column. The sum size of encoded sub-columns is thus roughly $O(n)$ times of the size of encoded origin. The more columns are extracted, the more space is wasted.

With this observation, we build a KNN based binary classifier to determine whether decomposing can improve compression ratios. We use five features from Section 4.2, namely Distinct Ratio, Length mean, Length Variance, Entropy Mean, and Entropy Variance. Experimental evaluation shows that this simple classifier is able to achieve 91.9% accuracy
on our dataset. We demonstrate in Figure 5.2b that applying the classifier prior to decomposing, we successfully eliminate the long tail seen in previous figures and still maintain decent overall compression ratio.

Sub-attribute Extraction and Compression

In this section, we show that decomposing a column has a similar effect as compressing a column. We have shown in previous section that even after being encoded, some columns can further gain size reduction by applying compression. As sub-attribute extraction has similar effect, one interesting question arise, If we decompose a column and encode children columns, is compression still helpful?

To answer this question, we decompose the string columns, encode then compress the children columns, and compare the benefit brought by compression before and after decomposing. The result is shown in Figure 5.3. X-axis represents ratio between file size after compression over that before compression. Higher ratio means less compression benefit. GZip-Origin and LZO-Origin shows the ratio one get by compressing the encoded original
column, and GZip-Sub/LZO-Sub shows the ratio one can obtain by applying compression to the encoded children columns.

We notice that after decomposing, compression over encoded data no longer helps that much. While GZip can compress 80% of encoded columns to at most half of encoded size, and 95% to at most 3/4, after decomposing, it can only compress 2% and 18% columns to the same ratio. What’s more, LZO can compress over 80% encoded columns to at most 3/4 of encoded size, but only less than 5% after decomposing. There are even 20% columns having their size enlarged after LZO. This shows decomposing + Encoding Selection can be a efficient replacement for classical compression algorithms.

Figure 5.3: Sub-Attribute Extraction and Compression
CHAPTER 6

SPEED UP DATA FILTERING ON ENCODED DATA WITH SIMD

We build SBoost, a columnar data store supporting SIMD-based fast table scan based on Apache Parquet[8], a prevalent open source columnar format. We design SBoost to be system independent, allowing it to be easily migrated to other columnar stores.

6.1 System Architecture

Figure 6.1 describes SBoost’s system architecture. A Parquet file is comprised of multiple column chunks, each consisting of fixed size pages, which are binary data buffers storing encoded column data. When filtering data /decode data from a column, SBoost locates corresponding data pages from Parquet file, maps each page to off-heap memory, and invokes corresponding SIMD algorithms, which are implemented in C++, through JNI to process all data items in that page. Result is then passed back to JVM for further processing. This design avoids data movement between JVM and native memory, also reducing the number of JNI invocations, which has non-negligible cost.

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

For columns that appear only in a select but not in a project, SBoost applies filter directly on encoded data buffer to generate the bitmap, which can be further used to filter other columns. Most open-source systems decode data before they can be fed to a predicate, which incurs both unnecessary CPU and memory overhead. 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 \texttt{decode} on them. SBoost designs novel algorithms utilizing SIMD parallelization to speed up decoding process.

For columns involved in both select and projection, SBoost first uses \texttt{filter} to generates bitmap on the column, and uses the bitmap result to efficiently perform data skipping, saving time for decoding operations on unmatched data.

\subsection*{6.1.1 Operator for Data Filtering}

SBoost supports common predicates, including equal, not equal, greater than, less than, and their logical combinations in data filtering. We implement these predicates using two operators: \texttt{equal}, which tests whether the target is equal to a given value \( a \), and \texttt{less}, which tests whether the target is less than a given upper-bound \( a \). These operators take as input the encoded data and output a bitmap.

It is easy to see that all predicates and their combinations can be implemented using these two operators with simple logical operations. For example, \texttt{less-equal}(x, a) = x \leq
\( a \) can be obtained by \( \text{or}(\text{less}(x, a), \text{equal}(x, a)) \), and \( \text{range}(x, a, b) = a \leq x < b \) can be obtained by \( \text{xor}(\text{less}(x, a), \text{less}(x, b)) \), with the presumption that \( a \leq b \). When introducing our implementation of \text{filter}, we will focus on describing how we implement \text{equal} and \text{less} operators.

### 6.2 SIMD Algorithms

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 when needed.

#### 6.2.1 Data Filtering for Bit-Packed Encoded Integers

In this section, we introduce our algorithm using AVX-512 for \text{filter} on bit-packed encoded integers. It also serves as the foundation of subsequent algorithms.

**Preprocessing** The first step of our algorithm is loading encoded data in a 512-bit SIMD word, and align them to 64-bit lanes. We load 4 128-bit SIMD words separately, combining them as one 512-bit word, and use _mm512_shuffle_epi8_ to do 128-bit lane shuffling, sending bytes belonging to each entry into corresponding 64-bit lanes. We then use _mm512_srlv_epi64_ to shift data to be aligned to lane boundary.

The purpose of this operation is to get data ready for the arithmetic operation we perform in the next step. Intel’s SIMD instruction set only provides arithmetic instructions within 64-bit lane. While previous methods such as BitWeaving handle the problem by aligning data to 64-bit lanes when storing data, we perform such alignment on the fly. This saves both storage space and data transformation cost. Experiment shows executing the alignment
operation at runtime introducing negligible performance impact.

In addition, we also study an alternative of directly performing 512-bit arithmetic operations, eliminating the need of performing data alignment. This is detailed in Section 6.3.

**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 $a$.

We first present the following theorem

**Theorem 1.** Let $x$ and $a$ be two unsigned integers of $n$-bits length. We denote the most significant bit of $x$ by $x_{msb}$, and the remaining bits by $x_{rb}$. Let $m = 1 \ll (n - 1)$, $d = x \oplus a$, and

$$r = d \mid ((d \& \sim m) + \sim m)$$

We have

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

The proof can be found in Appendix A.

Following the theorem, let $M$ be the most 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 to $a$, 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 the equality test result in the MSB of each entry.

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

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 $X = 101011, A = 011011$. 


The MSB mask $M = 100100$. Applying the computations above, we obtain $R = 101000$. 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:

- $d_{msb} = 1$
- $d_{rb} \neq 0 \iff (d_{rb} + \sim m)$ generates a carry to MSB
  \[ \iff (d_{rb} + \sim m)_{msb} = 1 \]
  \[ \iff ((d \& \sim m) + \sim m)_{msb} = 1 \]

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

\[ x = a \iff d = 0 \iff r_{msb} = 0 \]

**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 present the following theorem.

**Theorem 2.** Let $x$ and $a$ be two unsigned integers of $n$-bits length.

\[ c = (x \mid m) - (a \& \sim m) \]
\[ t = (\sim a \& (x \mid c)) \mid (x \& c) \]

We have

\[ x < a \iff t_{msb} = 0 \]
The proof can be found in Appendix A.

Following the theorem, we construct \( M \) and \( A \) in the same way as described above, and compute
\[
U = (X \mid M) - (A \& \sim M) \\
R = (\sim A \& (X \mid U)) \mid (X \& U)
\]
then return \( R \) as a sparse bitmap satisfying
\[
X_i < a \iff (R_i)_{msb} = 0
\]
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,
\[
x_{msb} = 0 \text{ and } a_{msb} = 1 \iff (a \& \sim x)_{msb} = 1
\]
In the second case, let \( u = (x \mid m) - (a \& \sim m) \)
\[
x_{msb} = a_{msb} \iff [\sim(x \oplus a)]_{msb} = 1
\]
\( x_{rb} - a_{rb} \) generates a carry
\[
\iff (x \& \sim m) - (a \& \sim m) \text{ generate a carry} \\
\iff [(m + x \& \sim m) - (a \& \sim m)]_{msb} = 0
\]
\[
\iff [(x \mid m) - (a \& \sim m)]_{msb} = 0 \\
\iff u_{msb} = 0
\]
Combining the equations above we have

\[ x < a \iff (a \& \sim x) \mid (\sim(a \oplus x) \& \sim u) \] 
\text{Equation (A.1) \quad Equation (A.2) \quad Equation (A.3)}

\(msb = 1\)

Using boolean algebra to simplify the formula, we have

\[(a \& \sim x) \mid (\sim(a \oplus x) \& \sim u) = \sim(a \& (x \mid c)) \mid (x \& c) = \sim r\]

This shows:

\[x < a \iff r_{msb} = 0\]

Our algorithm exhibits several advantages comparing to previous methods. SIMD-Scan \cite{54} moves each data entry into a separate 32-bit lane and makes comparison, allowing it to process at most 16 entries in parallel with AVX-512. We perform the comparison \textit{in situ}, avoiding unnecessary data movement, and process up to 256 entries in parallel. BitWeaving \cite{37} requires one bit to be preserved between data entries, and data be aligned to 64-bit lanes. We allows data to be tightly packed when stored, saving up to 30% storage space, and process up to 50% more data in parallel.

**Dealing with cross-boundary entries** For entries crossing SIMD word boundary, we use unaligned load instruction to load the next SIMD word including that entry. Previous research \cite{54} suggests that unaligned load/store leads to negligible performance penalties on recent Intel CPUs, and our experiments also justify this conclusion.

On platforms where unaligned load/store may lead to unacceptable performance penalties, we propose an alternative solution that simply extracts 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.

6.2.2 Data Filtering for Run-Length Encoded Integers

As is described in Section 2.1, run-length encoded data comprises of consecutive number pairs (val, run-length). These pairs are often then tightly bit packed. While the approach described in this section targets bit-packed integers, the approach can be generalized to run-length encoding of other fixed-size attributes.

We utilize the bit-packed filter algorithm described in Section 6.2.1 to generate this run-length bitmap. The basic idea is to execute predicates on val fields, while leaving run-length fields unchanged. This generates a run-length encoded bitmap. For example, when executing predicate $x < 200$ on a run-length encoded data sequence \{105, 2, 339, 4, 242, 1, 132, 8\}, the output is \{1, 2, 0, 4, 0, 1, 1, 8\}. This kind of bitmap had been widely adopted in previous works [19, 24, 57, 35].

We show that by setting bits corresponding to run-length fields to 0 in all input parameters except for $X$ used in Equation (6.1) and Equation (6.3), the run-length fields from $X$ will be preserved during the computation of bit-packed filter algorithm. In Figure 6.2, we draw the operation tree for Equation (6.1). The numbers above each nodes shows how the bits from input change after each operation. We can see that if all parameters except for $X$ (the gray blocks in figure) have their run-length fields set to 0, the run-length values from will be preserved. We perform the same check for less operator, as is shown in Figure 6.3 and get the same conclusion. This shows that by leaving the run-length fields as 0 in all parameters except for $X$ used in Equation (6.1) and Equation (6.3), we can get a run-length bitmap generated by applying the bit-packed filter algorithm.

Some complex operators may need additional processing, though. For example, range operator can be obtained by $\text{range}(x, a, b) = \text{less}(x, a) \oplus \text{less}(x, b)$. Per our analysis above,
\[(x \oplus a) \& \sim m + \sim m \mid (x \oplus a)\]

Figure 6.2: Operation Tree for \texttt{equal} operator

\texttt{less}(x, i) will preserve the run-length fields in input. Thus both operands of \texttt{xor} will have the same value in their run-length fields, and leads to 0 after the operation. To solve such problem, we simply rewrite \texttt{range}(x, a, b) = \texttt{less}(x, a) \oplus (\texttt{less}(x, b) \& \underbrace{11\ldots\ldots1}_{value~fields} \underbrace{00\ldots0}_{run-length~fields}).

That is, adding a mask that erases run-length fields from the right operand. This allows run-length fields data to be preserved during \texttt{range} operator. Similar technique can be applied to other operators.

### 6.2.3 Fast Decoding and Filtering for Delta Encoded Data

In this section, we introduce our vectorized algorithm for decoding delta encoded integer and float data utilizing AVX2’s \texttt{hadd} instruction.

As is described in Section 2.1, delta encoding stores delta between consecutive numbers in a tightly bit-packed format. We first use the same algorithm as is described in the preprocessing step of Section 6.2.1 to unpack the bit-packed numbers into 16-bit or 32-bit lanes, depending on the size of original data.

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 introduce a \texttt{cumsum} function that computes the cumulative sum of each entry in a SIMD register. That is,
given SIMD word \( B = [B_0, B_1, \ldots, B_n] \), \( A = \text{cumsum}(B) \) computes \( A = [A_0, A_1, \ldots, A_n] \) where \( A_i = \sum_{k=0}^{i} B_k \). The \text{cumsum} function for 256 bit SIMD word and 16/32 bit integer is demonstrated in Algorithm 2.

Figure 6.4 illustrates how the 32-bit algorithm works, where we use \( b_{ij} \) to denote \( \sum_{k=i}^{j} b_k \). 16-bit \text{cumsum} works in a similar manner and we skip the description here for succinctness.

Line 7 uses \text{permute} instruction to shift the input to left by 32 bits, shifting in 0. Line 8 uses \text{hadd} on the original input \( b \) and the shifted input \( bp \) to obtain sum of adjacent number pairs. Line 9 reorders the result using \text{permute} instruction, and line 10 uses \text{hadd} one more time to obtain partial sum of at most 4 consecutive numbers. Line 11 shifts the result of line 10 to the left by 128 bits, shifting in 0. Line 12 performs a 32-bit \text{add} to obtain cumulative sum for each index, and line 13 reorders the entry to correct sequence.

With the \text{unpack} and \text{cumsum} operations described above, it is now straight-forward to implement \text{decode} and \text{filter} for delta encoded data, which is shown in Algorithm 3 and Algorithm 4. We describe the 32 bit version here, and the 16 bit version can be implemented in a similar manner.

The variable \text{latest} in Algorithm 3 tracks the latest number we have computed so far,
Algorithm 2 Vectorized Cumulative Sum with 256 bit SIMD and 16/32 bit Integer

1: const ZERO = _mm256_set1_epi64(0);
2: const IDX = _mm256_setr_epi32(8,0,1,2,3,4,5,6);
3: const IDX2 = _mm256_setr_epi32(0,8,2,8,1,4,3,6);
4: const IDX3 = _mm256_setr_epi32(8,8,8,8,0,1,2,3);
5: const INV = _mm256_setr_epi32(3, 2, 1, 0, 7, 6, 5, 4);
6: function cumsum32(b)
7:   bp = _mm256_permutex2var_epi32(b, IDX, ZERO);
8:   s1 = _mm256_hadd_epi32(b, bp);
9:   s2 = _mm256_permutex2var_epi32(s1, IDX2, ZERO);
10:  s3 = _mm256_hadd_epi32(s1, s2);
11:  s4 = _mm256_permute2x128_si256(s3, IDX3, ZERO);
12:  result = _mm256_add_epi32(s3, s4);
13: return _mm256_permutevar8x32_epi32(result, INV);
14: end function
15: const SHIFT16 = _mm256_set1_epi64x(16);
16: const MASK16 = _mm256_set1_epi32(0xffff);
17: const INV16 = _mm256_setr_epi16(0xF0E, 0xD0C, 0xB0A, 0x908, 0x706, 0x504, 0x302, 0x100, 0xF0E, 0xD0C, 0xB0A, 0x908, 0x706, 0x504, 0x302, 0x100);
18: function cumsum16(b)
19:   bp = _mm256 bslli_epi128(current, 2);
20:   s1 = _mm256_hadd_epi16(current, bp);
21:   s2 = _mm256_sllv_epi64(s1, SHIFT16);
22:   s3 = _mm256_hadd_epi16(s1, s2);
23:   s4 = _mm256_and_si256(s3, MASK16);
24:   result = _mm256_hadd_epi16(s3, s4);
25: return _mm256_shuffle_epi8(result, INV16);
26: end function
and is initialized to 0. Line 4 unpacks the bit-packed entry into SIMD words, and line 5 computes the cumulative sum on it. Line 6 adds \texttt{latest} to the cumulative result, obtaining the decoded value, and line 7 updates \texttt{latest} with the last entry.

**Algorithm 3 decode** for Delta Encoded 32 bit Integer

1: \textbf{function} Decode(stream)  
2: \hspace{1em} latest = 0;  
3: \hspace{1em} \textbf{while} stream.hasNext \hspace{1em} \textbf{do}  
4: \hspace{2em} word = UNPACK(stream.next);  
5: \hspace{2em} cumsum = CUMSUM(word);  
6: \hspace{2em} decoded = _mm256_add_epi32(cumsum, latest);  
7: \hspace{2em} latest = _mm256_extract_epi32(decoded, 7);  
8: \hspace{1em} \textbf{end while}  
9: \hspace{1em} \textbf{end function}  

\texttt{filter} uses the output from \texttt{decode}, and utilize SIMD comparison operations to execute predicate on decoded entries. The result is a dense bitmap and can efficiently be used in future operations.

To the best of our knowledge, no previous vectorized algorithm has been proposed for standard delta encoding. Lemire et al. [34] propose a vectorized variation of delta encoding...
Algorithm 4 filter for Delta Encoded 32 bit Integer

1: function filter(stream, predicate)
2:     latest = 0;
3:   while stream.hasNext do
4:     decoded = DECODE(stream.next);
5:     if predicate.op == EQUAL then
6:         scanRes = _mm256_cmp_epi32_mask(decoded, predicate.val, _MM_CMPINT_EQ);
7:     elseif predicate.op == LESS then
8:         scanRes = _mm256_cmp_epi32_mask(decoded, predicate.val, _MM_CMPINT_LT);
9:     end if
10:    output(scanRes)
11:  end while
12: end function

for SIMD using SSE4 instructions. Instead of computing delta between adjacent numbers, Lemire’s algorithm computes and stores delta between number pairs whose index are differ by 4 (as SSE4 registers can hold 4 32-bit integers). For example, given number \([a_0, a_1, \ldots, a_7]\), it stores \([a_0, a_1, a_2, a_3, a_4 - a_0, a_5 - a_1, a_6 - a_2, a_7 - a_3]\). When performing decoding, it loads every 4 entries into a SSE4 word and performs SIMD add operation to get original data. This variation speeds up decoding at the cost of storage space. In average, delta between these number pairs is four times of delta between adjacent numbers, and cost 2 more bits per entry for storage. When migrating this algorithm to larger SIMD word such as AVX-512, the extra space cost can be up to 4 bits per entry.

6.3 Experiments

We use an experiment platform equipped with 2 Intel(R) Xeon(R) Silver 4116 CPUs@2.10GHz, and 190G memory. SIMD codes are compiled using GCC 5.4.0, with -O3 flag. Software platforms used in the experiment include JDK Version 1.8.0_152, Scala Version 2.12.4, Apache Parquet version 1.9.0.
6.3.1 Microbenchmarks

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

![Graphs showing performance comparison between SBoost, BitWeaving-H, and SIMDScan.](image)

Figure 6.5: SBoost Performance on Bit-Packed Data

**Data Filtering on Bit-Packed Integer**

Figure 6.5 shows the experimental result of SBoost’s filter operation on bit-packed encoded integers. In Figure 6.5a we compare SBoost with Willhalm’s SIMDScan algorithm [54, 34], rewritten in AVX-512, and Apache’s Parquet implementation, rewritten in C++. We can see that both algorithms outperform 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 sizes.

SBoost achieves higher efficiency on smaller entry size primarily due to higher parallelization. While SIMDScan uses one 32-bit lane for each bit-packed entry, SBoost can fit more than one entry in each 32-bit lane and compare them in parallel, thus achieves higher throughput for smaller entry sizes. SBoost is able to achieve up to 12x performance compared to SIMDScan (over 18 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 [37], rewritten in AVX-512. As mentioned before, BitWeaving-H 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 lanes, with one bit reserved between each entry. We show that SBoost outperforms BitWeaving-H by 10~25% for small entry sizes, again due to higher parallelization. In Figure 6.5b, we show that SBoost outperforms BitWeaving-H by 10~25% for small entry sizes, again due to higher parallelization. In Figure 6.5c, 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).

We see that SBoost does not only outperform previous algorithms on tightly bit-packed integers, it also achieves same or better performance than BitWeaving-H. This shows using SBoost with tightly bit-packed integers is the best choice for both data filtering speed and storage efficiency.

Next, we propose a instruction modification that is able to further improve the efficiency of this algorithm, and may inspire future research. As is described in Section 6.2.1, our algorithm needs 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 limitation, we implement a software AVX-512 add/sub instruction and test its performance. We also evaluate the throughput if this 512-bit arithmetic instruction is supported and it takes the same cycles as 64-bit arithmetic operation, by counting the number of instructions executed. 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 has potential to further improve throughput and we explore the possibility of using dedicated hardware for a hardware implementation of this instruction to verify this in the future.

Finally, we conduct a performance evaluation on filter for dictionary encoded data. Previously we mention that for filter on dictionary-bit-packed encoded data, we use a order preserving dictionary and rewrite query to convert the operation into a filter on bit-packed encoded integers. We omit the result here for succinctness as it is identical to what is shown in Figure 6.5a. As a summary, SBoost is able to achieve nearly two orders of magnitude throughput comparing to Parquet, and can filter up to 18 billion bit-packed entries per second.

**Data Filtering on Run-Length Encoded Integer**

Next, we report our experimental result of SBoost’s filter performance on run-length encoded integer. We vary both value field size and run-length field size, and report the result in Figure 6.6. Based on analysis on a real-world public dataset collection containing over 15,000 columns we have seen that over 99% of the datasets 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 a 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.

**Decoding Delta Encoded Integer**

We report our experiment on SBoost’s decode algorithm for Delta encoding. We compare our algorithm with the following methods.

- Scalar Decoding algorithm, which extract entries and computes the cumulative sum entry by entry.

- Lemire’s vectorized variation of delta encoding [34], rewritten using AVX2. Parquet uses a similar encoding format as is used in this algorithm.

In Figure 6.7 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 reaches 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 outperform the scalar method by one order of magnitude. This shows that if one need to process standard delta encoding or save storage space, SBoost is still a good choice.

Overall, we show that SBoost’s algorithms achieves a similar or better performance comparing to previous state-of-art results, especially on small entry sizes, and improves space utilization by using standard (tight) encoding. SBoost also has obvious advantages comparing to widely used open-source implementations, and exhibits great potential in speeding up database queries.

### 6.3.2 Boosting JVM-based Columnar Stores

In this section, we demonstrate our experiment results of using SBoost to speed up data filtering/decoding in Apache’s Java implementation of Parquet, and further improve query
Figure 6.6: SBoost Performance on Run-Length Encoded Data
efficiency. We hand-craft a simple query engine, and execute TPC-H queries against both SBoost and Parquet. SBoost utilizes JNI to invoke SIMD algorithms for columns with supported data type and encoding, while retreating to Parquet’s default implementation for columns that are not supported.

As SBoost aims at improving table filtering /decoding speed, we choose Q1 and Q6 from TPC-H queries as they only involves select/project operators. We use the TPC-H data generator to generate test datasets with scale varied from 1 to 30, and read files from both disk and memory (ramdisk) – simulating executions in both OLAP data stores and in-memory data stores.

We encode string columns \texttt{shipdate}, \texttt{line\_status}, and double columns \texttt{extend\_price}, \texttt{discount}, \texttt{tax} with dictionary-bit-packed encoding using a order-preserving dictionary, and integer column \texttt{quantity} with bit-packed encoding. We use SBoost \texttt{filter} to execute predicates on \texttt{shipdate}, and \texttt{quantity}, and use \texttt{decode} to extract \texttt{line\_status}.

The experiment results are shown in Figure 6.8. For execution against files stored in both physical disk and in ram disk (simulating a in-memory database), we observe similar results. In Q1, the only predicate is on \texttt{shipdate} column, which can be executed efficiently.
with SBoost. In addition, quantity can benefit from SBoost’s decode function. As a result, SBoost is one order of magnitude faster than Parquet’s default implementation. For Q6, there are four columns involved in predicate execution, of which only two (quantity and shipdate) can be speed up using SBoost. The projected columns are all of double type thus do not benefit from SBoost. Even with these limitations, SBoost uses only 45% of Parquet’s execution time.

In addition, we notice that the time difference between in-disk and in-memory execution, which is caused by I/O latency, is relatively small (5% ~ 10% of total time). As our experiment platform has large memory capacity, data files stored on hard disk can be efficiently read into page cache upon first access, making later operations equivalent to in-memory operations. This also shows that CPU computation, rather than disk IO, becomes the critical performance bottleneck for these queries, which further justify the effectiveness of our approach.

Overall, we believe this result clearly demonstrate SBoost’s potential application in both disk-based OLAP and in-memory databases.

6.3.3 Scalability

In this section, we study the scalability of SBoost algorithms. It is straight forward to parallelize algorithms we introduce in this paper for bit-packed encoding, run-length encoding, and dictionary encoding. We simply split the input/output into multiple slices and process 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
Figure 6.8: Accelerating Queries in Parquet
Figure 6.9: Scalability of Bit-packed filter

Figure 6.9 shows the performance of bit-packed filter algorithm using multithreading. Run-length filter and dictionary filter, which are based on the same algorithm, exhibit similar patterns.

It can be noticed that multithreading 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 of 3, adding more threads causes throughput to drop around 10%. For all other entry sizes, throughput stalls at some plateaus.

The multi-threaded Delta decode algorithm, as is demonstrated in Figure 6.10, exhibits a similar pattern. Using more threads helps in the beginning, but no longer has obvious effect after using more than 16 threads.

This result is likely caused by hardware limitation. Our hardware platform is equipped with two CPUs, each with 12 cores. As the decoding process is highly CPU intensive, when
the number of threads exceed available cores of each socket, system performance will not benefit from more threads.

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

Figure 6.10: Scalability of Delta decode
CHAPTER 7
CONCLUSION

In this paper, we evaluate the impact of encoding selection given a large corpus of diverse datasets. In particular, we evaluate default methods provided by a popular open-source columnar framework, a state of the art decision tree, and propose a lightweight data-driven encoding selection that models the ideal encoding given a particular implementation and corpus of datasets. We study how encoding and popular compression algorithms influence each other, and provide guidelines on how to properly choose encoding/compression combinations.

We further analyze attributes that do not encode well, yet exhibit good compression under byte-oriented compression, and propose a framework to discover and extract sub-attributes from string columns to improve compression. We believe this work demonstrates weaknesses in existing methods for encoding selection, differences in encoding implementations, serves as a general guideline on a data-driven encoding selection, and highlights opportunities for further research on columnar encoding.

Hardware acceleration plays an important role in database research. Among all possible methods, SIMD has exhibited great potential, with advantages such as direct memory access and fused control flow. In this paper, we introduce novel SIMD algorithms for prevalent encoding schemes that support predicate execution directly on encoded data. Our algorithms work on standard encodings, requiring no additional storage space or special file format, yet providing lightening processing speed. Our data filter algorithm for bit-packed encoded integer and dictionary-bit-packed encoded integer / string can process over 18 billions numbers per second. Our algorithm for delta encoded integers and run-length encoded integers also achieves a throughput of over 1 billion numbers per second. We implement these algorithms and build a columnar data store SBoost based on Apache’s Parquet. Our experimental results demonstrate that the new algorithms outperform their counterparts by at least one
order of magnitude. It reduces query time by over 60% for on-disk queries and over 80% for in-memory queries.

In the future, we plan to extend this work in several directions. We observe several limitations due to lacking SIMD instruction support, and are interested in developing accelerators of our algorithms to further improve efficiency. Furthermore, we would like to utilize our bit-packed data filtering algorithm for faster table joins and aggregations directly on encoded data.
APPENDIX A

THE CORRECTNESS OF DATA FILTERING ALGORITHM
ON BIT-PACKED DATA

A.1 Proof of Equality Test on bit-packed data

In this section, we prove the correctness of Theorem 1.

Proof.

\[ x = a \iff d = 0 \]
\[ \iff d_{msb} = 0 \text{ and } d_{rb} = 0 \]

\[ d_{rb} = 0 \iff d_{rb} + \sim m \text{ does not generate carry} \]
\[ \iff (d_{rb} + \sim m)_{msb} = 0 \]
\[ \iff ((d \& \sim m) + \sim m)_{msb} = 0 \]

Thus we have

\[ x = a \iff d_{msb} = 0 \text{ and } ((d \& \sim m) + \sim m)_{msb} = 0 \]
\[ \iff (d \mid ((d \& \sim m) + \sim m))_{msb} = 0 \]

Noticing that all operations in the proof does not cause carry beyond the \(n\)-bit boundary, the correctness of Equation (6.2) directly follows the theorem.
### A.2 Proof of Range Test on bit-packed data

We prove the correctness of Theorem 2 by proving the following theorem for comparing two numbers without carry.

**Proof.** There are two possible cases when \( x < a \).

- \( x_{msb} = 0 \) and \( a_{msb} = 1 \)

- \( x_{msb} = a_{msb} \) and \( x_{rb} - a_{rb} \) causes a carry

In the first case,

\[
x_{msb} = 0 \text{ and } a_{msb} = 1 \iff (\sim x \& a)_{msb} = 1 \tag{A.1}
\]

In the second case,

\[
x_{msb} = a_{msb} \iff (x \oplus a)_{msb} = 0 \tag{A.2}
\]

\( x_{rb} - a_{rb} \) generate a carry

\[
\iff (x \& \sim m) - (a \& \sim m) \text{ generate a carry}
\]

\[
\iff [(m + x \& \sim m) - (a \& \sim m)]_{msb} = 0 \tag{A.3}
\]

\[
\iff [(x|m) - (a \& \sim m)]_{msb} = 0
\]

\[
\iff c_{msb} = 0
\]

Combining the two cases we have

\[
x < a \iff (\underbrace{(a \& \sim x)}_{\text{Equation (A.1)}} \mid (\underbrace{\sim(a \oplus x)}_{\text{Equation (A.2)}} \& \underbrace{\sim c}_{\text{Equation (A.3)}}))_{msb} = 1
\]
And by boolean algebra we have

\[ \sim(a \& \sim x)[(\sim(a \oplus x) \& \sim c)] \]

\[ =(x \mid \sim a) \& ((a \oplus x) \mid c) \]

\[ =(x \& \sim a) \mid ((x \mid \sim a) \& c) \]

\[ =((\sim a \& (x \mid c)) \mid (x \& c) \]

\[ =t \]

This shows \( x < a \iff t_{msb} = 0 \) and complete the proof.

Observing that Equation (6.4) is using \( ?? \) to compute the result of \( x < a \) and \( x < b \), and

\[ R[i]_{msb} = 1 \iff (x < a) \oplus (x < b) \]

\[ \iff a \leq x < b \]

This complete the correctness proof of Equation (6.4).
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

\[
\begin{align*}
    r[0] &= a[0] + b[0] \\
    & \vdots \\
\end{align*}
\]

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 5. 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 \(a[i + 1] + b[i + 1]\) 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 5 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_epi64_mask (0xff, nc, a, _MM_CMPINT_LT);
6: wcval = _mm512_mask_cmp_epi64_mask (0xff, wc, a, _MM_CMPINT_LT);
7: blendIdx = ((wcval & 0x7e) \ll 6) | (ncval & 0x7f);
8: blend = BLEND_TABLE[blendIdx];
9: return _mm512_blend_epi64(nc, wc, blend);
10: end function

The blend table stores the correspondence 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 B.1, 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. This gives us the magic number seen in line 7 of Algorithm 5.

It can also be noticed from Figure B.1 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 \( 1?01?0\). 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 B.1 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 6.
Algorithm 6 Compute 512-bit add Blend Table

for i = 0 to 8191 do
  wc = (i ≪ 6);
  nc = i & 0x7f;
  usenc = true
  result = 0
  for j = 0 to 7 do
    current = usenc? nc:wc;
    if !usenc then
      result —= (1 ≫ j)
    end if
    usenc = (current & (1 ≫ j)) == 0
  end for
  BLEND_TABLE[i] = result;
end for
REFERENCES


[16] 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.


