Predicate Pushdown in FastLanes

MSc Thesis *(Afstudeerscriptie)*

written by

Raufs Duŋamalijevs
(born March 29th, 2000 in Riga, Latvia)

under the supervision of **Azim Afroozeh**, **Prof Dr Peter Boncz** and **Dr Balder ten Cate**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
December 6, 2024	Prof Dr Peter Boncz
	Dr Balder ten Cate
	Dr Malvin Gattinger
	Dr Ronald de Haan
	Prof Dr Stefan Manegold

Institute for Logic, Language and Computation

Abstract

This project explores predicate evaluation for the FastLanes file format within the framework of cascaded encoding, which encodes the data in multiple layers to achieve higher compression ratios. Predicate pushdown is an optimisation technique that accelerates selective queries by applying filters directly within the data scanning process, thus reducing the volume of data entering the query execution pipeline. While predicate pushdown was widely studied for single-layer encodings, it was not examined in the context of cascaded encodings. Evaluating predicates on data with multi-layer cascades introduces new challenges, such as determining whether to fully decode the data before filtering or to apply filters on partially decoded data. Furthermore, previous works do not utilise data parallelism to its full extent, missing a critical opportunity for performance improvement of predicate evaluation. We address these gaps by developing a fast and portable predicate pushdown technique that functions across multiple architectures without requiring platform-specific implementations. We introduce a data-parallel method for predicate evaluation that stores results in a bitmap with a specific layout, enabling rapid evaluation across multiple columns. In addition, we demonstrate that certain encodings can be partially decompressed and evaluated, avoiding full decompression and increasing efficiency.

Acknowledgements

My deep thanks go to Azim Afroozeh (عظیم افروزه, with two dots missing in the name – Overleaf, please improve Persian integration!), a true dojo master not only in table tennis but also in the art of highly efficient programming. Peter Boncz, thank you for welcoming me into your group, offering invaluable advice, and for the cookies you brought almost daily. Balder ten Cate, I'm grateful for your support and for helping make this thesis well-written. Special thanks to Malvin Gattinger for chairing my thesis defence, Ronald de Haan for teaching me all things complexity with great spirit and enthusiasm, and Stefan Manegold for our lunchtime conversations about everything from Kaliningrad to the education system.

I am grateful to the Priests of Staburags for insights beyond the scope of this section. Jeewon Heo (허지원), your optimism kept the group's spirits high. Ziya Muxtarov, you somehow manage to be shy and hilarious at the same time! Ronald de Wolf, I appreciate your guidance and the many hours we spent exploring the theoretical foundations of NISQ devices. I'm also grateful to Professor John McCall, whose passion for science and support during my later undergraduate years left a lasting impact on my journey. Alyssa Renata, your energy is truly inspiring. Thank you, Niels Nes, for enabling me to perform benchmarks on Apple devices. Leonardo Xavier Kuffo Rivero, I'm grateful for the beautiful music you make and for providing your device for benchmarks. Niclas Haderer, you made me love German bread even before I tried it. Madelon Hulsebos, thank you for helping me lure the group into a different canteen with better vegetarian options on fish Fridays. Boaz Leskes (Overleaf, you can do better – please add the support of the Hebrew alphabet!), thank you for bringing humour to our lunch breaks. Thank you, Daniel Gomm, for the enjoyable conversations. Connor de Bont, I know everything about custom coffee machines, thanks to you! Adithya Krishnan (the name is originally written in Tamil script, but Overleaf is proving consistent... in

letting us down yet again), thank you for being so wonderfully multidisciplinary, from AI in databases to how poop flows in sewers and even to making and selling pasta! Steffano Papandroudis (**Στφανο Παπανδροδη**), sharing rooms with you was a joy. Till Döhmen, I enjoyed the jokes you've made with a dead serious face. Daniël ten Wolde and Lotte Felius, I appreciate how you introduced me to Dutch culture in such an engaging way (and for just being a pleasure to be around). Effy Xue Li (李雪), your kindness stands out – I'll never forget how we took your friend to the hospital after bouldering, a tragic yet oddly fun experience. Sven Hielke Hepkema, our discussions about taxes and industry made me richer – unfortunately, only in the future! Paul Groß, even though you almost always won at table tennis, I thoroughly enjoyed our games. Florian Gerlinghoff, your constant smile brightened the office. Philipp Klocke, thank you for bringing style to our group. Elena Krippner, your early arrivals made the mornings less lonely and created a productive atmosphere. And to Jelte Fennema, just a cool guy – thanks for being you. My friend Kirill Kopnev (Кирилл Копнев) inspired me to pursue a master's in logic after discovering his blog. Studying together and learning from your ~~mistakes~~ experiences has been invaluable. I'm also grateful to all the staff at CWI – the friendly cleaners, canteen workers, and the director who keeps everything running smoothly.

Most importantly, I want to thank my girlfriend, Karīna Sudņicina, for making our home a place of comfort and support. Finally, my deepest appreciation goes to my parents, Rabi Duņamalijevs, for teaching me mathematics from a young age, and Čičaka Karimova, for supporting me in all my endeavours.

Contents

1 Introduction **1**

I Background **4**

2 Compression **5**
2.1 General-Purpose Compression 5
2.2 Light-Weight Compression 6
2.2.1 Bit-packing 7
2.2.2 Dictionary Encoding 7
2.2.3 Run-Length Encoding 7
2.2.4 Frame of Reference 8
2.2.5 Delta Encoding 8
2.2.6 Cascaded Encoding 8
2.2.7 Patched Encoding 9

3 The FastLanes File Format **10**
3.1 Single Instruction, Multiple Data (SIMD) 11
3.1.1 Scalar Code and Auto-Vectorisation 11
3.1.2 Intrinsics 12
3.2 FastLanes Compression 13
3.2.1 Interleaved and Unified Transposed Layouts 13
3.2.2 Cascaded Encoding in FastLanes 15
3.3 FastLanes Decompression 16
3.3.1 Dictionary Encoding 16
3.3.2 Run-Length Encoding 16
3.3.3 Dictionary + Run-Length Encodings 17

4 Predicate Pushdown **18**
4.1 SIMD in Predicate Pushdown . . . 19
4.2 Selection Data Structure . . . 20
4.2.1 Selection Vector . . . 20
4.2.2 Selection Byte Map . . . 21
4.2.3 Selection Bitmap . . . 21
4.2.4 Reverse Selection Vector . . . 22
4.2.5 Adaptive Approach . . . 22
4.3 Dictionary Optimisation . . . 22
4.4 Other Optimisations . . . 23
4.4.1 Min/Max Skipping . . . 23
4.4.2 Pruning Optimisation . . . 24

II Research **25**

5 Predicate Evaluation on Uncompressed Data **26**
5.1 Predicate Evaluation Algorithms . . . 27
5.1.1 Scalar: raw . . . 28
5.1.2 Scalar: native . . . 31
5.1.3 Scalar: mem . . . 32
5.1.4 Scalar: or . . . 32
5.1.5 Intrinsics . . . 33
5.1.6 Results . . . 35
5.2 Merge optimisation . . . 38
5.3 Low Selectivity Optimisation . . . 40
5.4 Discussion . . . 44

6 Predicate Evaluation on Semi-Compressed Data **46**
6.1 Dictionary Encoding . . . 47
6.2 Run-Length Encoding . . . 52
6.3 Dictionary + Run-Length Encodings . . . 52
6.4 Discussion . . . 56

7 Discussion and Future Work **57**

A Benchmarking Environment **60**

B Algorithms **61**

Acronyms

AVX Advanced Vector Extensions.

CPU Central Processing Unit.

DELTA Delta Encoding.

DICT Dictionary Encoding.

FOR Frame of Reference.

GPU Graphics Processing Unit.

ISA Instruction Set Architecture.

RAM Random Access Memory.

RLE Run-Length Encoding.

SIMD Single Instruction, Multiple Data.

SSE Streaming SIMD Extensions.

Chapter 1

Introduction

FastLanes is an open-source project that seeks to enhance big data file formats such as Parquet and ORC by addressing two key needs: the rise of new workloads, particularly machine learning data engineering pipelines, and the potential for significantly improved compression and access speeds on current workloads. There are two main types of compression: general-purpose and lightweight. General-purpose compression, like Snappy or Zstd, can handle arbitrary data types and achieve high compression ratios at the expense of decompression speed. In contrast, lightweight compression methods focus on specific patterns in the data and are faster, making them particularly useful for modern databases where quick access to data is essential. In [9], authors describe how to make the decoding of popular lightweight compression schemes (Dictionary Encoding (DICT), Frame of Reference (FOR), Delta Encoding (DELTA), and Run-Length Encoding (RLE)) even faster through better data parallelism. This raises the logical question: is it possible to compress data further without relying on general-purpose compression, which introduces significant decompression overhead?

The FastLanes project addresses this challenge by implementing *cascaded encodings* [14]. It involves applying multiple lightweight compression schemes in sequence to the same data, effectively encoding data recursively. For example, consider a string column in a database: first, DICT replaces each unique string with an integer code, transforming `['Sales', 'Sales', 'Marketing', 'Sales', 'Engineering', 'Engineering', 'Engineering', 'Marketing']` into `[0, 0, 1, 0, 2, 2, 2, 1, 0]`. These integer codes may then exhibit patterns, such as consecutive repetitions, which can be further compressed using RLE, resulting in `[(0,2), (1,1), (0,1), (2,3), (1,1)]`. By layering compression schemes in this manner, each method targets specific patterns or redundancies in the

data, improving overall compression ratios while maintaining fast decompression speed. Since cascaded encoding relies on both the input and output of each compression layer being sequences of values, it cannot be applied to general-purpose compression.

Pushed-down predicate evaluation, also known as *predicate pushdown*, is a query optimisation technique that involves applying filters early on (*pushing them down*). Our project focuses on applying these filters while scanning the data, which is the earliest opportunity to do so. For example, consider the SQL query:

```
SELECT e.name
FROM Employees e
JOIN Departments d ON e.department_id = d.department_id
WHERE e.age > 30 AND d.department_name = 'Engineering';
```

Without predicate pushdown, the system could scan the entire `Employees` and `Departments` tables, perform the join on all records, and then apply the filters, resulting in the processing of numerous unnecessary records. Conversely, with predicate pushdown, the filter conditions `e.age > 30` and `d.department_name = 'Engineering'` are applied when reading the data, ensuring that only relevant records are retrieved and joined. This is critical for selective queries, where studies have shown it can yield up to an eightfold speedup [13].

Predicate pushdown has been extensively studied for single-layer encodings [13, 18, 22, 25, 26, 33, 34]. However, modern compression schemes often involve multiple layers, and to our knowledge, no study has focused on techniques applicable to cascaded encodings. Evaluating predicates on data encoded with multi-layer cascades is more complex, raising previously unaddressed questions. Namely, should we decode all the layers of the cascaded encoding before applying a filter, or would applying the filter on partially decoded data be faster? Moreover, previous works do not utilise data parallelism to its full extent for predicate evaluation, thereby missing a critical opportunity for performance improvement. Our work fills this gap by developing a predicate pushdown technique for FastLanes that is both fast and portable, meaning it can run across various architectures without the need for multiple platform-specific implementations, making the codebase simple to maintain and extend.

Our main contribution is a data-parallel predicate evaluation method, which we achieve by storing the results in a selection bitmap with a specific layout. This approach enables rapid predicate evaluation across multiple columns. When the number of columns is four, our method outperforms the current state of the art across all data widths, providing up to a 1.85x speedup in evaluation, with

performance gains increasing as the number of columns grows. Additionally, we demonstrated that instead of fully decompressing cascaded encodings, certain encodings can be decompressed to an intermediate representation and evaluated directly, achieving up to a 40.81x speedup over full decompression and evaluation.

As such, in this thesis, we answer three research questions:

1. Can we develop data-parallel predicate evaluation methods that are both fast and portable across various hardware architectures without resorting to multiple platform-specific implementations?

2. How can we efficiently perform predicate evaluation on data compressed with cascaded encoding without fully decompressing the data?

3. Which selection data structure – selection bitmap, selection byte map, selection vector, or reverse selection vector – is most suitable for efficient predicate evaluation, considering factors such as data parallelism, selectivity, and performance trade-offs?

We start Chapter 2 by introducing the two types of compression: lightweight and general-purpose. We examine their strengths, focusing on lightweight schemes relevant to this thesis. We explain cascaded encoding, a technique used in FastLanes, where multiple such schemes are applied recursively to increase compression ratios. Chapter 3 explores the FastLanes file format. We introduce Single Instruction, Multiple Data (SIMD) and discuss FastLanes' virtual 1024-bit SIMD register, Interleaved and Unified Transposed Layouts and [de]compression. Chapter 4 examines predicate pushdown, reviewing existing optimisation techniques and identifying the challenges of adapting them to FastLanes. Chapter 5 explores the most efficient methods for predicate evaluation in FastLanes on decompressed data. Chapter 6 presents algorithms for predicate evaluation on partially compressed data. We benchmark these algorithms against ones that operate on fully decompressed data to determine whether compressed execution offers better performance. In Chapter 7, we summarise our findings and propose possible directions for future research. In addition, we include Appendix A, where we present the devices and settings used in the benchmarks, and Appendix B, where we include the developed algorithms omitted in the main body of text.

Part I

Background

Chapter 2

Compression

There are two main types of compression: general-purpose and lightweight. General-purpose compression methods can handle arbitrary data types and achieve high compression ratios at the expense of [de]compression speed. Lightweight compression methods are faster and focus on specific patterns in the data, making them useful for modern databases where quick access to the data is essential. In this chapter, we review both, focusing on lightweight compression schemes relevant to the thesis. We also look at cascaded encoding, where several lightweight compression schemes are applied on top of each other to improve compression ratios.

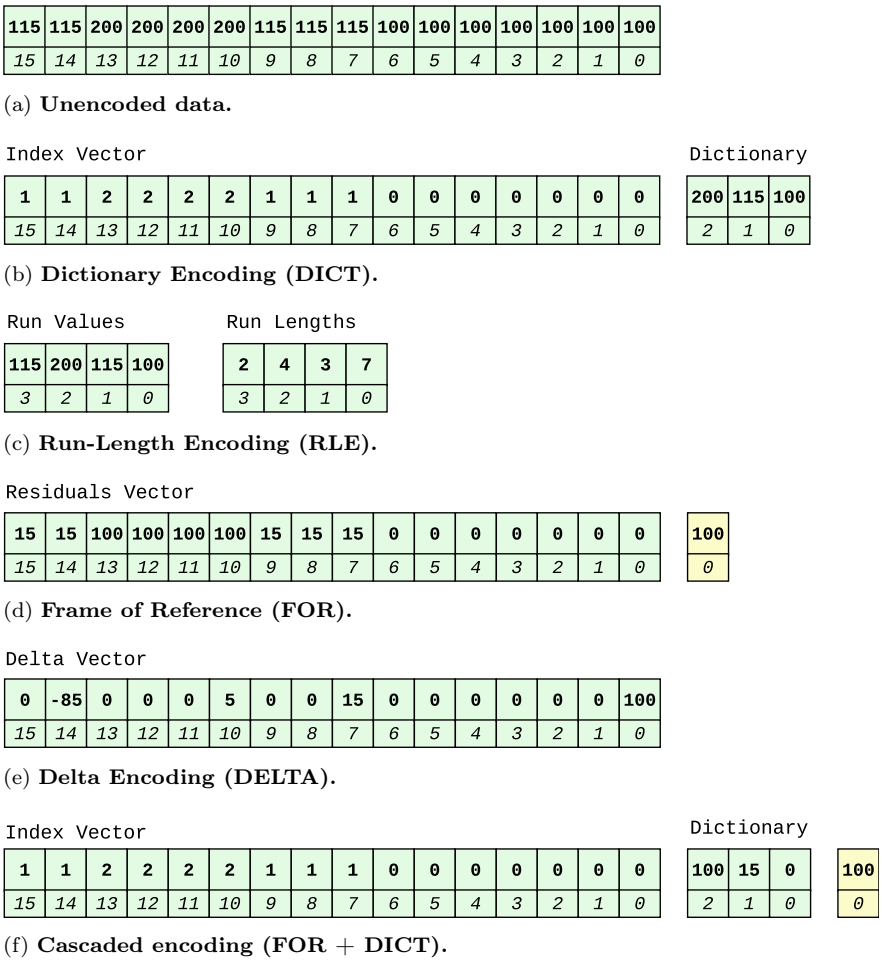
2.1 General-Purpose Compression

General-purpose compression consists of algorithms that can compress data without requiring knowledge about its specific data type [21, 37]. They work on a stream of bytes allowing compression of any data. Such algorithms usually exploit repeating patterns and can achieve relatively high compression ratios [23]. This flexibility and good compression ratios make it widely used.

However, general-purpose compression methods have relatively slow [de]compression speed [9, 23]. Since they compress large data blocks containing many vectors, accessing a specific record requires decompressing the entire block. Usually, they are large and exceed the Central Processing Unit (CPU) cache capacity, requiring them to be loaded into main memory, fully decompressed, and stored before executing queries on individual vectors. In contrast, decompressing data one vector at a time allows each vector to fit into the cache, enabling immediate decompression and query execution [38]. Therefore, general-purpose compression is less

effective for systems that require quick access to individual data records, as it involves additional overhead from loading large blocks into memory, decompressing them, and then processing each vector sequentially.

2.2 Light-Weight Compression



(a) **Unencoded data.**

(b) **Dictionary Encoding (DICT).**

(c) **Run-Length Encoding (RLE).**

(d) **Frame of Reference (FOR).**

(e) **Delta Encoding (DELTA).**

(f) **Cascaded encoding (FOR + DICT).**

Figure 2.1: Unencoded data and how it is compressed by lightweight compression schemes.

The demand for faster [de]compression led to the development of lightweight compression schemes such as DICT, FOR, DELTA, and RLE [9]. Lightweight compression exploits specific data type patterns and achieves better [de]compression times. This makes it well-suited for modern database systems, where the speed

of data retrieval is often a priority [35]. As a result, some columnar storage formats leverage it by default. For example, Parquet uses RLE or DICT with bit-packing on dictionary codes. Additionally, unlike general-purpose compression, lightweight schemes compress the actual data rather than blocks of bytes, enabling random access to the data. Until recently, lightweight compression was often associated with lower compression ratios compared to general-purpose compression. However, with the emergence of cascade encoding [14], this trend has changed [9, 23]. In this section, we review lightweight compression schemes relevant to our work and the concept of cascaded encoding.

2.2.1 Bit-packing

Bit-packing is a compression method that reduces the number of bits used to represent values. Typically, fixed-width data types like 32-bit integers use all 32 bits, even when numbers are small. Bit-packing compresses the data by using only the minimum number of bits required, which is particularly effective for datasets with small ranges. For example, if the largest value in the dataset is 3, only 2 bits are needed to represent all values: `00` for 0, `01` for 1, `10` for 2, and `11` for 3.

2.2.2 Dictionary Encoding

Dictionary Encoding (DICT) is useful when the data contains a small number of unique values [32]. It builds a dictionary where each unique value is mapped to a smaller integer key. During compression, original data entries are replaced with related keys, and decompression restores the original data by looking up the dictionary values. This method works best when data contains a small set of frequently repeated values. On the other hand, if there are few repeated values, the dictionary can become as large as the raw data, taking up more space than before encoding. For better compression, we can assign shorter keys to values occurring more frequently and compress the integer values in DICT using bit-packing.

2.2.3 Run-Length Encoding

Run-Length Encoding (RLE) is useful when data contains consecutive repetitions of the same values [30]. Instead of storing each occurrence of a repeated value, RLE compresses the data by recording the value once and the number of times it repeats consecutively. RLE is particularly useful in columnar databases where

values of the same type are stored together, increasing the likelihood of repeated runs and enabling better compression results.

2.2.4 Frame of Reference

Frame of Reference (FOR) works by selecting the minimum value in the data as a reference point and then subtracting this reference from all other values [19]. The result is a set of smaller integers. These smaller numbers require fewer bits to be represented and can be bit-packed for more efficient storage.

2.2.5 Delta Encoding

Delta Encoding (DELTA) stores the differences between consecutive values instead of the original ones [28]. If the data consists of numbers close to each other, the differences (or deltas) between them are small. When we apply bit-packing, these small deltas take up less space, leading to better compression.

We reconstruct the original values during decompression by adding each delta to the preceding value. While this process is simple, it introduces a dependency: we cannot decode a value without knowing the one before it. Random access to any value in the compressed data is not possible, as we might need to decode multiple preceding values first. There are variations of DELTA that allow faster [de]compression by utilising SIMD [9, 24, 36].

2.2.6 Cascaded Encoding

Cascaded encoding is a technique where multiple lightweight compression schemes are applied in sequence to achieve better compression ratios [14]. For example, a string column might first be compressed using DICT to replace strings with integer codes, and then consecutive repeating codes can be further compressed by RLE. This layered approach allows each scheme to focus on a particular pattern, increasing the overall compression. Since cascaded encoding relies on both input and output being sequences of values, it cannot be applied to general-purpose compression. Additionally, not all combinations of schemes are equally effective.

In some cases, encodings must be combined to achieve compression (for example, FOR is only helpful if we subsequently apply bit-packing). Kernel fusion can optimise this by merging the encoding steps of two schemes into a single function, eliminating unnecessary data transfer operations. Both BtrBlocks and

FastLanes use this optimisation and demonstrate that cascading achieves compression ratios comparable to general-purpose compression without the significant [de]compression overhead [23, 27].

2.2.7 Patched Encoding

Standard encoding techniques often apply uniformly across all values, which can be inefficient when there are outliers in the data distribution. These rare entries can inflate the size of encoding structures like dictionaries, requiring more bits for representation and reducing compression efficiency. To address this issue, Patched Encoding, or simply *patching*, is employed. Patching involves storing outlier values separately as uncompressed exceptions, allowing the main encoding, like DICT, to remain compact by focusing on the common values. There are patched versions of DICT, FOR and DELTA [24, 38].

Chapter 3

The FastLanes File Format

In big data processing, efficient storage formats are crucial for performance and scalability. Parquet and ORC are columnar storage formats widely used for storing large datasets, designed in 2013 and 2016, respectively. However, both the hardware landscape and workload requirements have evolved significantly since their inception [35]. The evolution has exposed several limitations in them that newer file formats like BtrBlocks [23] and FastLanes [9, 10] address.

FastLanes is a novel file format designed to enhance data compression ratios and [de]compression speed while maintaining portability across different hardware architectures. In their initial paper [9], the authors show how FastLanes leverages fully data-parallel encodings implemented using portable scalar code that can be auto-vectorised. Subsequent research demonstrates how FastLanes benefits from Graphics Processing Unit (GPU) data processing by allowing more data to fit into GPU memory and enabling faster computation [10]. Tests on Nvidia GPUs showed that queries executed on data compressed using FastLanes can run up to twice as fast compared to uncompressed data, without the slowdowns previously observed with GPU decompression.

As data parallelism is central to FastLanes, understanding Single Instruction, Multiple Data (SIMD) is essential. Therefore, we begin this chapter by explaining the concepts of SIMD, which provides the foundation for the data-parallel techniques employed by FastLanes. We then delve into the key aspects of FastLanes' [de]compression methods relevant to our work.

3.1 Single Instruction, Multiple Data (SIMD)

Modern CPUs often include Single Instruction, Multiple Data (SIMD) capabilities to exploit data-level parallelism. SIMD allows a single instruction to perform operations on multiple data points simultaneously, significantly enhancing computational efficiency for certain tasks. This section explores the concepts of SIMD and discusses two methods of leveraging it: scalar code with auto-vectorisation and the use of intrinsics.

3.1.1 Scalar Code and Auto-Vectorisation

Algorithm 1 Example of auto-vectorisable algorithm.

```
1 void equality(uint8_t* in, uint8_t* out, uint8_t filter) {
2   for (uint16_t i = 0; i < 1024; i++) {
3     out[i] = in[i] == filter;
4   }
5 }
```

Auto-vectorisation is a compiler feature that automatically transforms regular scalar code, which operates on one element at a time, into vectorised code using SIMD instructions [2]. The term *vectorisation* originates from the fact that SIMD instructions operate on vectors of data. Individual elements within these vectors are of fixed length and are typically referred to as *lanes*. Different CPUs support different SIMD Instruction Set Architectures (ISAs):

- Neon introduced 128-bit SIMD registers for the ARM architecture.
- Streaming SIMD Extensions (SIMD) introduced 128-bit xmm registers for the x86 architecture.
- Advanced Vector Extensions (AVX) and its successor AVX2 expanded the x86 SIMD capabilities to 256-bit ymm registers. AVX-512 further expanded this to 512-bit zmm registers. For example, AVX2 allows operating on eight 32-bit integers simultaneously, while AVX-512 doubles that capacity.

Newer processors introduce extended instruction sets that add new capabilities while using the same registers. For example, AVX-512, released with Intel's Skylake microarchitecture, includes instructions like VPCOMPRESSD and VPCOMPRESSQ [7, 8], which compress sparse-packed 32-bit and 64-bit integer values,

respectively, into dense memory or registers based on a mask. A later extension of AVX-512, known as *Vector Bit Manipulation Instructions 2* or *VBMI2* and introduced with Intel's Ice Lake microarchitecture, added instructions like VPCOMPRESSB and VPCOMPRESSW [6], extending compressed write capabilities to smaller data types (8-bit and 16-bit integers, respectively).

Algorithm 1 demonstrates a simple predicate evaluation where we check each element of the input array against a filter and store the result in the output array. Specifically, the i-th element of the output is assigned 1 if the i-th element of the input satisfies the equality, and 0 otherwise. A modern compiler with auto-vectorisation enabled can transform this loop to use SIMD instructions, processing multiple elements per iteration. Although the overall computational complexity remains $\mathcal{O}(n)$ since we still need to process all n elements, vectorisation reduces the constant factor by handling several elements at once, which can significantly speed up execution in practical applications.

Auto-vectorisation works best when loops have a predictable structure with a known number of iterations and no data dependencies between iterations. Data dependencies occur when the outcome of one iteration depends on the results of previous iterations, preventing simultaneous execution of multiple iterations. However, compilers do not always succeed in auto-vectorising code, especially when it includes complex control flow like branching (if-else statements).

3.1.2 Intrinsics

Algorithm 2 Example of an algorithm using AVX2 intrinsics.

```
1  void equality(uint8_t* in, uint8_t* out, uint8_t filter) {
2    __m256i fltr_vec = _mm256_set1_epi8(filter);
3    __m256i one_vec = _mm256_set1_epi8(1);
4
5    for (size_t i = 0; i + 32 <= 1024; i += 32) {
6      __m256i in_vec = _mm256_loadu_si256((__m256i*)(in+i));
7      __m256i cmp_vec = _mm256_cmpeq_epi8(in_vec, fltr_vec);
8      __m256i result = _mm256_and_si256(cmp_vec, one_vec);
9      _mm256_storeu_si256((__m256i*)(out + i), result);
10   }
11 }
```

Even when code is clean, the compiler might not generate the most optimal SIMD

instructions. For instance, the Clang 18.1.0 compiler may not generate the compressed write instructions described earlier, even when they could improve performance [5]. This means that relying solely on the compiler's auto-vectorisation might not always yield the best results.

In such cases, manual intervention might be necessary, such as rewriting the code using *intrinsics*. Intrinsics are special functions provided by the compiler that map directly to specific SIMD instructions [1, 4]. They give programmers more control but reduce portability as the code becomes tied to specific hardware instruction sets. Consider Algorithm 2, an analogue of Algorithm 1 that uses AVX2 intrinsics instead of relying on auto-vectorisation. While it is possible to write separate implementations for each ISA, this reduces the maintainability of the code and adds technical debt as new instruction sets will inevitably emerge.

3.2 FastLanes Compression

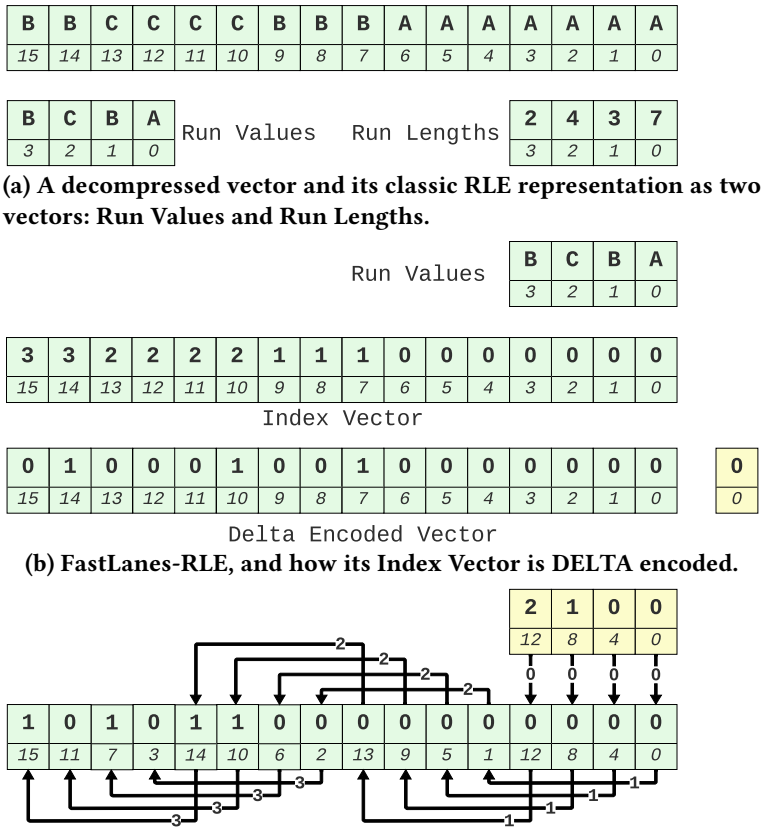
FastLanes file format employs advanced storage and compression techniques. In this section, we discuss the key aspects of these methods.

3.2.1 Interleaved and Unified Transposed Layouts

Bit-packing is a component of many lightweight compression schemes. In horizontal bit-packing, data is stored consecutively, with bits of adjacent values tightly packed together. However, this layout hinders the efficient use of SIMD instructions because adjacent values end up in the same lane of the SIMD registers. Since each lane operates on a single value, adjacent values in the same lane prevent parallel processing and require additional instructions to be moved.

FastLanes addresses this issue by separating the logical table format that applications expect from the physical data storage format. On the physical level, the data is not stored consecutively but rearranged into an order that allows faster [de]compression. However, the application still perceives the data in the original, consecutive order. This separation between the logical and physical formats ensures that applications can benefit from improved performance without needing to handle the complexities of the underlying data storage arrangements.

To achieve this, FastLanes introduces the *Interleaved Layout*, which defines the order in which bit-packed data is stored. After bit-packing, rather than storing consecutive bit-packed data with indexes 0, 1, 2, ..., FastLanes rearranges the data so that each lane contains non-adjacent values, allowing for efficient parallel operations without the need for extra data movement instructions. The



(a) A decompressed vector and its classic RLE representation as two vectors: Run Values and Run Lengths.

(b) FastLanes-RLE, and how its Index Vector is DELTA encoded.

(c) FastLanes-RLE reorders the Index Vector in Unified Transposed Layout: compatible with other columns and enabling fast decoding.

Figure 3.1: FastLanes-RLE: a fast and compact encoding scheme targeting in-flight partially compressed vectors [9].

Interleaved Layout is designed for a virtual 1024-bit SIMD register – a size that does not currently exist in CPUs – making it future-proof. Layouts optimised for larger registers can be mapped to smaller registers using multiple narrower SIMD instructions [20]. The implementation uses scalar code that relies on the compiler's auto-vectorisation to utilise the widest available SIMD registers, ensuring adaptability to future hardware without sacrificing portability.

In addition to the Interleaved Layout, FastLanes introduces the *Unified Transposed Layout*, which is another method of storing data to enable efficient SIMD decoding for lightweight compression schemes with data dependencies, such as DELTA. This layout rearranges the data so that each SIMD instruction operates on a batch of independent values, eliminating data dependencies within the

batch. This allows for efficient parallel processing regardless of the SIMD width.

Consider FastLanes-RLE, a new flavour of RLE introduced by FastLanes and depicted in Figure 3.1. Traditional RLE is difficult to vectorise due to data dependencies within the encoded data. FastLanes-RLE transforms RLE into a combination of DICT and DELTA that is compatible with Unified Transposed Layout. Instead of storing pairs of run values and lengths, FastLanes stores run values in a dictionary and encodes their positions using DELTA. This method allows RLE to benefit from SIMD parallelism, making its decoding times in most scenarios many times faster compared to both standard and SIMD-optimised RLE schemes.

3.2.2 Cascaded Encoding in FastLanes

Both FastLanes and BtrBlocks use cascaded encoding but differ in their approach and implementation [9, 23]. Unlike BtrBlocks, which recursively compresses entire column chunks of 64,000 values, FastLanes operates on smaller vectors of 1,024 elements. This approach allows FastLanes to fully decode data at a much smaller granularity, enabling vectorised execution. Smaller vectors fit within the CPU's L1 and L2 caches and are processed immediately, which minimises their spilling into slower Random Access Memory (RAM) during query processing.

Another difference lies in their bit-packing strategies. BtrBlocks' bit-packing is based on the SSE, which creates performance bottlenecks, as SSE's 128-bit lanes do not leverage the full potential of modern CPUs. In contrast, FastLanes adopts an Interleaved Layout that supports the widest ISAs currently available and will automatically support wider ISAs as they are developed, significantly improving [de]compression speeds.

Furthermore, BtrBlocks selects the optimal compression scheme for each column chunk by compressing a small subset of the data with all available lightweight compression schemes. The scheme that achieves the highest compression ratio on the sample is then applied to the entire column chunk. This process is repeated if the output is in a compressible format, up to a user-defined recursion depth, resulting in arbitrary cascade structures. In contrast, FastLanes provides predefined cascaded encodings (39 at the time of writing). Having a fixed number of schemes is beneficial during decompression, as it allows for optimisations tailored to these specific combinations, which we discuss in the next section.

3.3 FastLanes Decompression

The FastLanes file format follows the principle of *compressed execution*. That is, instead of fully decoding the data, FastLanes, whenever possible, partially decodes the data into a state that query engines can use. We refer to such partially decoded data states *intermediate representations*. This design decision offers multiple benefits. Firstly, full decompression is more costly in terms of computation and time. Secondly, running queries on encoded data may be faster because (1) encoded data may be of a thinner data type, allowing simultaneous processing of more values in SIMD registers and (2) compression schemes like RLE condense many values, enabling to process them together.

For most of the available encoding schemes, FastLanes has an appropriate intermediate representation, made possible because the schemes are predefined rather than recursively created at runtime. In Section 6, we explore predicate evaluation algorithms for these intermediate representations and quantify the impact of compressed execution on the speed of running filter queries. This section explores the intermediate representations currently available in FastLanes.

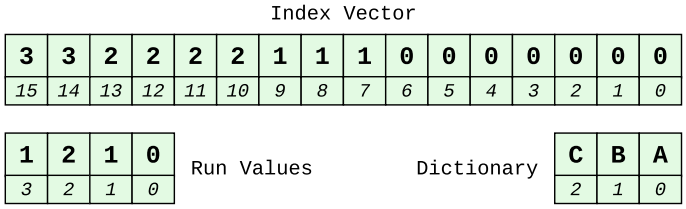
3.3.1 Dictionary Encoding

FastLanes uses two variations of Dictionary Encoding (DICT) as intermediate representations: the *standard* DICT and the *frequency* DICT, where more frequent values are assigned shorter keys. To fully decompress this intermediate representation, it is sufficient to iterate over the data and map each entry to its corresponding value in the dictionary. For clarity, we refer to this process as *flattening*.

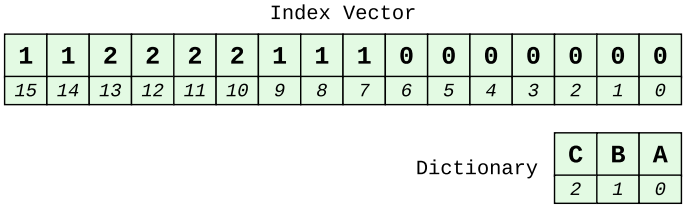
3.3.2 Run-Length Encoding

The RLE intermediate representation differs from both standard RLE and FastLanes-RLE. It resembles FastLanes-RLE, but without DELTA applied to the index vector. If we ignore the DELTA encoded vector in Figure 3.1b, we get the RLE intermediate representation. Upon closer inspection, we see that this representation is similar to DICT, but with the key difference that *run values* may repeat if they are not consecutive in the data, whereas in DICT, each value is unique. Thus, full decompression is analogous to DICT: flatten the *index vector* with the *run values*.

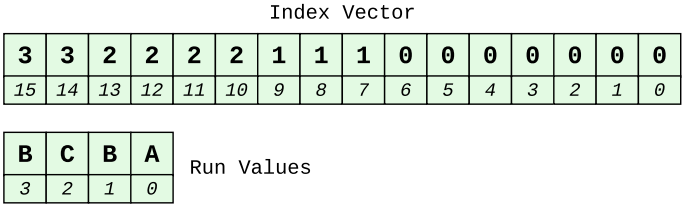
3.3.3 Dictionary + Run-Length Encodings



(a) **DICT + RLE intermediate representation. Equivalent to RLE data with Run Values encoded via DICT.**



(b) **RLE intermediate representation. The result of flattening of Run Values and Dictionary in 3.2a.**



(c) **DICT intermediate representation. The result of flattening of Run Values and Index Vector in 3.2a.**

Figure 3.2: DICT + RLE intermediate representation and two ways to decode one layer of it.

This intermediate representation combines DICT and RLE. As described in the previous section, RLE produces *run values* that may have non-unique entries. In this representation, these *run values* are further encoded using DICT.

There are two efficient ways to fully decode this representation. The first method is to flatten the dictionary with the *run values*, resulting in the RLE intermediate representation, which can then be decoded as described earlier. The second option is to decode the RLE component first by flattening the *index vector* and *run values*. This leaves us with the standard DICT, which is one flattening away from being fully decoded. Both approaches are illustrated in Figure 3.2.

Chapter 4

Predicate Pushdown

Predicate pushdown is a query optimisation technique that involves applying filters early on (*pushing them down*). Our project focuses on applying these filters while scanning the data, which is the earliest opportunity to do so. For example, consider the SQL query:

```
SELECT e.name
FROM Employees e
JOIN Departments d ON e.department_id = d.department_id
WHERE e.age > 30 AND d.department_name = 'Engineering';
```

Without predicate pushdown, the system could scan the entire `Employees` and `Departments` tables, perform the join on all records, and then apply the filters, resulting in the processing of numerous unnecessary records. Conversely, with predicate pushdown, the filter conditions `e.age > 30` and `d.department_name = 'Engineering'` are applied when reading the data, ensuring that only relevant records are retrieved and joined. This early filtering reduces the volume of data entering the expensive stages of the query execution pipeline, such as joins, thereby enhancing overall query performance. Predicate pushdown is essential for achieving good performance in benchmarks like TPC-H [11, 17].

While predicate pushdown has been researched for single-layer encodings [13, 18, 22, 25, 26, 33, 34], to our knowledge, its integration with cascaded encodings has not been explored. Their multi-layer nature makes applying filters without fully decompressing the data first challenging. Moreover, while some techniques employ data parallelism, they typically do so with limited efficiency or restricted hardware support. For instance, [25] introduces *selection pushdown*,

which evaluates predicates directly on bit-packed data and only decodes the values selected. This method works well in Parquet, where query time in benchmarks is dominated by decoding [25]. However, this assumption may no longer apply with FastLanes, where decoding is much faster. Furthermore, it works only on horizontally bit-packed data, which FastLanes avoids in favour of interleaved bit-packing for faster decompression [9]. Applying selection pushdown to interleaved bit-packing is still to be researched, and even if feasible, it would require `PEXT` and `PDEP` instructions, which are exclusive to x86 architecture and are slow on some CPUs [3].

Developing data-parallel predicate pushdown techniques compatible with cascaded encodings is critical for the success of FastLanes. Solving this challenge not only enhances the performance of FastLanes but also addresses a research gap crucial for the file formats of the future. In this section, we review existing optimisation techniques relevant to our work.

4.1 SIMD in Predicate Pushdown

The idea of using SIMD to accelerate predicate evaluation is not new. [22, 33, 34] propose loading bit-packed data into SIMD registers, performing bit-unpacking directly in the registers, and then evaluating the predicate – eliminating the need to store the unpacked data before evaluation. [22] demonstrates how this on-the-fly bit-unpacking can benefit predicate evaluation for data encoded in RLE, DICT, and DELTA. However, these approaches depend on specific ISAs and intrinsics, which makes them neither portable nor future-proof. [13] introduces the use of a selection byte mask to record selections, allowing predicate evaluation on many values in parallel using scalar code. We explore this idea further in the next section.

4.2 Selection Data Structure

When running predicate evaluation, such as determining which entries in a vector are smaller than 3, we need to record which elements satisfy the predicate. Several data structures can handle this, each with its own advantages.

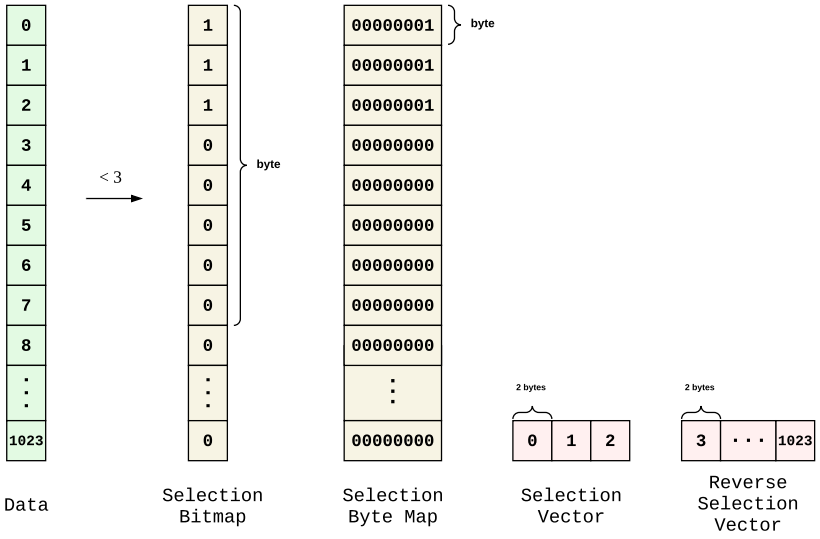


Figure 4.1: Visual representation of different selection data structures as a result of applying < 3 filter on a $0, 1, ..., 1023$ vector. As shown, selection bitmap and byte map always take up 128 and 1024 bytes of memory, while the [reverse] selection vector may require up to 2048 bytes.

4.2.1 Selection Vector

A selection vector holds the indices of elements that meet the selection criteria. This approach is particularly efficient when only a few elements satisfy the predicate [29]. In scenarios with multiple columns, if only a few elements meet the condition after filtering one column, the selection vector allows us to process the next column more efficiently. Instead of evaluating every value in the next column, we can focus only on the elements whose indices are stored in the selection vector, as the others will not be selected in any case. For example, in FastLanes, which processes vectors of 1,024 values, if only one element satisfies the predicate in the first column, we can skip evaluating the remaining 1,023 values in the subsequent column, knowing they are not selected.

Algorithm 3 Example of a predicate evaluation algorithm that uses a selection vector. `in` is the data to be scanned and `out` is the selection vector. The algorithm contains a data dependency on the `size` variable, preventing auto-vectorisation.

```
void equality(uint8_t* in, uint16_t* out, uint8_t filter) {
    uint16_t size = 0;
    for (uint16_t i = 0; i < 1024; i++) {
        out[size] = i;
        size += (in[i] == filter);
    }
}
```

Predicate evaluation with a selection vector does not vectorise. SIMD operates on multiple elements simultaneously, but when multiple entries satisfy the predicate, it becomes impossible to update the vector with their indices. Each time an index is added to the selection vector, its size must be incremented to accommodate the next value at the tail of the array, creating a dependency. While retrieving the data that the selection vector points to could be done in SIMD registers by using Gather/Scatter instructions, these instructions are known to be slow [36].

4.2.2 Selection Byte Map

This method creates a byte map, where each byte represents whether the corresponding value in the data vector satisfies the predicate (with 1 indicating satisfaction and 0 otherwise). For a data element at index i, the predicate result is recorded in the i-th byte of the byte map. Unlike the selection vector, this approach avoids dependencies, allowing SIMD parallelism, which is why [13] use it to implement predicate pushdown in Parquet. However, since the byte map only needs to store 0 or 1, it effectively uses just a single bit per byte, resulting in the underutilisation of memory space.

4.2.3 Selection Bitmap

In this approach, each input vector element is associated with a bit in a bitmap. This method is identical to the selection byte map but packs the values together, using eight times less space. It retains the benefits of the byte map, but packing the bits may introduce some overhead.

Another advantage of the bitmap is that merging the selection of two columns is a simple AND operation over 1,024 bits. This is not possible with a selection vector at all, while a selection byte map would require eight times more computation, needing to AND 8,192 bits as each byte is the result of a single evaluation.

4.2.4 Reverse Selection Vector

Recall that selection vectors are efficient when only a few elements satisfy the predicate. Now, imagine that most elements are selected. In this case, it would be more efficient to record the unselected elements instead. For example, if we have a data vector $[0, 1, \ldots, 1023]$ and the filter operation is $x > 0$, the selection vector would store $[1, 2, \ldots, 1023]$, while the reverse selection vector would only store $[0]$, the index of the unselected element. It shares strengths and weaknesses with the selection vector but is more space-efficient for over 50% selectivity.

4.2.5 Adaptive Approach

While we can benchmark whether a bitmap or byte map is a more efficient selection data structure, the map approaches and the [reverse] selection vector each perform better in different scenarios. The map methods are generally effective, while the latter excels in low selectivity. Therefore, we propose an adaptive approach where the appropriate data structure is chosen based on the context [31]. We provide more details about this in Section 5.3.

4.3 Dictionary Optimisation

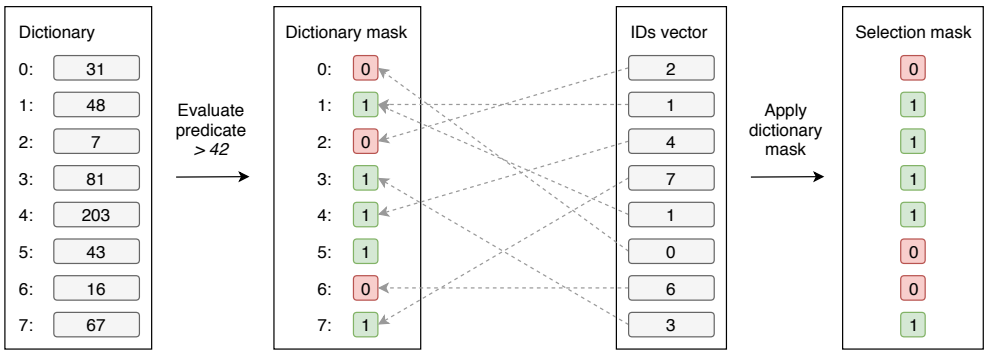


Figure 4.2: Example of how the dictionary optimisation works [13].

Suppose we have data encoded in DICT. The simplest way to perform predicate evaluation is to decode the data and evaluate the predicate on the unencoded values. [13] propose the following optimisation. Since the dictionary contains all the values present in the data without repetition, we can run predicate evaluation directly on the dictionary to produce a *dictionary mask*: a data structure where at index i, we have 1 if the value at index i in the dictionary satisfies the predicate, and 0 otherwise. Then, we map the index vector to the dictionary mask instead of the dictionary itself. The benefit of this approach is that when the dictionary is small, we evaluate fewer values. The downside is that since we do not know the size of the dictionary at compile time, the predicate evaluation of the dictionary does not vectorise.

4.4 Other Optimisations

4.4.1 Min/Max Skipping

Min/Max Skipping is an optimisation technique that uses min/max metadata to avoid unnecessary predicate evaluations. For example, consider a column containing numbers up to 10 and a filter condition $x = 11$. By checking the maximum value of the column in metadata, we can immediately determine that no values satisfy the predicate without evaluating each element [13].

Typically, min/max statistics is maintained at the granularity of data blocks. If it indicates that no values satisfy the predicate, evaluation of the entire block can be skipped. This technique is highly effective when data is well-blocked, meaning similar data is grouped within the same chunk. Effective blocking reduces the need for predicate evaluations, saving compute time and I/O operations [16]. However, implementing min/max skipping becomes challenging when multiple columns are involved, as a single-column sort key is less effective. [15] demonstrate that by recording user queries in Amazon Redshift and creating a list of expected filters, data can be partitioned based on their eligibility for these filters. This approach can achieve up to an 85% reduction in end-to-end workload runtime and up to 100x faster performance on individual queries.

Our work focuses on predicate evaluation at the granularity of a single vector (1,024 elements). Currently, FastLanes does not record min/max statistics at this level, meaning this technique is irrelevant to our work.

4.4.2 Pruning Optimisation

Pruning optimisation reduces the amount of data processed by eliminating elements that do not meet certain predicate criteria early in the evaluation process. Two notable methods in this area are BitWeaving [26] and ByteSlice [18]. These approaches reorganise the bits of multiple values in a radix-like fashion: corresponding bit positions across different values are stored contiguously in memory. This layout enables fast predicate evaluations by SIMD parallelism.

BitWeaving introduces two distinct bit-packed layouts: horizontal and vertical. The first is optimised for efficient lookup operations and does not benefit from pruning optimisation, unlike the second layout, which has fast filter scanning at the expense of slower lookups. ByteSlice builds on these concepts by applying bit-level separation on a byte-by-byte basis rather than bit-by-bit, achieving a balance between fast lookups and efficient filter scans.

For instance, consider vertical bit-packing, which stores the first bits of multiple values followed by the second bits. If we apply a filter such as $x = 256$ on `uint8_t` data and observe that all first bits are 0, we can immediately conclude that none of the values satisfy the predicate. This allows us to skip evaluating the remaining bits, as it is evident that all numbers are below 256. This short-circuiting reduces the time needed to evaluate predicates by avoiding unnecessary bit evaluations. However, both BitWeaving and ByteSlice do not provide efficient decompression mechanisms. That is why FastLanes does not organise data in such a way, making pruning optimisation impossible.

Part II

Research

Chapter 5

Predicate Evaluation on Uncompressed Data

Efficient predicate evaluation is critical in database systems, especially with large datasets. A fundamental question arises: *can we develop data-parallel predicate evaluation methods that are both fast and portable across various hardware architectures without resorting to multiple platform-specific implementations?* Additionally, we must know *which selection data structure – selection bitmap, selection byte map, selection vector, or reverse selection vector – is most suitable for efficient predicate evaluation, considering factors such as data parallelism, selectivity, and performance trade-offs.* This chapter explores these research questions by developing fully data-parallel predicate evaluation algorithms auto-vectorisable by the compiler to utilise the widest available SIMD registers. While our work is within the context of FastLanes, a file format that employs cascaded encoding, the methods we develop apply to decompressed data beyond this system.

Evaluating predicates on data compressed with cascaded encoding presents three options. First, we can *fully decompress* the data before applying the predicate. This approach, while straightforward, may not be performance-efficient due to large decompression overhead [25]. However, this could suit FastLanes, where decoding is fast. The second option is to *partially decompress* the data. For multi-layer cascaded encodings, this involves decoding some, but not all, cascade layers, avoiding the overhead of complete decompression. The third option is to *evaluate predicates directly on the compressed data*, which requires specialised algorithms for each encoding (currently, FastLanes has 39) and is left for future work. Data-parallel predicate pushdown was explored before, but with limited efficiency [13], restricted hardware support [25], or applied to data stored in layouts tailored for

predicate pushdown [18, 26]. Thus, our goal is to develop predicate evaluation algorithms that are efficient and portable across different hardware architectures without relying on specialised data layouts or platform-specific implementations.

We structure this chapter as follows. In Section 5.1, we describe different data-parallel predicate evaluation algorithms when the selection data structure is a bitmap and benchmark them. We compare the results to the algorithm that uses the selection byte map [13]. In Section 5.2, we explain how to obtain a single selection bitmap when evaluating multiple predicates or columns and present the main disadvantage of the byte map – the speed of merging multiple selections. Section 5.3 examines the speed of predicate evaluation algorithms that use a selection vector. We observe that with low selectivity, predicate evaluation algorithms that utilise the selection vector are faster than those using a selection bitmap. We explain how to quickly convert a bitmap to a selection vector and benchmark whether switching from a selection bitmap to a selection vector is advantageous when selectivity drops below a certain threshold.

All the algorithms presented use the equality predicate and are easily adaptable to other predicates. The trends we observe in the benchmarks with equality will hold with other predicates, as changing a predicate alters only a few assembly instructions responsible for the comparison, affecting all the algorithms similarly. We explain the benchmark devices and settings in Appendix A and include miscellaneous algorithms in Appendix B. In the next chapter, we focus on predicate evaluation on semi-compressed data, building on the methods we develop here.

5.1 Predicate Evaluation Algorithms

Algorithm 4 The equality evaluation algorithm using selection byte map [13]. The `__restrict` qualifiers indicate that the input vector and output byte map do not overlap in memory, allowing optimised memory access. The template parameter `T` allows the algorithm to support different data types, enabling vector widths from 8 to 64 bits.

```
1 template<typename T>
2 void naive(T* __restrict vec, uint8_t* __restrict bytemap, T
      filter) {
3     for (uint16_t i = 0; i < 1024; i++) {
4         bytemap[i] = vec[i] == filter;
5     }
6 }
```

Our goal is to evaluate predicates on vectors containing 1,024 values using data parallelism on SIMD and record the result in a bitmap. The data can consist of 8, 16, 32, or 64-bit unsigned integers. Even though the algorithms differ, their core idea remains the same: load a batch of data into the SIMD register, compare the values against the filter in each lane, and generate a mask of 0s and 1s based on whether each value satisfies the predicate. Finally, the mask needs to be bit-packed to produce a bitmap.

Since the query execution engine ultimately receives a selection vector constructed from the selection bitmap, the exact order of values within the bitmap is not critical as long as we have a fast method to convert the bitmap into a selection vector, which we describe in Section 5.3. We leverage this flexibility to choose a bitmap order ideally suited for data-parallel evaluations and develop a method that quickly transforms it into a selection vector. However, the order must remain consistent across all data widths to allow us to merge the bitmaps from columns containing different data types. It is important to note that bitmaps consisting of 128x `uint8_t`, 64x `uint16_t`, 32x `uint32_t`, or 16x `uint64_t` elements are all equivalent as they all represent 1,024 bits.

Our bitmap algorithms will be compared to Algorithm 4 used for predicate pushdown in Parquet [13]. It stores each result in a byte rather than a bit.

5.1.1 Scalar: raw

The *raw* algorithm partitions the 1,024-element vector into eight segments, as indicated by `128*0, ..., 128*7` in Algorithm 5. In scalar execution, each iteration processes one element from each segment. However, for 8-bit data when the algorithm is vectorised using a virtual 1,024-bit SIMD register, the `src = *(vec + 128 * 0 + i)` loads 128 consecutive 8-bit values starting from position 0. This pattern repeats at positions `128, 256, ..., 896`, allowing the processing of 128 values per iteration.

The bitmap stores results in an interleaved order: the first vector's results are at indices `0, 8, 16, ..., 1016`, the second at `1, 9, 17, ..., 1017`, and so on, as shown in Figure 5.1. Consequently, the bitmap represents predicate evaluation results in the sequence: `0, 128, 256, 384, 512, 640, 768, 896, 1, 129, 257, 385, 513, 641, 769, 897, ..., 895, 1023`. This layout allows data-parallel predicate evaluation.

However, the algorithm is more complex with wider data types. For example, with `uint16_t`, only 64 values from a segment can fit into the 1,024-bit SIMD register, thus dividing each segment into two blocks. As a result, a second pass

Algorithm 5 The *raw* algorithm for equality evaluation. `vec` is the input vector containing data to be evaluated, `filter` is the value used for equality testing, `bitmap` is the array where the results are stored, and `sizeof(T)` is the size of the type `T` in bytes.

```
template<typename T>
void raw(T* __restrict vec, uint8_t* __restrict bitmap, T
   filter) {

    constexpr int lanes = 128 / sizeof(T);
    uint8_t  tmp {0}, equality_result {0};
    T src;

    for (int j = 0; j < sizeof(T); j++) {
        for (int i = j * lanes; i < (j + 1) * lanes; i++) {
            src             = *(vec + 128 * 0 + i);
            equality_result = (src == filter);
            tmp             = equality_result;

            src             = *(vec + 128 * 1 + i);
            equality_result = (src == filter);
            tmp |= equality_result << 1;

            src             = *(vec + 128 * 2 + i);
            equality_result = (src == filter);
            tmp |= equality_result << 2;

                              ...

            src             = *(vec + 128 * 7 + i);
            equality_result = (src == filter);
            tmp |= equality_result << 7;

            *(bitmap + i) = tmp;
        }
    }
}
```

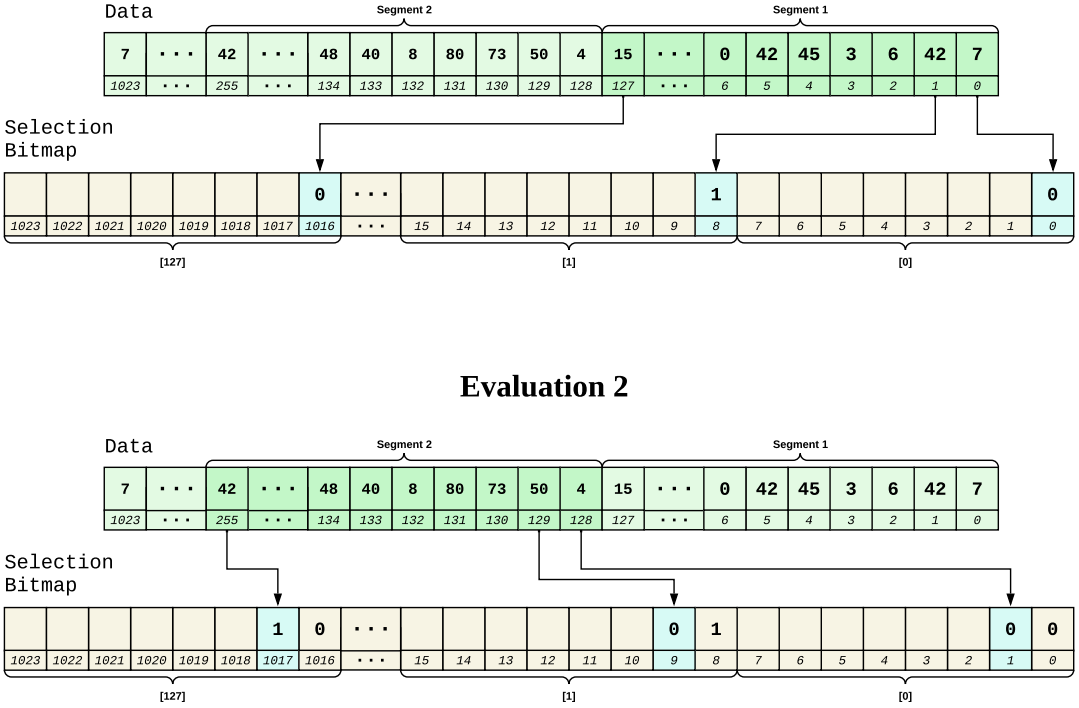


Figure 5.1: Data-parallel evaluation of "= 42" on 8-bit data using a selection bitmap. Evaluation 1 demonstrates the first SIMD evaluation acting on the first segment of the data and how the result maps to the bitmap. Evaluation 2 shows the same but with the next segment. Each segment consists of 128 consecutive values.

through all segments is required to handle the remaining values, as illustrated in line 8 of the algorithm. This limitation arises because we cannot process each block chronologically: even-indexed blocks affect `bitmap[i]`, while odd-indexed blocks affect `bitmap[i + 64]`. This non-uniformity is bad for SIMD execution. Thus we process all first blocks across segments, then all second blocks.

Additionally, with 16-bit data, evaluation occurs in 16-bit lanes, but the results are stored in an 8-bit bitmap. Such misalignment hinders performance. Ideally, for 16-bit data, the vector should be divided into 16 segments with 64 elements instead of 8 segments with 128 elements. The order of results in the bitmap would then follow a pattern of `0, 64, 128, ..., 960, 1, 65, ..., 961, ..., 1023`. This allows processing every segment in chronological order.

However, this method produces a separate bitmap layout for each data width.

Maintaining the same bitmap layout across all data types allows us to merge the bitmaps from different columns without the overhead of converting between different orders. Since FastLanes utilises compressed execution, we expect most evaluations to involve narrow data types. Thus, we choose the optimal order for `uint8_t` data. In the next chapter, we will explore predicate evaluation on DICT without decompression, allowing us to evaluate `uint8_t` or `uint16_t` instead of wider data types.

This design is future-proof, as it anticipates the availability of 1,024-bit SIMD registers. The implementation is also backwards-compatible: for existing SIMD widths, such as 512-bit AVX-512 and 256-bit AVX2, the compiler will load 64 and 32 values, and for 128-bit Neon or SSE, it will load 16 values per iteration. Thus, it utilises the widest available SIMD registers without requiring architecture-specific code.

5.1.2 Scalar: native

Algorithm 6 The *native* algorithm for equality evaluation.

```
template <typename T>
void native(T* __restrict vec, T* __restrict bitmap, T
   filter) {
    for (size_t i = 0; i < 128; i++) {
        for (size_t j = 0; j < 8; j++) {
            bitmap[i] |= static_cast<T>((vec[i + j * 128] ==
                filter)) << j;
        }
    }
}
```

The *native* algorithm aligns the bitmap's width with the data width, avoiding situations where the bitmap occupies only part of the SIMD lane. However, for all data widths other than `uint8_t`, only the rightmost 8 bits of each `bitmap[i]` are filled, while the remaining bits on the left remain zero. If this method proves faster than others, it could be used for single-column predicates where merging selections is not required.

5.1.3 Scalar: mem

Algorithm 7 The *mem* algorithm for equality evaluation. The *unaligned mem* algorithm is similar, but on line 3, it uses `uint8_t` instead of `T`, and omits the `static_cast` on lines 7 and 12.

```
1  template <typename T>
2  void mem(T* __restrict vec, uint8_t* __restrict bitmap, T
       filter) {
3      T res[128] = {0};
4
5      for (size_t i = 0; i < 128; i++) {
6          for (size_t j = 0; j < 8; j++) {
7              res[i] |= static_cast<T>((vec[i + j * 128] ==
                   filter)) << j;
8          }
9      }
10
11     for (size_t i = 0; i < 128; i++) {
12         bitmap[i] = static_cast<uint8_t>(res[i]);
13     }
14 }
```

The *mem* algorithm follows the same principle as the *native* approach: the bitmap occupies the entire width of the lane. However, *mem* algorithm stores the result with trailing zeros in a temporary array and later removes them by casting each bitmap block from the width of the data down to 8 bits.

To see if such alignment is beneficial, we have also developed an unaligned version of this algorithm, where SIMD computations are not driven by the data width and interact with a bitmap represented by `uint8_t` elements.

5.1.4 Scalar: or

The idea behind the *or* algorithm is similar to the aligned *mem* approach. However, instead of downcasting the temporary bitmap to remove trailing zeros, we bit-pack the useful values before writing them to the final bitmap. Bit-packing is written in an auto-vectorisable way to maximise performance.

Algorithm 8 The *or* algorithm for equality evaluation.

```
template <typename T>
void or(T* __restrict vec, T* __restrict bitmap, T filter) {
 T res[128] = {0};

 for (size_t i = 0; i < 128; i++) {
  for (size_t j = 0; j < 8; j++) {
   res[i] |= static_cast<T>((vec[i+j*128] == filter)) << j;
  }
 }

 for (size_t j = 0; j < 128 / sizeof(T); j++) {
  T tmp = 0;
  for (size_t i = 0; i < sizeof(T); i++) {
   tmp |= res[i + j * sizeof(T)] << (i * 8);
  }
  bitmap[j] = tmp;
 }
}
```

5.1.5 Intrinsics

We implemented two algorithms using AVX-512 intrinsics to assess potential performance gains over scalar auto-vectorised code. The overall idea is similar to the *raw* algorithm: load consecutive data from each of the eight segments and compare it with the filter to produce a mask. For `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`, the masks contain 64, 32, 16, and 8 values. Next, we map the results from each block into the bitmap layout described earlier. For this, we generate a vector where each mask element is assigned a specific value (e.g., `0x01`, `0x02`, ..., `0x80`) if the mask is active, and `0x00` otherwise. This process is repeated for each segment, with each segment setting its respective bit (segment 1 sets `0x01`, segment 2 sets `0x02`, and so on, up to segment 8 setting `0x80`).

We combine the results using bitwise `OR` operations. For 8-bit data, the combined result represents which elements satisfy the predicate and we can store the result and proceed to the next iteration, as seen in Algorithm 12. However, with wider data types, only the first 8 bits of each lane are useful. For example, with `uint32_t`, at the end of the iteration, each lane will contain 8 bits of the actual results followed by 24 trailing zeros. We need to eliminate them.

Algorithm 9 The AVX-512 *intrinsics* algorithm to evaluate "=" on 32-bit data.

```
1  void avx512_32(uint32_t* __restrict vec, uint8_t* __restrict
       bitmap, uint32_t filter) {
2
3   __m512i fltr_vec = _mm512_set1_epi32(*reinterpret_cast<
       int32_t*>(&filter));
4
5   for (int j = 0; j < 4; j++) {
6    for (int i = j * 32; i < (j + 1) * 32; i += 16) {
7     __m512i src0 = _mm512_loadu_si512(vec + 128 * 0 + i);
8     __m512i src1 = _mm512_loadu_si512(vec + 128 * 1 + i);
                                  ...
14    __m512i src7 = _mm512_loadu_si512(vec + 128 * 7 + i);
15    __mmask16 cmp0 = _mm512_cmpeq_epi32_mask(src0, fltr_vec);
16    __mmask16 cmp1 = _mm512_cmpeq_epi32_mask(src1, fltr_vec);
                                  ...
23    __mmask16 cmp7 = _mm512_cmpeq_epi32_mask(src7, fltr_vec);
24
25    __m512i result = _mm512_maskz_set1_epi32(cmp0, 0x01) |
26                     _mm512_maskz_set1_epi32(cmp1, 0x02) |
27                     _mm512_maskz_set1_epi32(cmp2, 0x04) |
28                     _mm512_maskz_set1_epi32(cmp3, 0x08) |
29                     _mm512_maskz_set1_epi32(cmp4, 0x10) |
30                     _mm512_maskz_set1_epi32(cmp5, 0x20) |
31                     _mm512_maskz_set1_epi32(cmp6, 0x40) |
32                     _mm512_maskz_set1_epi32(cmp7, 0x80);
33
34    // Option 1: truncate and save
35    __m128i packed_result = _mm512_cvtepi32_epi8(result);
36    _mm_mask_storeu_epi8(bitmap + i, 0xFFFF, packed_result);
37
38    // Option 2: compress store
39    // __mmask64 mask = 0x1111111111111111ULL;
40    // _mm512_mask_compressstoreu_epi8((void*)(bitmap + i),
         mask, result);
41   }
42  }
43 }
```

Thus, algorithms that evaluate `uint16_t` – `uint64_t` data also include steps to remove the zeroes. For this, we employ two different strategies. In the first one, we use truncation to downsize the lanes to 8 bits, effectively moving from the 512-bit `zmm` register to a 128-bit `xmm` register. Since the trailing zeros are at the end of each lane, this achieves the desired effect, and the resulting lanes only contain the data. The second variant involves saving only the bits we want in memory using compress store instructions. Specifically, we want to store the first 8 bits from each lane. On AVX-512, there are two such instructions for integers: `VPCOMPRESSD` and `VPCOMPRESSB`. The former allows us to select which 4-byte blocks to store – this is too large for our purposes. The latter provides finer granularity, allowing us to select a byte. Since we want to save the first 8 bits from each lane, we use `VPCOMPRESSB`, passing a mask that selects the lane's first byte.

5.1.6 Results

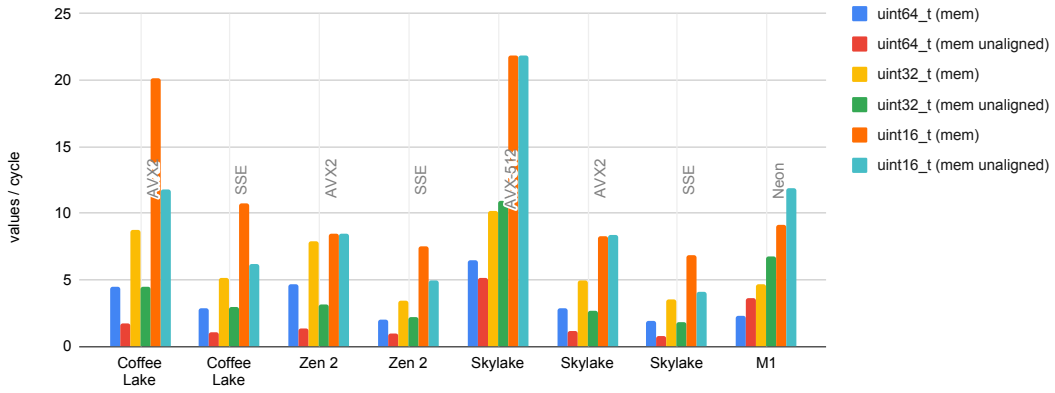
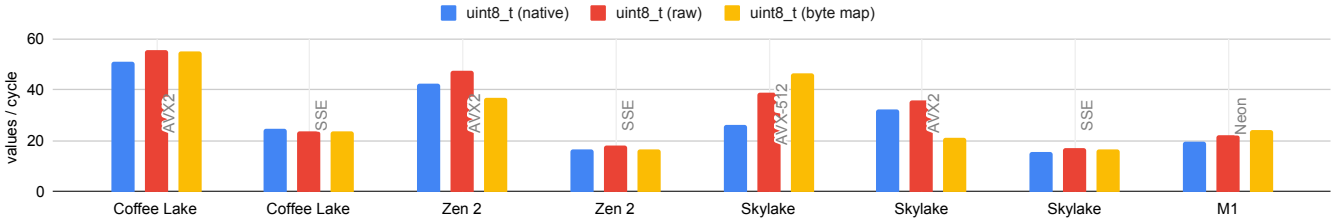


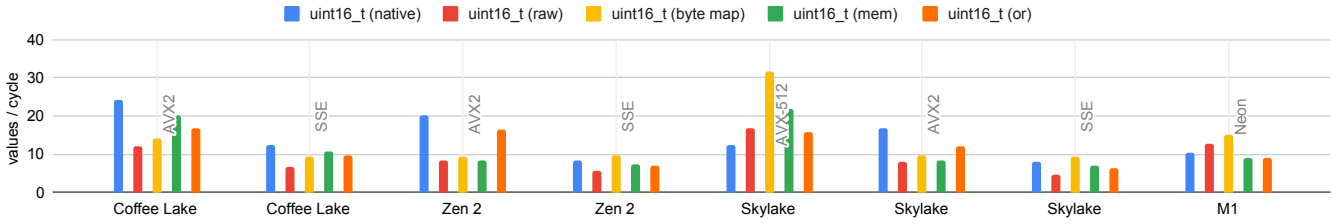
Figure 5.2: Speed comparison of *mem* and *unaligned mem* algorithms.

Firstly, we conducted microbenchmarks to compare the performance of the two variants of *mem* algorithm. As shown in Figure 5.2, the aligned version consistently outperforms the unaligned version on SSE, AVX2, and AVX-512 ISAs. Conversely, the unaligned version is faster across all data widths on Neon. Thus, the former is more robust, and in the subsequent benchmarks, we only present the results of the aligned *mem* algorithm.

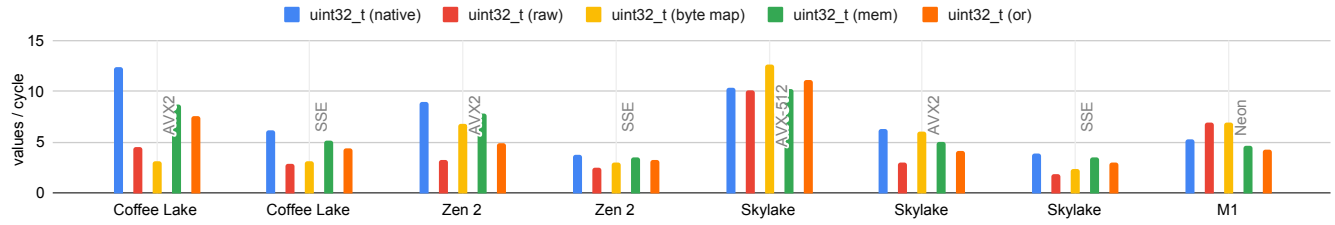
Figure 5.3 presents the benchmark results for all algorithms across different data widths. As expected, narrower data types result in faster predicate evaluations.



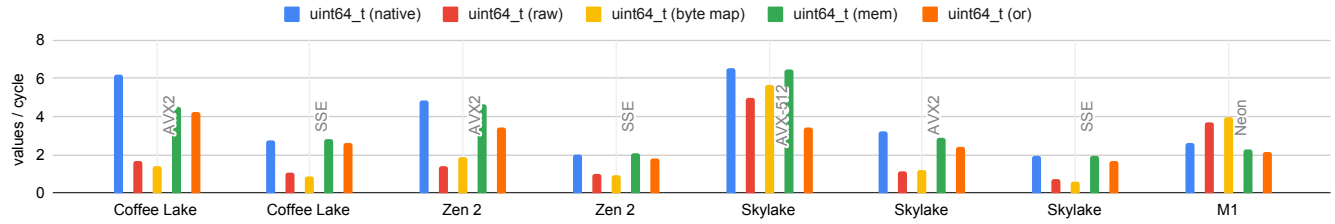
(a) `uint8_t`



(b) `uint16_t`



(c) `uint32_t`



(d) `uint64_t`

Figure 5.3: The speed of predicate evaluation algorithms across different microarchitectures and ISAs. Each subfigure presents results on a distinct data type.

When using `uint8_t` data, all algorithms perform similarly on SSE. On AVX2, the *raw* algorithm shows a clear advantage over the *native* algorithm, with the byte map approach trailing in third place. On AVX-512 and Neon, the byte map method is the fastest, followed by the *raw* algorithm and then the *native* algorithm. However, on Neon, the speed differences between these algorithms are relatively small. For wider data types, the performance varies depending on the architecture and the ISA in use. On Neon, the *raw* and byte map algorithms perform exceptionally well, the latter being slightly faster. On SSE and AVX2, the *native* algorithm is the fastest in most cases.

Since we aim for portability and future-proofness, our recommendations focus on the most advanced ISAs: AVX-512 and Neon. Among the bitmap algorithms, the *raw* algorithm is consistently robust across these architectures. Therefore, it is our algorithm of choice. While the byte map approach demonstrates strong performance, it consumes more space and requires significantly more time to merge selections of multiple columns – a trade-off we explore further in Section 5.2.

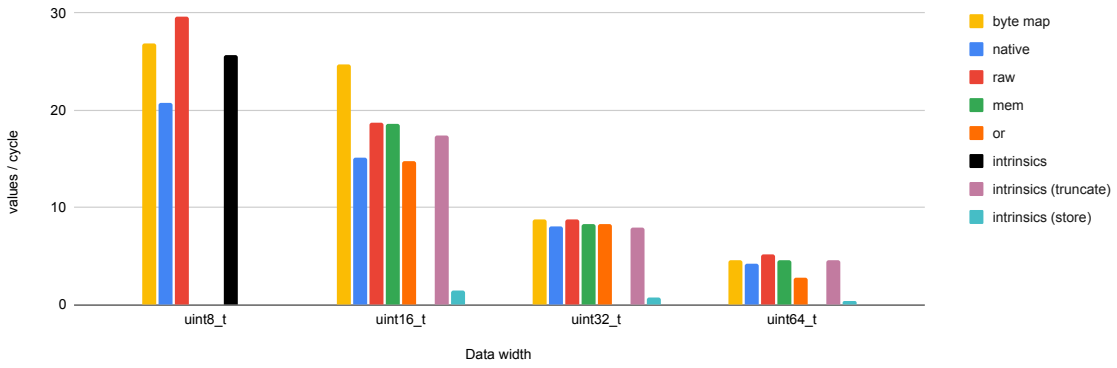


Figure 5.4: Performance comparison of various predicate evaluation algorithms, including *intrinsics* algorithms, across different data widths on AVX-512.

Regarding the *intrinsics* algorithm, as shown in Figure 5.4, it performs slightly worse than the *raw* algorithm on 8-bit data. For other data widths, the *intrinsics* algorithm using truncation offers performance still slightly inferior to the *raw* algorithm. In contrast, the *intrinsics* algorithm that employs compress store instructions is significantly slower than all other methods. We suspect this is due to the high latency of the compress store instruction [4]. Therefore, we recommend opting for the more portable solution – the auto-vectorised *raw* algorithm.

5.2 Merge optimisation

Algorithm 10 Bitwise AND of two 1024-bit bitmaps. The result is stored in `main` bitmap.

```
template<typename T>
void bitmap_and(T* __restrict main, T* __restrict other) {
    constexpr size_t N = 128 / sizeof(T);
    for (size_t i = 0; i < N; ++i) {
        main[i] &= other[i];
    }
}
```

A selective query may contain multiple conditions on the same or across different columns. These conditions could be combined using conjunctions, disjunctions, or negations. Our goal is to support all these operations effectively. Supporting negation is straightforward: evaluate everything within the `NOT (...)` expression and flip all the bits in the resulting bitmap. For conjunctions like `A AND B`, we evaluate both predicates separately and then apply the bitwise `AND` operation to merge the two bitmaps. Disjunctions are handled using negation and conjunction operations, leveraging De Morgan's law, which allows us to transform `A OR B` into `NOT (NOT A AND NOT B)` [25].

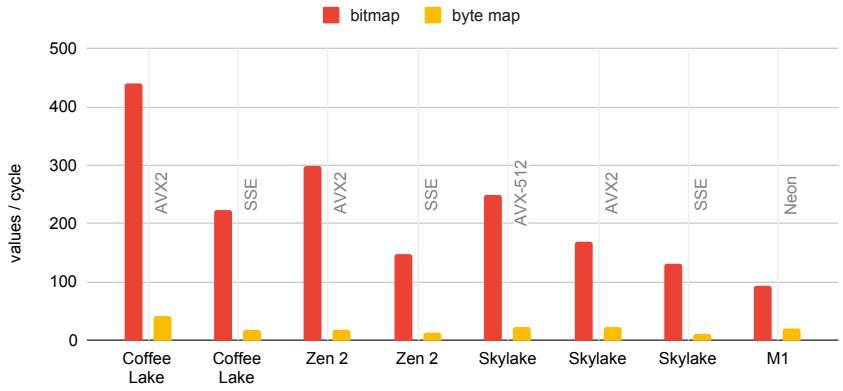


Figure 5.5: Speed comparison of merging selection bitmap against byte map across various [micro]architectures and ISAs.

The implementation of `AND` of two bitmaps is trivial and can be seen in Algorithm 10. Since 1,024 bits can be represented as 128x `uint8_t`, 64x `uint16_t`, 32x `uint32_t`, or 16x `uint64_t` elements (and we can switch between these representations in C++ with no overhead), we can have four versions of the algorithm. Figure 5.5 shows that merging selection bitmap is much faster than byte map. On AVX-512 and Neon, the bitmap merging is 10.5x and 4.6x faster, respectively.

Although the byte map algorithm for predicate evaluation is often faster than the bitmap approach, the significant difference in the speed of merging two selections means that as we evaluate more columns, the latter becomes faster (when combining the speed of vector evaluation and result merging). We calculated the minimum number of columns from which the evaluation is faster with the bitmap and quantified the speedup, presenting them in Tables 5.1 and 5.2, respectively.

Table 5.1: Minimum number of columns where bitmap evaluation outperforms byte map evaluation for different data widths in bits on AVX-512 and Neon ISAs.

Data Width	**AVX-512**	**Neon**
8	2	2
16	4	2
32	3	2
64	3	2

Table 5.2: X-fold speedup achieved by bitmap over byte map evaluation for different data widths in bits on AVX-512 and Neon ISAs, with performance gains highlighted in grey.

(a) AVX-512 Performance Boost

Columns	**Data Width**			
	8	16	32	64
2	1.53	0.86	0.99	0.98
4	1.85	1.01	1.08	1.02
8	2.00	1.09	1.13	1.05
16	2.07	1.13	1.15	1.06

(b) Neon Performance Boost

Columns	**Data Width**			
	8	16	32	64
2	1.29	1.08	1.12	1.01
4	1.45	1.18	1.19	1.04
8	1.53	1.24	1.22	1.06
16	1.57	1.26	1.23	1.07

5.3 Low Selectivity Optimisation

As we mentioned in Section 4, the selection vector is likely fast for predicate evaluation at low selectivities, whereas the selection bitmap offers consistently good performance. Recognising that no single method is universally superior, we propose an adaptive approach that dynamically chooses the most efficient representation based on the current selectivity [31]. Consider a scenario where we evaluate a predicate on two columns. In the first column, only one element is selected. If we continue evaluating the second column using a bitmap, we disregard this information and evaluate all the values, leading to avoidable computation. However, when using a selection vector, we can focus exclusively on the selected index, knowing that only one element can meet the condition. Motivated by this, we propose the following procedure:

1. Initialise an empty bitmap to store the results of the predicate evaluations.
2. Perform predicate evaluation using the raw algorithm.
3. After evaluating N columns, compute the selectivity of the bitmap by utilising the `popcnt` instruction to count the number of set bits.
4. If the calculated selectivity falls below M, convert the bitmap to a selection vector for subsequent evaluations. Otherwise, return to step 3.

The N (number of evaluations after which to measure selectivity) can be determined at run-time using micro adaptivity [31]. Since the rate at which selectivity decreases after each evaluation depends on the data and queries, we cannot provide a single value that would be effective in all scenarios.

To efficiently convert a selection bitmap into a selection vector, we use Algorithm 15, which leverages precomputed lookup tables, storing the number of set bits in each possible byte value and the positions of those set bits. By iterating over each byte in the bitmap, we use these tables to quickly identify the indices of elements that satisfy the predicate, avoiding examining each bit individually. As a result, we have a selection vector containing all the indices of selected elements. We benchmarked this algorithm and presented the results in Figure 5.6. The performance is robust across all architectures and ISAs.

As noted above, the point after which we need to measure selectivity depends on the queries and data. However, we can determine at which selectivity it becomes advantageous to convert the bitmap into a selection vector (M). Specifically, we consider a situation where we have measured the bitmap's selectivity,

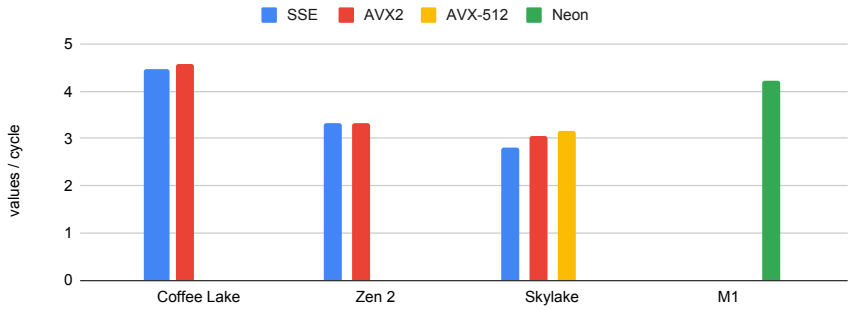


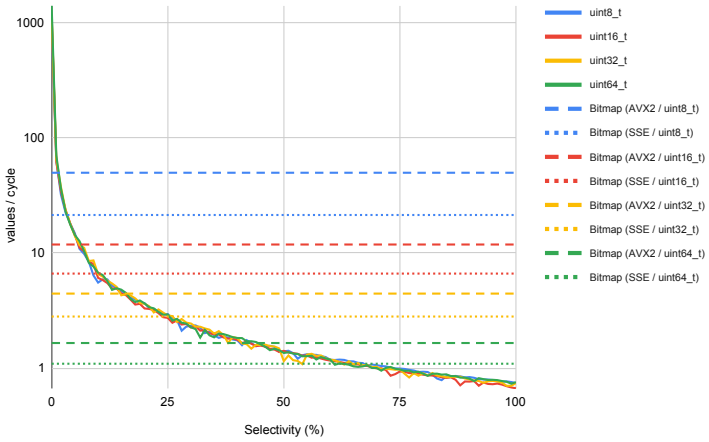
Figure 5.6: The speed of converting a bitmap into a selection vector across various architectures and ISAs.

have additional columns to process, and want to understand whether converting the bitmap into a selection vector will accelerate subsequent evaluations (considering the time required for this conversion). To determine this, we conducted benchmarks where, for each bitmap selectivity, we measured the time taken to convert to a selection vector and perform an equality evaluation, compared to the time taken for a subsequent evaluation on the bitmap without this conversion.

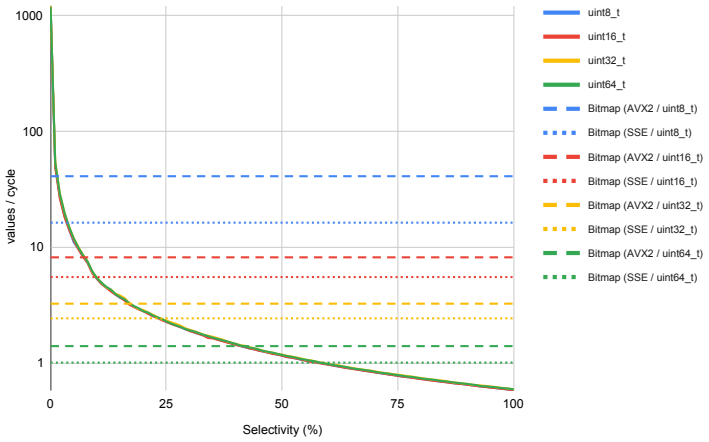
The results are shown in Figure 5.7. As expected, the selection vector outperforms the bitmap at low selectivities. To simplify the analysis, we present Table 5.3, indicating the selectivity thresholds below which converting to a selection vector yields a speedup. Additionally, for AVX-512 and Neon, we made Table 5.4, showing the extent to which conversion and evaluation on the vector is faster than evaluation on the bitmap for all selectivities that offer a performance boost.

As we can see, the wider the data, the higher the threshold at which switching to a selection vector is optimal. This is because evaluating wider data types in SIMD is slower than narrower ones, as seen in Section 5.1.6. Nevertheless, we observe speedups for all data widths, architectures and ISAs. For AVX-512 and Neon, our main targets, a speedup occurs when 9 and 18 or fewer elements are selected. Selecting 18 elements out of 1,024 corresponds to a selectivity of 1.8%. Since such low selectivities will occur rarely and significant speedups are present only when the selectivity is even lower, we decided not to complicate the FastLanes implementation and avoid incorporating this optimisation.

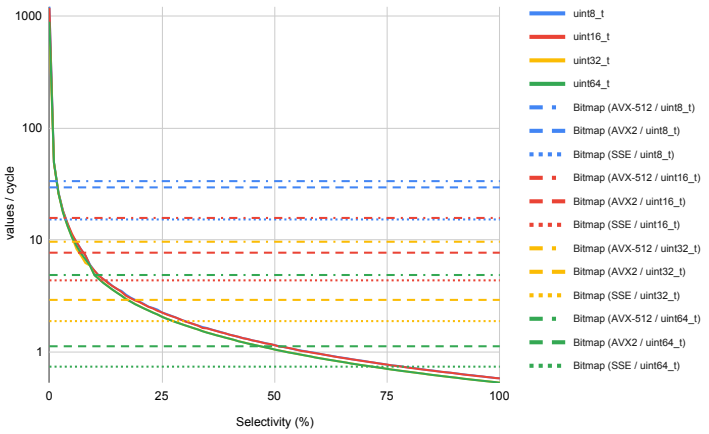
The reverse selection vector is designed to optimise the space efficiency of the regular selection vector for selectivities greater than 50%. However, our findings indicate that the regular selection vector is only a viable option for predicate evaluation at selectivities far below this threshold. Consequently, the reverse selection vector is not suitable for fast predicate evaluation.



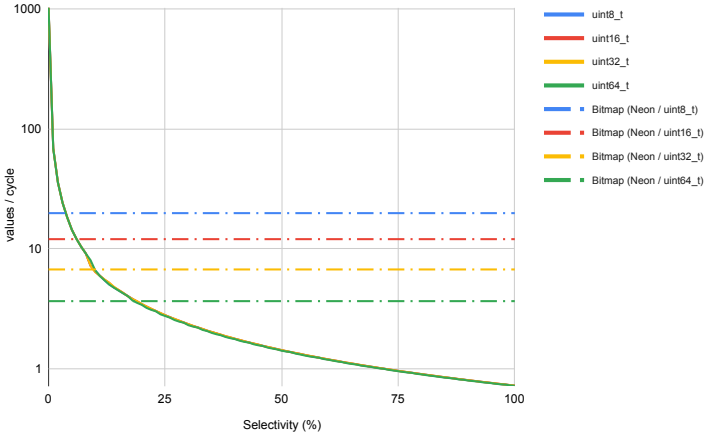
(a) Coffee Lake



(b) Zen 2



(c) Skylake



(d) M1

Figure 5.7: Benchmark results for low selectivity optimisation across different CPU architectures. Solid lines represent the speed of converting to a selection vector and doing a predicate evaluation across various data widths, while the dashed lines show the speed of predicate evaluation on a bitmap without switching.

Table 5.3: Number of selected elements out of 1,024 at or below which converting to a selection vector results in a performance improvement across various architectures, ISAs, and data widths.

Architecture	**ISA**	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`
Coffee Lake	AVX2	1	5	14	44
	SSE	3	9	26	60
Zen 2	AVX2	1	7	17	41
	SSE	3	9	23	58
Skylake	AVX-512	1	3	5	9
	AVX2	1	7	17	46
	SSE	3	12	27	71
M1	Neon	3	6	9	18

Table 5.4: Performance improvement of converting to a selection vector, including the time needed for conversion, over a bitmap across varying data widths and numbers of selected values (out of 1,024) on AVX-512 and Neon ISAs. The values represent the X-fold speedup achieved by the optimised approach, with values indicating an increase in performance highlighted in grey.

(a) AVX-512 Performance Boost

Selected	**Data Width (bits)**			
Values	`64`	`32`	`16`	`8`
1	10.10	5.10	3.09	1.47
2	5.39	2.72	1.67	0.78
3	3.62	1.84	1.13	0.53
4	2.76	1.39	0.85	0.40
5	2.17	1.10	0.68	0.32
6	1.83	0.86	0.56	0.27
7	1.57	0.75	0.49	0.23
8	1.38	0.65	0.39	0.18
9	1.21	0.61	0.38	0.18

(b) Neon Performance Boost

Selected	**Data Width**			
Values	`64`	`32`	`16`	`8`
1	18.71	10.26	5.73	4.36
2	9.76	5.33	2.98	2.26
3	6.60	3.60	2.01	1.53
4	4.99	2.72	1.52	1.15
5	3.93	2.14	1.20	0.91
6	3.30	1.80	1.00	0.76
7	2.84	1.55	0.86	0.66
8	2.49	1.36	0.76	0.58
9	2.20	1.08	0.60	0.46
10	1.81	0.96	0.53	0.42
11	1.62	0.88	0.50	0.38
12	1.49	0.82	0.46	0.34
13	1.38	0.77	0.43	0.32
14	1.30	0.72	0.40	0.30
15	1.23	0.67	0.38	0.28
16	1.16	0.64	0.36	0.27
17	1.10	0.60	0.33	0.25
18	1.01	0.57	0.32	0.24

5.4 Discussion

We presented a bitmap layout that allows data-parallel predicate evaluation and four scalar algorithms that exploit it and auto-vectorise to use the widest SIMD registers available. After benchmarking, we concluded that the *raw* algorithm is the most suitable for FastLanes due to its robust performance with AVX-512 and Neon ISAs. The bitmap order we propose is optimised for 8-bit data. While we could have chosen an order that performs better with other data widths, we anticipate that our algorithm will mainly operate on semi-compressed data with narrow data types. In future work, it would be interesting to explore the other bitmap orders described in Section 5.1. This would involve assessing the performance of algorithms that use them and determining how we can switch orders for it to stay uniform across columns with varying data widths. However, we hypothesise that this approach will be less effective than our current proposal, with the overhead of mapping between bitmap orders being a bottleneck.

We also developed two types of algorithms using AVX-512 intrinsics. Our benchmarks revealed that the performance of the intrinsics algorithm using truncation is comparable to the scalar *raw* algorithm, whereas the version employing compressed store instructions is considerably slower. Consequently, we conclude we are not sacrificing much performance-wise by using portable scalar code for predicate evaluation.

We compared the *raw* algorithm with one that writes results into a byte map and observed that, in many cases, the byte map approach is faster, though the difference is not significant. Next, we discussed handling selective queries with multiple conditions. We proposed evaluating each condition separately and combining the results using `AND` operations on the 1,024 bits. We found that merging byte maps is very slow – to the extent that, when dealing with multiple columns, using a bitmap becomes faster (and the more columns involved, the greater the advantage).

Next, we explored the possibility of transitioning from a selection bitmap to a selection vector when selectivity is low. When evaluating a new column using a selection vector, we can evaluate the values at the previously selected indices, skipping the rest. In contrast, with the selection bitmap, we would need to evaluate the entire column and then merge the old and new selections. We demonstrated a method that converts a selection bitmap to a selection vector and benchmarked it, confirming its high efficiency. We also conducted benchmarks to determine whether switching to a selection vector for evaluation is faster and indeed found that this approach leads to speedup at low selectivities. However,

the speedups start to occur only at marginally low selectivities of 1.8%, with significant gains at even lower values. Since these scenarios are rare, we decided not to complicate the FastLanes implementation by adding this optimisation. The reverse selection vector is designed to optimise the space efficiency of the regular selection vector for selectivities greater than 50%, and our findings indicate that the regular selection vector is only a viable option for predicate evaluation at selectivities far below this threshold. Considering this and the earlier results with the byte map, we concluded that FastLanes should use a bitmap as the selection data structure.

Chapter 6

Predicate Evaluation on Semi-Compressed Data

When performing predicate evaluation on data compressed with cascaded encoding, a fundamental question arises: *How can we efficiently perform predicate evaluation on data compressed with cascaded encoding without fully decompressing the data?* This approach is advantageous because decoding some layers of the cascade is faster than decoding all of them. Encodings that result from such partial decompression but contain sufficient detail for query execution we call *intermediate representations*. An intermediate representation can be any lightweight compression scheme that lacks data dependencies. For example, although we do not consider it in this thesis, FOR can function as an intermediate representation because it allows independent access to each data element. In contrast, DELTA cannot, as it introduces data dependencies.

In this chapter, we explore whether predicate evaluation can be performed more efficiently by decompressing some, but not all, layers of the cascade or if complete decompression followed by evaluation remains the optimal strategy. Our work proves that such optimisations on data compressed with cascaded encoding are feasible.

We focus on DICT, RLE, and their combination because they are widely used [9] and, at the time of writing, were implemented as representations in FastLanes. When decompressing data, FastLanes stops at these semi-compressed states when possible rather than fully decoding. In the previous chapter, we recommended the *raw* algorithm for predicate evaluation on uncompressed data and thus use it as a baseline for predicate evaluation in this chapter.

6.1 Dictionary Encoding

In DICT, the data is represented by a *dictionary array* containing all unique values and an *index vector* that points to these values, as shown in Figure 2.1. Elements in the dictionary can vary in width, up to 64 bits, while the index array uses 8-bit indices if there are 256 or fewer unique values or 16-bit otherwise. If the dictionary array contains 8-bit data, it can have at most 256 unique values, meaning the index array will also contain 8-bit indices. Thus, we note two things: (1) the data in the index array is always of equal or smaller width than that in the dictionary and the decompressed vector, and (2) dictionary encoding is typically applied where the column contains relatively few unique values. With these factors in mind, we propose an optimisation for this encoding. For equality predicate, the optimisation is as follows:

1. If the dictionary contains fewer than N elements, proceed; otherwise, fully decompress the data for standard evaluation.

2. Iterate over the dictionary array to locate the index of the element matching the filter. If no match is found, return an empty bitmap and halt.

3. Perform a standard equality evaluation on the index array, using the index identified in the previous step as the filter.

In step two, we iterate over all elements in the dictionary, which, for large dictionaries, may slow down the process enough that complete decompression before predicate evaluation would be faster. Hence, in step one, we check the dictionary size and only proceed if it is below a threshold N, which we determine in benchmarks. If no value in the dictionary matches the filter in step two, we can safely return a bitmap filled with zeros, which indicates that the value is absent in the data. In step three, once we find the index pointing to the value for which we are evaluating equality, we can directly search for this index in the index array, bypassing decompression. While full decompression would later require evaluation on potentially wide data, this approach allows us to perform predicate evaluation on a `uint8_t` or `uint16_t` index array, which, as established in Section 5.1, is faster.

For the $\neq$ predicate, we can apply the same approach as for $=$, simply inverting all bits in the resulting bitmap. However, $<$, $>$, $\leq$ and $\geq$ present a challenge because multiple entries in the dictionary array might meet the condition, and these entries are not stored in any particular order. This means the indices satisfying the predicate could be dispersed irregularly throughout the dictionary. With a

sorted dictionary array, we could isolate a contiguous range of indices that match the condition and perform a single, efficient range-based evaluation on the index array. With an unsorted dictionary, however, we instead need to conduct a series of equality checks on the index array – one for each dictionary entry that meets the predicate. This approach requires multiple evaluations and the merging of the intermediate bitmaps to evaluate a single predicate, which, given that FastLanes already supports efficient decompression and evaluation, would likely negate any potential gains. Currently, FastLanes supports only unsorted dictionaries. However, optimisation for additional predicates would become feasible once sorted dictionaries are implemented. For this reason, we focus solely on $=$ and $\neq$, where the optimisation provides performance improvements without added complexity, while other predicates remain as future work.

To determine the optimal threshold N for step one and quantify the speedup our optimisation provides, we benchmarked performance across dictionary sizes up to 1,024. We implemented the worst-case scenario, where the last element in the dictionary matches the filter, to ensure that the filter is evaluated against all dictionary elements. In practice, the process is likely to be faster, as a matching element could be found before the dictionary's end, allowing us to proceed to the next step early. Additionally, if the value is absent from the dictionary, we could immediately return an empty bitmap without further evaluating the index array.

Moreover, in our benchmark, we evaluated only a single vector compressed

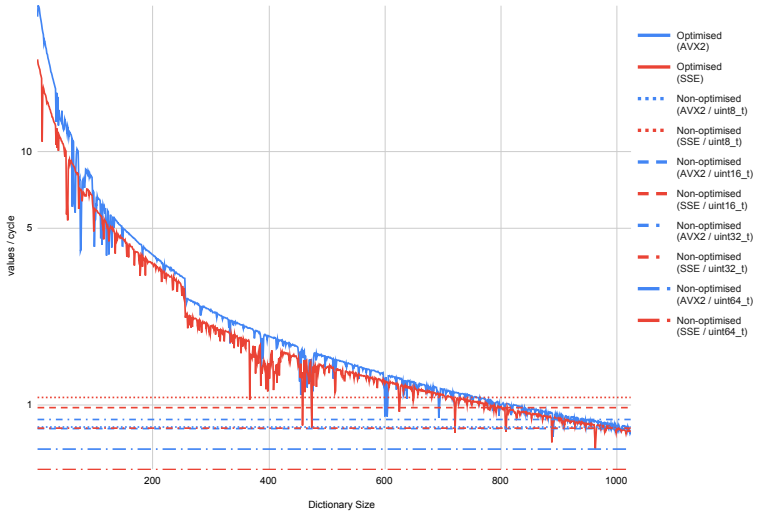
Algorithm 11 Optimised equality evaluation algorithm for the DICT intermediate representation. The `T` and `U` specify the data type of the dictionary and index arrays, respectively. The algorithm identifies the element in the dictionary array that matches the filter value and conducts a predicate evaluation on the index array using this element's index as the new filter.

```
template <typename T, typename U>
void dict_optimisation(U* __restrict idx_arr, T* __restrict
    dict, uint8_t* __restrict bitmap, T filter, U dic_sz) {
  for (U i = 0; i < dic_sz; i++) {
    if (dict[i] == filter) {
      raw(idx_arr, bitmap, i);
      return;
    }
  }
}
```

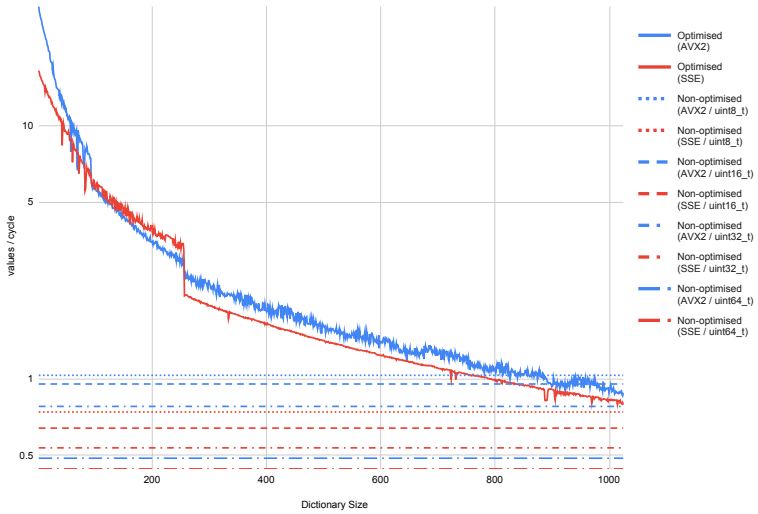
with DICT, whereas in FastLanes, DICT is applied over a row group potentially containing multiple vectors, all sharing the same dictionary array. This means that the cost of evaluating the dictionary is incurred just once for the entire row group. As a result, after evaluating the dictionary, we can perform predicate evaluation on the `uint8_t` or `uint16_t` index arrays of all vectors within the row group rather than on decompressed wide data for each vector individually. This amplifies the performance benefits of our method over full decompression, especially for wider data types, as it reduces the decompression overhead and leverages faster evaluation on narrower data types across multiple vectors.

This optimisation is detailed in Algorithm 11, while benchmark results are depicted in Figure 6.1. When the optimised algorithm line (solid) lies above the non-optimised one (dashed), the optimisation offers a speedup. The intersection point indicates the dictionary size at which both versions perform equally. For convenience, Table 6.1 summarises the dictionary sizes at or below which our algorithm outperforms complete decompression and evaluation – these serve as our recommended values for N.

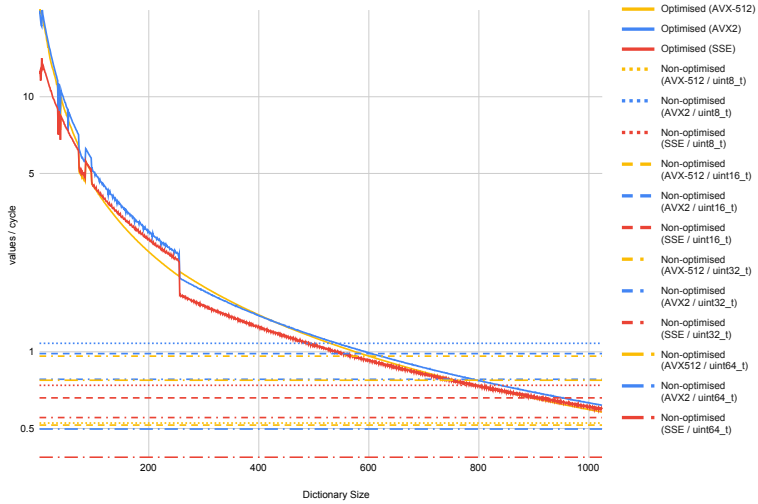
Table 6.1 shows that the threshold N is consistently large, reaching the maximum value tested (1,024) in 15 of the 32 cases. In practical terms, the number of elements in the dictionary may be even larger, given that the dictionary applies to a row group rather than individual data vectors. Table 6.2 illustrates the performance gains achieved by employing this optimisation over full decompression and evaluation for AVX-512 and Neon ISAs. Notably, the speedup remains substantial even with larger dictionary sizes. For instance, when the dictionary comprises 256 elements or fewer, the optimisation provides a speedup of 2x or greater, and when it contains 64 elements or fewer, the speedup surpasses 7x. This optimisation is highly effective, and we recommend avoiding full decompression when N is at or below the established thresholds. Considering DICT encoding is applied when the number of unique elements in the row group is small, resulting in small dictionary arrays, we expect this optimisation to provide a significant performance boost.



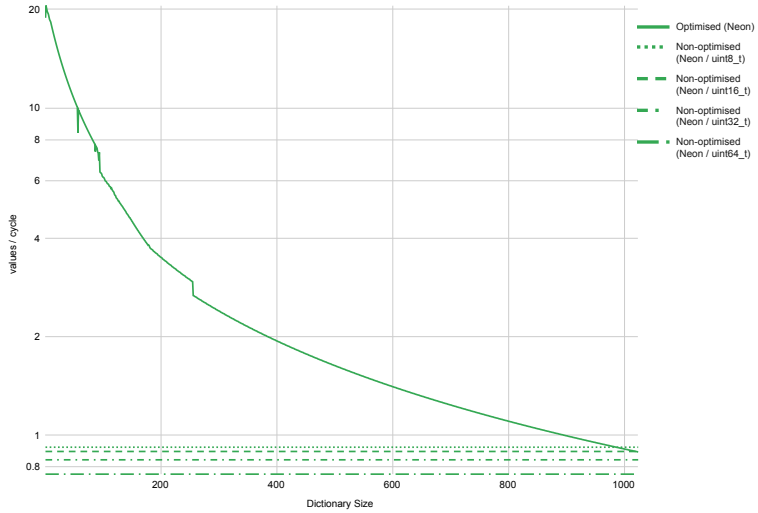
(a) Coffee Lake



(b) Zen 2



(c) Skylake



(d) M1

Figure 6.1: Performance of predicate evaluation on dictionary-encoded data across four microarchitectures. Solid lines indicate the speed of predicate evaluation on compressed data (optimised), while dashed lines illustrate the speed when data is fully decompressed before evaluation (non-optimised).

Table 6.1: Dictionary sizes at or below which doing predicate evaluation without decompressing DICT intermediate representation is faster across various microarchitectures, ISAs, and data widths.

Architecture	**ISA**	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`
Coffee Lake	AVX2	889	889	889	1024
	SSE	366	457	720	1024
Zen 2	AVX2	812	897	1024	1024
	SSE	1024	1024	1024	1024
Skylake	AVX-512	1024	1024	600	756
	AVX2	542	605	787	1024
	SSE	785	901	1024	1024
M1	Neon	985	1020	1024	1024

Table 6.2: Performance gains of predicate evaluation on compressed DICT over full decompression and evaluation across different dictionary sizes and data widths on AVX-512 and Neon ISAs. The values represent the X-fold speedup achieved by the optimised approach, with values indicating an increase in performance highlighted in grey.

Dictionary Size	**AVX-512 Performance Boost**				**Neon Performance Boost**			
	`uint64_t`	`uint32_t`	`uint16_t`	`uint8_t`	`uint64_t`	`uint32_t`	`uint16_t`	`uint8_t`
1	27.23	21.91	40.81	40.07	24.73	22.36	21.09	20.47
64	9.09	7.31	13.63	13.38	12.23	11.05	10.43	10.12
128	4.68	3.77	7.02	6.89	6.81	6.16	5.81	5.64
192	3.39	2.73	5.09	4.99	4.84	4.37	4.12	4.00
256	2.66	2.14	3.99	3.92	3.51	3.17	2.99	2.90
320	2.20	1.77	3.30	3.24	3.01	2.72	2.56	2.49
384	1.87	1.50	2.80	2.75	2.63	2.38	2.24	2.18
448	1.63	1.31	2.44	2.39	2.34	2.11	1.99	1.94
512	1.44	1.16	2.16	2.12	2.10	1.90	1.79	1.74
576	1.29	1.04	1.93	1.90	1.91	1.73	1.63	1.58
640	1.17	0.94	1.76	1.72	1.75	1.59	1.49	1.45
704	1.07	0.86	1.60	1.58	1.62	1.46	1.38	1.34
768	0.99	0.79	1.48	1.45	1.50	1.36	1.28	1.24
832	0.91	0.74	1.37	1.34	1.40	1.27	1.20	1.16
896	0.85	0.69	1.28	1.25	1.31	1.19	1.12	1.09
960	0.80	0.64	1.20	1.17	1.24	1.12	1.05	1.02
1024	0.75	0.60	1.12	1.10	1.17	1.06	1.00	0.97

6.2 Run-Length Encoding

As we noted in Section 3.3.2, the RLE intermediate representation is similar to DICT. In both cases, an index vector points to indices in another array: the *run values array* and *dictionary array* for RLE and DICT, respectively. The difference is that all values in the dictionary array are unique, whereas values in the run values array can repeat if they are non-consecutive. Because of this, we cannot apply the optimisation from the previous section, for the same reason that makes it unsuitable for $<$, $>$, $\leq$ and $\geq$ predicates (i.e. evaluating on the dictionary could yield multiple satisfying elements).

Instead of applying the same optimisation as in the DICT case, we could use an adaptive approach. If the run values array is smaller than some threshold, we could scan it to identify the indices where values satisfy the predicate. If only one such index exists, we have a scenario similar to DICT and can evaluate the narrow index vector searching for that index. If there are a few indices, we can run the equality evaluation once for each index and then combine the resulting bitmaps with a bitwise OR. However, if many values match, this approach requires many evaluations and bitmap merges, which may be slower than simply decompressing the entire RLE representation and evaluating the uncompressed data once.

This strategy involves a trade-off: we must initially scan the run values array to discover the number of matches, incurring overhead even if we end up fully decompressing. To address this, we could train a model that would predict how many run values would be selected [31]. If it predicts many matches, we fully decompress. If it predicts a few matches, we proceed with the partial evaluation and merging steps. We leave this adaptive strategy as a direction for future work.

6.3 Dictionary + Run-Length Encodings

As discussed in Section 3.3.3 and illustrated in Figure 3.2, this intermediate representation can be decompressed into both DICT and RLE intermediate representations through a single flattening operation. Since we have an optimisation for DICT but not for RLE, we compare decompressing to DICT and applying the DICT-optimised evaluation described in the previous section against fully decompressing and applying predicate evaluation on the decompressed data.

The comparison results are shown in Figure 6.2. For ease of analysis, we also created Table 6.3, which indicates the dictionary sizes at or below which evaluation with this optimisation is faster than complete decompression followed by evaluation. Table 6.4 presents the performance improvements achieved by

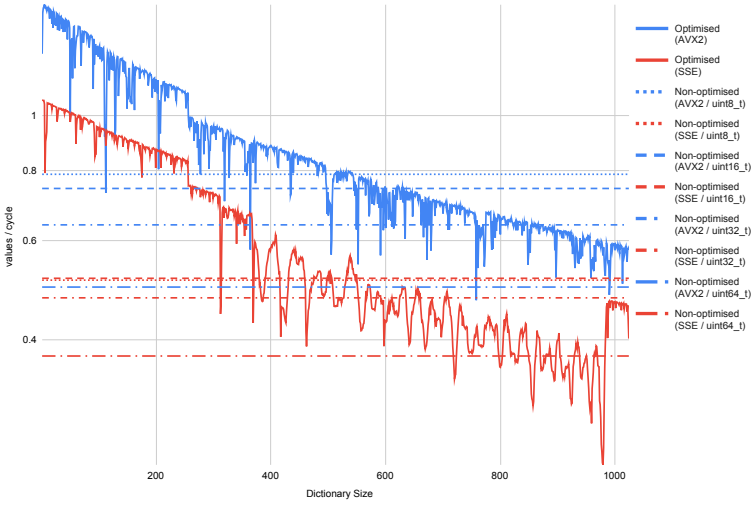
decompressing to DICT and evaluating on semi-compressed data, compared to full decompression and subsequent evaluation on AVX-512 and Neon.

The thresholds for dictionary sizes at which this optimisation provides a speedup are similar to those of DICT on all architectures except Coffee Lake. Meanwhile, in Table 6.4, we see that for AVX-512 and Neon, the speedup is present but more modest than with DICT alone in Table 6.2. Here, the best speedup we can expect is up to 2.15x faster than full decompression.

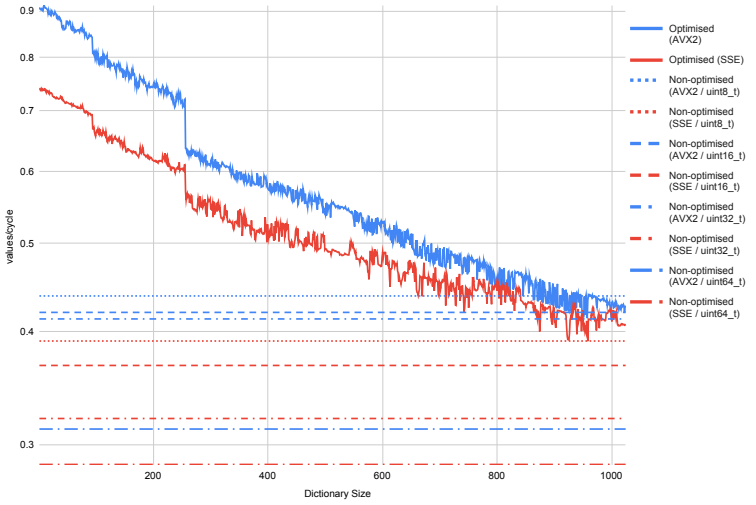
We explain this result as follows. At the beginning of our method, as with full decompression, the data is decoded into the DICT format. Subsequently, everything proceeds as with the DICT intermediate representation: either we use the optimisation from Algorithm 11, or the data is further decompressed and then evaluated. Essentially, it is the same process as in the previous section, but with additional time spent decompressing from DICT + RLE to DICT. This extra decompression step dilutes the speedup, so the results in Tables 6.1 and 6.3 are similar. Regarding Coffee Lake, as we discuss in Appendix A, the device suffers from throttling, leading to volatile results. The speedup for DICT + RLE is more modest than for DICT alone for the same reason: we added the same decompression time layer to both variants, which reduced the relative speedup because

$$\frac{x}{y} < \frac{x+c}{y+c}$$

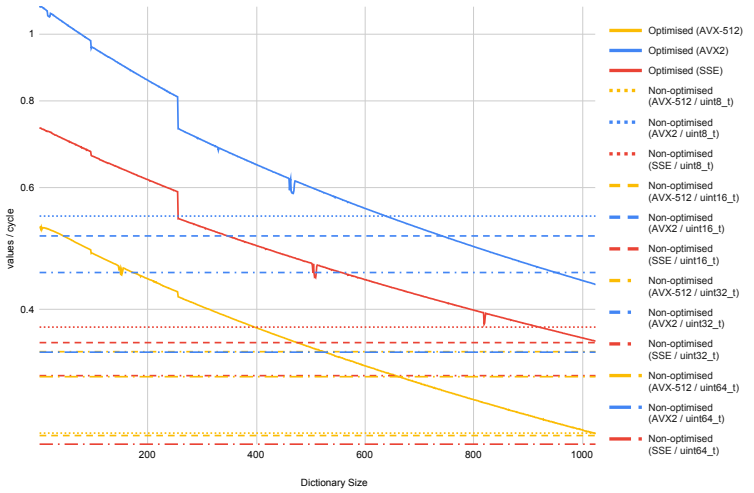
for any positive x, y, and c. In this context, x and y represent the time of optimised evaluation on DICT and evaluation via decompression, respectively, while c is the time taken to decompress from DICT + RLE to DICT. Therefore, we recommend always decompressing DICT + RLE into DICT first and then applying the optimisation for the DICT intermediate representation with the thresholds indicated in the previous section.



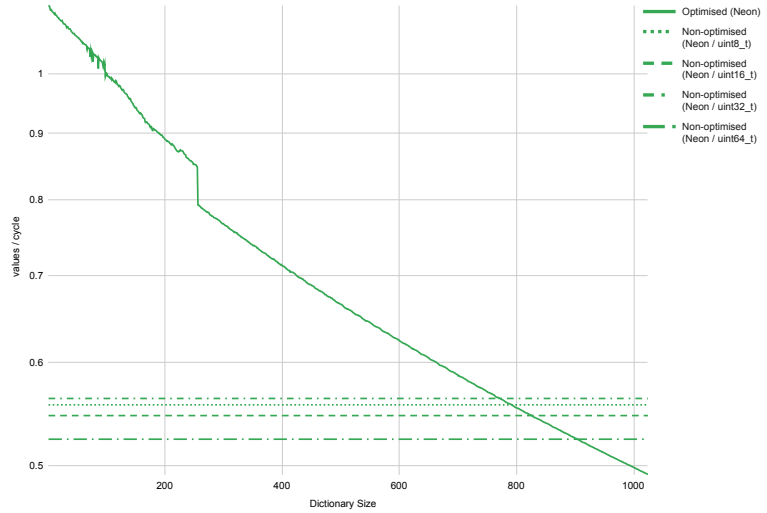
(a) Coffee Lake



(b) Zen 2



(c) Skylake



(d) M1

Figure 6.2: Performance of predicate evaluation on DICT + RLE data across four different microarchitectures. Solid lines represent the speed of predicate evaluation on compressed data (optimised), while dashed lines show the speed when data is decompressed before evaluation (non-optimised).

Table 6.3: Dictionary sizes at or below which doing predicate evaluation on DICT + RLE by decompressing it to DICT intermediate representation and evaluating on it is faster than full decompression with subsequent evaluation across various microarchitectures, ISAs, and data widths.

Architecture	**ISA**	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`
Coffee Lake	AVX2	111	111	363	757
	SSE	311	311	311	718
Zen 2	AVX2	844	947	1024	1024
	SSE	957	1024	1024	1024
Skylake	AVX-512	1020	1024	523	661
	AVX2	638	741	950	1024
	SSE	920	1024	1024	1024
M1	Neon	791	825	771	903

Table 6.4: Performance gains of decompressing DICT + RLE intermediate representation into DICT and then performing predicate evaluation on compressed DICT over full decompression and evaluation across different dictionary sizes and data widths on AVX-512 and Neon ISAs. The values represent the X-fold speedup achieved by the optimised approach, with values indicating an increase in performance highlighted in grey.

Dictionary Size	**AVX-512 Performance Boost**				**Neon Performance Boost**			
	`uint64_t`	`uint32_t`	`uint16_t`	`uint8_t`	`uint64_t`	`uint32_t`	`uint16_t`	`uint8_t`
1	1.64	1.51	2.00	1.98	2.15	2.00	2.06	2.03
64	1.57	1.44	1.91	1.89	2.01	1.87	1.93	1.89
128	1.47	1.35	1.79	1.77	1.86	1.73	1.78	1.75
192	1.39	1.28	1.69	1.68	1.71	1.59	1.64	1.61
256	1.30	1.20	1.59	1.57	1.51	1.41	1.45	1.42
320	1.24	1.15	1.51	1.50	1.44	1.34	1.38	1.36
384	1.19	1.09	1.45	1.44	1.38	1.28	1.32	1.29
448	1.14	1.05	1.39	1.37	1.31	1.22	1.26	1.24
512	1.09	1.01	1.33	1.32	1.26	1.17	1.21	1.19
576	1.05	0.97	1.28	1.27	1.21	1.12	1.16	1.14
640	1.01	0.93	1.23	1.22	1.16	1.08	1.11	1.09
704	0.98	0.90	1.19	1.18	1.12	1.04	1.07	1.05
768	0.94	0.87	1.15	1.14	1.08	1.00	1.03	1.01
832	0.91	0.84	1.11	1.10	1.04	0.97	1.00	0.98
896	0.88	0.81	1.07	1.06	1.00	0.93	0.96	0.94
960	0.85	0.79	1.04	1.03	0.97	0.90	0.93	0.91
1024	0.83	0.76	1.01	1.00	0.94	0.87	0.90	0.88

6.4 Discussion

We explored predicate evaluation on semi-compressed data. For DICT, we provided an algorithm capable of evaluating data without decompression for the $=$ and $\neq$ predicates. We explained why such optimisation is impractical in FastLanes with other predicates at the moment of writing and why it is only effective when the dictionary size is small. To find the thresholds below which it should be used, we conducted benchmarks and identified the dictionary sizes required to achieve speedups across different microarchitectures and ISAs. The minimum size we observed was 366, which is large given that dictionaries are typically used when data has few unique values. We also quantified the speedup obtained with AVX-512 and Neon ISAs with different dictionary sizes. Evaluating compressed DICT data is at least 7 times faster when the dictionary contains 64 elements or fewer and at least 3 times faster when it contains 128 elements or fewer, compared to full decompression followed by evaluation. Since we considered only the scenario with a single vector of data, while a dictionary may contain values from a row group comprising many vectors, the actual speedup could be even greater. Therefore, our favourable results represent a lower bound on the potential speedup.

We then examined the RLE intermediate representation, noting that its run values array resembles the dictionary array in DICT, but where values can repeat if they are non-consecutive. Due to this property, we cannot apply an optimisation similar to that used for DICT. We discussed the idea of an adaptive approach to optimrise RLE evaluations based on match counts in the run values array, leaving its implementation for future work. Then, we discussed the combined DICT + RLE intermediate representation. We demonstrated the speedup achievable by decompressing it into DICT and then applying our optimisation. This approach is faster than full decompression followed by evaluation because full decompression would involve decompressing RLE regardless, which the results of our benchmarks support. We observed that while there is a speedup, it is not as substantial as with DICT alone. We recommend decompressing DICT + RLE into DICT and then applying the optimisation if the dictionary array size is below the thresholds that we found.

Chapter 7

Discussion and Future Work

At the beginning of this thesis, we established three research questions. Let us reflect on each of them.

Can we develop data-parallel predicate evaluation methods that are both fast and portable across various hardware architectures without resorting to multiple platform-specific implementations? As we have demonstrated, the answer to this question is affirmative. We implemented and benchmarked four algorithms that utilise the widest available SIMD registers. We achieved data parallelism primarily by observing that, at runtime, the specific order in which we store the results in the selection bitmap does not matter. We leveraged this flexibility to our advantage and developed fully data-parallel algorithms.

Additionally, we implemented two algorithms using AVX-512 intrinsics and showed that employing platform-specific implementations yielded no performance gains over our best scalar algorithm. Therefore, it is possible to create fast and portable data-parallel predicate evaluation methods without relying on multiple platform-specific implementations.

Which selection data structure – selection bitmap, selection byte map, selection vector, or reverse selection vector – is most suitable for efficient predicate evaluation, considering factors such as data parallelism, selectivity, and performance trade-offs? The efficiency of predicate evaluation algorithms depends on the selection data structure used. We demonstrated that when evaluating a single column, the data-parallel algorithm using a byte map yields the fastest results, closely followed by the data-parallel bitmap algorithm. Selection and reverse selection vectors are not competitive in this context because their algorithms are not data-parallel.

When evaluating multiple columns, we need to merge the results from each

column for both byte maps and bitmaps. We found that merging bitmaps is significantly faster because it requires performing an `AND` operation on 1,024 elements, whereas merging byte maps involves processing eight times more data. The difference is substantial enough that the bitmap becomes the preferred option in scenarios involving more than one column. Moreover, as the number of columns increases, the efficiency advantage of the bitmap over the byte map increases.

The selection vector outperforms other methods only at low selectivities (when very few or nearly all elements are selected). For instance, if, after evaluating some columns, we find that only a single element at the index x is selected, then when evaluating the next column, we only need to check whether the element at index x meets the predicate. This short-circuit evaluation is not possible with other selection data structures. For this reason, we proposed using a selection bitmap by default and converting the bitmap into a selection vector when selectivity falls below a certain threshold. To facilitate this, we developed and benchmarked a highly efficient method that uses lookup tables for this conversion. Our benchmarks indicated that the threshold at which it becomes advantageous to switch to a selection vector is at a selectivity of 1.8%, with significant speedups at even lower selectivities. However, since such scenarios are rare, we decided not to include this optimisation in FastLanes to avoid complicating the implementation. The same consideration applies to the reverse selection vector – it is designed to optimise the space efficiency of the regular selection vector for selectivities greater than 50%, and our findings indicate that the regular selection vector is only a viable option at far lower selectivities. Therefore, the answer to this question is that the selection bitmap is the most suitable data structure for efficient predicate evaluation in FastLanes. These conclusions only apply to numeric data and may change if we consider string columns. Since evaluating string predicates is more expensive, the ability to short-circuit evaluation with a selection vector could become more valuable.

How can we efficiently perform predicate evaluation on data compressed with cascaded encoding without fully decompressing the data? We demonstrated that with multi-layer cascaded encoding, it is possible to avoid fully decompressing the data by retaining one layer of lightweight compression and performing evaluation directly on it. Specifically, we showed that if we can decompress the data encoded in many layers to the DICT, a widely used lightweight encoding scheme [12], we can perform predicate evaluation on it. We devised an algorithm capable of evaluating equality and inequality predicates on data in DICT and demonstrated that when our dictionary contains 64 elements or fewer, we achieve a speedup of 7x or more compared to complete decompression.

Furthermore, we showed that if it is possible to decompress data into a DICT + RLE format, resulting from compressing data with RLE and then applying DICT on the run values, we can still decompress such data into the DICT format and then apply our optimisation on the dictionary to achieve a speedup, even though the first layer of the cascade is RLE and the second is DICT.

Therefore, we conclude that efficient predicate evaluation on data compressed with cascaded encoding can be achieved by partially decompressing the data to a one-layer encoding scheme like DICT and performing evaluation directly on it, thereby avoiding the overhead of full decompression.

All our results were obtained using the unsigned integer data type and apply to standard integers too. Our work can be extended to double data with additional effort, as it has additional encoding schemes. Developing specialised predicate evaluation algorithms for fully compressed data in the most commonly used compression schemes would also be beneficial. For instance, the most popular compression method in FastLanes for integers is FOR + DICT. We think it is possible to implement an algorithm that performs evaluation directly on this format, and it will likely be faster than decompressing to the DICT intermediate representation and then evaluating. Additionally, the optimisation for DICT discussed in Chapter 5 could yield even greater speedups for strings, which in many systems are compressed using DICT and on which evaluation is slow [12].

Appendix A

Benchmarking Environment

Table A.1: Devices and configurations used in the benchmarks, including architecture, SIMD extensions, core and thread counts, and base/turbo frequencies.

Processor	**Architecture**	**SIMD Extensions**	**Cores / Threads**	**Base / Turbo (GHz)**
Xeon W-2145	Skylake	SSE4.2, AVX2, AVX-512	8 / 16	3.7 / 4.5
EPYC 7402	Zen 2	SSE4.2, AVX2	24 / 48	2.8 / 3.35
i7-8750H	Coffee Lake	SSE4.2, AVX2	6 / 12	2.2 / 4.1
M1	M1	Neon	8 / 8	3.2 / 3.2

The benchmark results integrate into the narrative of this thesis. This section describes the devices and configurations used in our benchmarks. Table A.1 lists the devices, their clock frequencies, and the ISAs utilised. We compiled the code using Clang version 18.1.0. When leveraging the latest ISA available on each device, we used the compiler flag `-march=native`. When downgrading to AVX2 or SSE, we used `-mavx2` and `-mno-avx -msse4.2`, respectively. Therefore, whenever we refer to results produced with AVX-512 or Neon ISAs, we are referencing the Intel Xeon W-2145 and Apple M1 devices, respectively. An exception applies to the benchmark results, shown in Figure 5.4. Since the `VPCOMPRESSB` instruction used in one of the intrinsics algorithms is not available on the previously mentioned devices, all the results presented in that Figure were obtained using AMD Ryzen 9 7900, built on the Zen 4 architecture, operating at 3.7 GHz.

The device based on Coffee Lake is a 2018 MacBook that experiences thermal throttling. As a result, the benchmarks shown in Figures 5.7, 6.1 and 6.2 display volatility, especially when compared to results from other [micro]architectures. Since the general performance trends are still observable and throttling may also occur for end-users, we decided to include these results for their practical relevance.

Appendix B

Algorithms

In this Appendix, we present algorithms that are not included in the main body of the text. Algorithms 12, 13 and 14 are AVX-512 intrinsics algorithms for equality evaluation on `uint8_t`, `uint16_t`, and `uint64_t` data, respectively. As described in Section 5.1.5, immediately after evaluation, Algorithm 12 stores the data because it is already in the required layout. Algorithms 13 and 14, unlike Algorithm 12, contain unnecessary bits in each lane. Therefore, at the end of these algorithms, we extract the relevant data from the lanes. We employ the truncation method but also have a commented-out variant that uses compress stores, as detailed in Section 5.3. Algorithm 9 presented in the same Section implements such approach for `uint32_t` data.

Algorithm 15 is designed to efficiently convert a selection bitmap into a selection vector using precomputed lookup tables. We describe its operation and benchmark it in Section 5.3.

Algorithm 12 The AVX-512 intrinsics algorithm to evaluate "=" on 8-bit data.

```
void avx512_8(uint8_t* __restrict vec, uint8_t* __restrict
   bitmap, uint8_t filter) {

  __m512i fltr_vec = _mm512_set1_epi8(*reinterpret_cast<
     int8_t*>(&filter));

  for (int i = 0; i < 128; i += 64) {
    __m512i src0 = _mm512_loadu_si512(vec + 128 * 0 + i);
    __m512i src1 = _mm512_loadu_si512(vec + 128 * 1 + i);
    __m512i src2 = _mm512_loadu_si512(vec + 128 * 2 + i);
    __m512i src3 = _mm512_loadu_si512(vec + 128 * 3 + i);
    __m512i src4 = _mm512_loadu_si512(vec + 128 * 4 + i);
    __m512i src5 = _mm512_loadu_si512(vec + 128 * 5 + i);
    __m512i src6 = _mm512_loadu_si512(vec + 128 * 6 + i);
    __m512i src7 = _mm512_loadu_si512(vec + 128 * 7 + i);

    __mmask64 cmp0 = _mm512_cmpeq_epi8_mask(src0, fltr_vec);
    __mmask64 cmp1 = _mm512_cmpeq_epi8_mask(src1, fltr_vec);
    __mmask64 cmp2 = _mm512_cmpeq_epi8_mask(src2, fltr_vec);
    __mmask64 cmp3 = _mm512_cmpeq_epi8_mask(src3, fltr_vec);
    __mmask64 cmp4 = _mm512_cmpeq_epi8_mask(src4, fltr_vec);
    __mmask64 cmp5 = _mm512_cmpeq_epi8_mask(src5, fltr_vec);
    __mmask64 cmp6 = _mm512_cmpeq_epi8_mask(src6, fltr_vec);
    __mmask64 cmp7 = _mm512_cmpeq_epi8_mask(src7, fltr_vec);

    __m512i result = _mm512_maskz_set1_epi8(cmp0, 0x01) |
                     _mm512_maskz_set1_epi8(cmp1, 0x02) |
                     _mm512_maskz_set1_epi8(cmp2, 0x04) |
                     _mm512_maskz_set1_epi8(cmp3, 0x08) |
                     _mm512_maskz_set1_epi8(cmp4, 0x10) |
                     _mm512_maskz_set1_epi8(cmp5, 0x20) |
                     _mm512_maskz_set1_epi8(cmp6, 0x40) |
                     _mm512_maskz_set1_epi8(cmp7, 0x80);

    _mm512_storeu_si512(bitmap + i, result);
  }
}
```

Algorithm 13 The AVX-512 intrinsics algorithm to evaluate "=" on 16-bit data.

```
1  void avx512_16(uint16_t* __restrict vec, uint8_t* __restrict
       bitmap, uint16_t filter) {
2
3   __m512i fltr_vec = _mm512_set1_epi16(*reinterpret_cast<
       int16_t*>(&filter));
4
5   for (int j = 0; j < 2; j++) {
6    for (int i = j * 64; i < (j + 1) * 64; i += 32) {
7     __m512i src0 = _mm512_loadu_si512(vec + 128 * 0 + i);
8     __m512i src1 = _mm512_loadu_si512(vec + 128 * 1 + i);
                                  ...
14    __m512i src7 = _mm512_loadu_si512(vec + 128 * 7 + i);
15
16    __mmask32 cmp0 = _mm512_cmpeq_epi16_mask(src0, fltr_vec);
17    __mmask32 cmp1 = _mm512_cmpeq_epi16_mask(src1, fltr_vec);
                                  ...
23    __mmask32 cmp7 = _mm512_cmpeq_epi16_mask(src7, fltr_vec);
24
25    __m512i result = _mm512_maskz_set1_epi16(cmp0, 0x01) |
26                     _mm512_maskz_set1_epi16(cmp1, 0x02) |
27                     _mm512_maskz_set1_epi16(cmp2, 0x04) |
28                     _mm512_maskz_set1_epi16(cmp3, 0x08) |
29                     _mm512_maskz_set1_epi16(cmp4, 0x10) |
30                     _mm512_maskz_set1_epi16(cmp5, 0x20) |
31                     _mm512_maskz_set1_epi16(cmp6, 0x40) |
32                     _mm512_maskz_set1_epi16(cmp7, 0x80);
33
34
35     // Option 1: truncate and save
36    __m256i packed_result = _mm512_cvtepi16_epi8(result);
37    _mm256_mask_storeu_epi8(bitmap + i, 0xFFFFFFFF,
        packed_result);
38
39    // Option 2: compress store
40    // __mmask64 mask = 0x5555555555555555ULL;
41    // _mm512_mask_compressstoreu_epi8((void*)(bitmap + i),
        mask, result);
42   }
43  }
44 }
```

Algorithm 14 The AVX-512 intrinsics algorithm to evaluate "=" on 64-bit data.

```
void avx512_64(uint64_t* __restrict vec, uint8_t* __restrict
    bitmap, uint64_t filter) {

 __m512i fltr_vec = _mm512_set1_epi64(*reinterpret_cast<
    int64_t*>(&filter));

 for (int j = 0; j < 8; j++) {
  for (int i = j * 16; i < (j + 1) * 16; i += 8) {
   __m512i src0 = _mm512_loadu_si512(vec + 128 * 0 + i);
   __m512i src1 = _mm512_loadu_si512(vec + 128 * 1 + i);

                              ...

   __m512i src7 = _mm512_loadu_si512(vec + 128 * 7 + i);

   __mmask8 cmp0 = _mm512_cmpeq_epi64_mask(src0, fltr_vec);
   __mmask8 cmp1 = _mm512_cmpeq_epi64_mask(src1, fltr_vec);

                              ...

   __mmask8 cmp7 = _mm512_cmpeq_epi64_mask(src7, fltr_vec);
   __m512i result = _mm512_maskz_set1_epi64(cmp0, 0x01) |
                    _mm512_maskz_set1_epi64(cmp1, 0x02) |
                    _mm512_maskz_set1_epi64(cmp2, 0x04) |
                    _mm512_maskz_set1_epi64(cmp3, 0x08) |
                    _mm512_maskz_set1_epi64(cmp4, 0x10) |
                    _mm512_maskz_set1_epi64(cmp5, 0x20) |
                    _mm512_maskz_set1_epi64(cmp6, 0x40) |
                    _mm512_maskz_set1_epi64(cmp7, 0x80);

    // Option 1: truncate and save
   __m128i packed_result = _mm512_cvtepi64_epi8(result);
   _mm_mask_storeu_epi8(bitmap + i, 0xFF, packed_result);

   // Option 2: compress store
   // __mmask64 mask = 0x0101010101010101ULL;
   // _mm512_mask_compressstoreu_epi8((void*)(bitmap + i),
      mask, result);
  }
 }
}
```

Algorithm 15 This algorithm transforms a bitmap into a selection vector by iterating over each byte, using the `LENGTHS` array to count set bits, and retrieving corresponding indices from the `INDICES` array. It then fills the `selection_vector` with these indices and updates the total selection size.

```
1    static constexpr uint8_t LENGTHS[256] {
2        0, 1, 1, 2, ..., 7, 8,
3    };
4
5    static constexpr uint16_t INDICES[256 * 8] {
6        0,   0,   0,   0,   0,   0,   0,   0,   // 0b0
7        0,   0,   0,   0,   0,   0,   0,   0,   // 0b1
8        128, 0,   0,   0,   0,   0,   0,   0,   // 0b10
9        0,   128, 0,   0,   0,   0,   0,   0,   // 0b11
                                  ...
1027     128, 256, 384, 512, 640, 768, 896, 0,   // 0b11111110
1028     0,   128, 256, 384, 512, 640, 768, 896, // 0b11111111
1029 };
1030
1031 uint16_t bitmap_to_selection_vector(uint8_t* __restrict
        bitmap, uint16_t* __restrict selection_vector) {
1032
1033     uint16_t selection_vector_size {0};
1034
1035     for (uint16_t col_idx {0}; col_idx < 128; ++col_idx) {
1036         const uint8_t   offset    = bitmap[col_idx];
1037         const uint8_t   length    = LENGTHS[offset];
1038         const uint16_t* indices_p = INDICES + (8 * offset);
1039
1040         for (size_t row_idx {0}; row_idx < 8; ++row_idx) {
1041             uint16_t index           = indices_p[row_idx];
1042             uint16_t right_index     = index + col_idx;
1043             selection_vector[row_idx] = right_index;
1044         }
1045
1046         selection_vector = selection_vector + length;
1047         selection_vector_size += length;
1048     };
1049
1050     return selection_vector_size;
1051 }
```

Bibliography

[1] Arm Intrinsics Guide. https://developer.arm.com/architectures/instruction-sets/intrinsics/. Accessed: 14/9/2024.

[2] Designing a SIMD Algorithm from Scratch. https://mcyoung.xyz/2023/11/27/simd-base64/. Accessed: 14/9/2024.

[3] Dolphin Progress Report: December 2019 and January 2020. https://dolphin-emu.org/blog/2020/02/07/dolphin-progress-report-dec-2019-and-jan-2020/. Accessed: 1/11/2024.

[4] Intel Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed: 14/9/2024.

[5] Missed 'compress' codegen opportunity. https://github.com/llvm/llvm-project/issues/42210. Accessed: 14/9/2024.

[6] VPCOMPRESSB/VCOMPRESSW — Store Sparse Packed Byte/Word Integer Values Into DenseMemory/Register. https://www.felixcloutier.com/x86/vpcompressb:vcompressw. Accessed: 14/9/2024.

[7] VPCOMPRESSD — Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register. https://www.felixcloutier.com/x86/vpcompressd. Accessed: 14/9/2024.

[8] VPCOMPRESSQ — Store Sparse Packed Quadword Integer Values Into Dense Memory/Register. https://www.felixcloutier.com/x86/vpcompressq. Accessed: 14/9/2024.

[9] Afroozeh, A., and Boncz, P. The FastLanes Compression Layout: Decoding 100 Billion Integers per Second with Scalar Code. *Proceedings of the VLDB Endowment 16*, 9 (May 2023), 2132–2144.

[10] Afroozeh, A., Felius, L., and Boncz, P. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (June 2024), vol. 200 of *SIGMOD/PODS '24*, ACM, p. 1–11.

[11] Boncz, P., Neumann, T., and Erling, O. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark.* Springer International Publishing, 2014, p. 61–76.

[12] Boncz, P., Neumann, T., and Leis, V. FSST: fast random access string compression. *Proc. VLDB Endow. 13*, 12 (July 2020), 2649–2661.

[13] Braams, B. Predicate Pushdown in Parquet and Apache Spark, December 2018. Available at `https://homepages.cwi.nl/~boncz/msc/2018-BoudewijnBraams.pdf`.

[14] Damme, P., Habich, D., Hildebrandt, J., and Lehner, W. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses), 2017.

[15] Ding, J., Abrams, M., Bandyopadhyay, S., Di Palma, L., Ji, Y., Pagano, D., Paliwal, G., Parchas, P., Pfeil, P., Polychroniou, O., Saxena, G., Shah, A., Voloder, A., Xiao, S., Zhang, D., and Kraska, T. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (June 2024), SIGMOD/PODS '24, ACM, p. 55–67.

[16] Ding, J., Minhas, U. F., Chandramouli, B., Wang, C., Li, Y., Li, Y., Kossmann, D., Gehrke, J., and Kraska, T. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (June 2021), SIGMOD/PODS '21, ACM.

[17] Dreseler, M., Boissier, M., Rabl, T., and Uflacker, M. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment 13*, 8 (Apr. 2020), 1206–1220.

[18] Feng, Z., Lo, E., Kao, B., and Xu, W. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (May 2015), SIGMOD/PODS'15, ACM.

[19] Goldstein, J., Ramakrishnan, R., and Shaft, U. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering* (USA, 1998), ICDE '98, IEEE Computer Society, p. 370–379.

[20] Habich, D., Damme, P., Ungethüm, A., and Lehner, W. Make Larger Vector Register Sizes New Challenges?: Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *Proceedings of the Workshop on Testing Database Systems* (June 2018), SIGMOD/PODS '18, ACM, p. 1–6.

[21] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[22] Jiang, H., and Elmore, A. J. Boosting data filtering on columnar encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (June 2018), SIGMOD/PODS '18, ACM.

[23] Kuschewski, M., Sauerwein, D., Alhomssi, A., and Leis, V. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data 1*, 2 (June 2023), 1–26.

[24] Lemire, D., and Boytsov, L. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper. 45*, 1 (Jan. 2015), 1–29.

[25] Li, Y., Lu, J., and Chandramouli, B. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proceedings of the ACM on Management of Data 1*, 2 (June 2023), 1–26.

[26] Li, Y., and Patel, J. M. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (June 2013), SIGMOD/PODS'13, ACM.

[27] Mukhtarov, Z. Nested Data-Type Encodings in FastLanes, July 2024.

[28] Ng, W.-K., and Ravishankar, C. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering 9*, 2 (1997), 314–328.

[29] Ngom, A., Menon, P., Butrovich, M., Ma, L., Lim, W. S., Mowry, T. C., and Pavlo, A. Filter Representation in Vectorized Query Execution.

In *Proceedings of the 17th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2021), DAMON '21, Association for Computing Machinery.

[30] Robinson, A., and Cherry, C. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE 55*, 3 (1967), 356–364.

[31] Răducanu, B., Boncz, P., and Zukowski, M. Micro adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (June 2013), vol. 27 of *SIGMOD/PODS'13*, ACM, p. 1231–1242.

[32] Welch. A Technique for High-Performance Data Compression. *Computer 17*, 6 (1984), 8–19.

[33] Willhalm, T., Oukid, I., Müller, I., and Faerber, F. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013* (2013), R. Bordawekar, C. A. Lang, and B. Gedik, Eds., pp. 1–12.

[34] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment 2*, 1 (Aug. 2009), 385–394.

[35] Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., and Zhang, H. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment 17*, 2 (Oct. 2023), 148–161.

[36] Zhang, W., Wang, Y., and Ross, K. A. Parallel Prefix Sum with SIMD, 2023.

[37] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5 (1978), 530–536.

[38] Zukowski, M., Heman, S., Nes, N., and Boncz, P. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)* (2006), IEEE.