2025

FASTLANES

Azim Afroozeh

VRIJE UNIVERSITEIT

FastLanes: A Next-Gen File Format

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
volgens besluit van de decaan
van de Faculteit der Bètawetenschappen
in het openbaar te verdedigen
op vrijdag 9 januari 2026 om 13.45 uur
in de universiteit

door

Azim Afroozeh
geboren te Yazd, Iran

promotor:	prof.dr. P. Boncz
copromotor:	dr. H. Mühleisen
promotiecommissie:	dr. P. Tözün prof.dr. D. Lemire prof.dr. V. Leis prof.dr. J.R. van Ossenbruggen dr. J. Urbani

SIKS Dissertation Series No. 2026-06
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

DOI: 10.5463/thesis.1348

Declaration of Authorship

I, Azim Afroozeh, hereby declare that the thesis titled *"FastLanes: A Next-Gen File Format"* and its content are the result of my own work.
I confirm that:

- This work was primarily conducted during my pursuit of a research degree at these universities.
- If any portion of this thesis has been previously submitted for an academic degree or any other qualification at these universities or any other educational institution, I have explicitly disclosed this information.
- I have consistently acknowledged the sources of published works by other authors that I consulted.
- In cases where I have included excerpts from the work of others, I have consistently provided proper source attributions. Aside from these quotations, the entirety of this thesis represents my independent effort.
- I have acknowledged all significant sources of assistance.
- In cases where this thesis draws upon collaborative efforts with others, I have provided a clear distinction between the contributions made by collaborators and my own individual contributions.

17 November 2025

To My Mother, Nane Goli,
To My Father, Baba Goli,
To My Love, Jasmin Goli

Everything that happens is a chance to move forward, to grow, to improve.

Ryan Holiday

Contents

Acknowledgments

Peter: My first and foremost thanks go to Peter. Simply put, *FastLanes* would not have been possible without him. I would like to thank Peter for being so generous in sharing his knowledge, ideas, and time—whether after 12 PM, on weekends, during holidays, or whenever I needed support. It felt amazing to have someone so consistently present for my research. I learned a lot from Peter on many levels—both academic and personal—and I will always be grateful to him.
I would also like to thank him for leading the CWI Database Group in a way that gave all of us—and me personally—the opportunity to work on meaningful projects, the kind that can genuinely make data processing more efficient and, hopefully, in some way, improve people's lives. Finally, thank you for giving me the opportunity—a postdoc position at CWI—to further advance and promote *FastLanes*.
Daniel: I would like to thank my roommate, Daniel. It is because of Daniel that I connected with the group, saw myself as part of it, and truly experienced what it meant to be part of CWI. I can't think of a single day in the office when we didn't laugh. Thank you for teaching me Dutch language and culture, and for listening to me during tough times. You are the best office mate I could ask for.
CWI: I would also like to thank everyone at CWI for making me feel so welcome and happy during my time there. Special thanks to Laurens for being such a kind and cheerful roommate—short but sweet! Thanks to Stefan for always being helpful and full of guidance. I'm grateful to everyone who contributed to the CWI group atmosphere: Dean, Tim, Gabor, Hannes, Mark, Ilaria, Pedro, Madelon, Matheus, Nantia, Orson, Diego, Yimming, Simei, Florian, Thijs (WebGPU expert), Thijs, Connor, and Stefano. I truly felt at home at CWI.
My Students: I want to thank the students I had the pleasure of supervising—they played a significant role in the *FastLanes* journey.
Leonardo: Together with Peter and Lonardo, we created ALP and submitted a paper to SIGMOD in less than two months. More than this major achievement, I enjoyed every second of working with him—and the best possible outcome is that I now have a true friend. Sharing a room with him makes L324 the happiest place on Earth.
Thomas: Thomas created C3 and significantly contributed to *FastLanes*, especially in handling multiple columns. We also had a lot of fun, particularly on our trip to Texel.
Ziya: A brilliant C++ developer and an algorithm genius. Thanks to him, we now have the RealNest benchmark suite, which involved substantial work and thanks to him we know how to compress nested data efficiently.
Raufs: Thanks to Raufs, we learned how to handle predicate pushdown in *FastLanes*. He is one of the kindest people I've ever met, and we've shared many great moments—especially playing table tennis.
Sven: The true GPU expert. I thoroughly enjoyed working with him, always laughing at how the GPU works. He made a significant contribution in helping *FastLanes* find its way to GPU processing.

Sebastiaan: My newest student, and without a doubt one of the most positive and enthusiastic people I've worked with. I truly enjoy collaborating with him and am confident he will do just as great as my other students.
Paul: Thank you for making L324 an even happier place. We laughed a lot together, and I'm happy to see you've already surpassed my Dutch level!
Lotte: Thank you for your help with Dutch pronunciation and for our collaboration on GPU. I really enjoyed working with you, and our early efforts significantly contributed to the GPU side of *FastLanes*.
Reza: I am always grateful for the trip and the conversations we had over the course of almost a month during the most critical part of my PhD. They helped me get through it.
Anastasia and Nadezhda: I would like to thank you for your support throughout my PhD—especially for the headphone, which I used every single day of my PhD.
My Parents: I would like to thank my parents with all my heart. It is because of your sacrifices that I had the opportunity to move to Amsterdam—to study, to live, and to work on something I truly love. Thank you for all the support you gave me—every night through our Skype calls, and throughout my PhD journey. It is very hard to put into words how much I owe you. You are the best Mom and Dad I could ask for. I hope you are proud of me.
Ali: My big brother—thank you. Because of you, I never felt like I had moved to a new country. Your constant support made it feel like home. I'm especially grateful for all your help during my Master's thesis, which eventually led to my position at CWI. And thank you for standing by me through the toughest times during my PhD. You are the best brother anyone could ask for.
Amir: Thank you for paying my tuition fees and supporting me at such a critical time. I'm also grateful for all the fun we had throughout my PhD—you made the journey lighter and more joyful.
Omid: You are the coolest brother I could ask for! All the weekends we spent together, and working side by side, kept me productive and gave me the stamina to push through the final year of my PhD and bring this thesis to the finish line. But remember—you're still my little brother, and I'll always take care of you :D.
Jasmin: My last and biggest thanks go to you—my love. I wrote this thesis while being with you—literally by your side—and you supported me through the final and most important steps of the journey. You cooked for me, encouraged me, and stood by me, especially during the writing of this thesis and my final paper. Without your love, support, and gentle push, I don't know when I would have finished. Thank you for everything.

1

Introduction

File formats have evolved alongside computing, progressing from early text-based and proprietary formats to highly efficient big data formats such as Apache Parquet [1], which builds on decades of research in database storage and file format design. This evolution has been driven by critical and evolving needs across different eras and use cases, including compatibility with heterogeneous systems, human readability, support for hierarchical data, efficient compression of large volumes of data, and adaptability to modern hardware, including GPUs.

Early Computer Systems and Proprietary Formats (1950s–1960s). In the early days of computing, each computer manufacturer developed proprietary file formats tailored to specific hardware and software [2]. These formats were often undocumented and incompatible across different systems [3].

Emergence of Standardized File Formats (1970s–1980s). As computing expanded into business, academia, and government applications, the need for standardization became evident, as organizations required efficient ways to exchange data between different systems—something proprietary formats could not provide. One widely adopted solution was **ASCII text files**, originally standardized in the 1960s, which became a common format for storing and exchanging textual data across platforms in the following decades [4]. Building on this, **Comma-Separated Values (CSV)** gained popularity as a simple and effective format for tabular data, particularly with the rise of database management systems and early spreadsheet software.

PC Era (1980–1995) As personal computers became general-purpose tools for business and administrative tasks, tabular file formats emerged as a foundation for everyday data management. With affordable microcomputers such as the IBM PC becoming available to businesses and individuals [5], computing power moved out of centralized IT departments and into everyday workplaces. This shift empowered non-technical users—such as accountants, office clerks, and small business owners—to perform tasks that had previously required manual processing or specialized systems [6]. Common activities like inventory tracking, payroll management, customer databases, and budgeting demanded reliable mechanisms for storing and manipulating structured, tabular data [7]. This created a

1

growing need for file formats capable of representing tabular information in a consistent and accessible way.

To meet this demand, software developers created file formats specifically designed for tabular data. Formats like dBASE's `.dbf` and Microsoft Excel's `.xls` offered compact, portable representations of tabular data that could be easily created, saved, edited, and exchanged across applications [8, 9]. These formats stored data in a way that mirrored how real-world information was organized—using rows and columns to represent entries and attributes.

Internet Era and Hierarchical Formats (1995–2010) The rapid expansion of the internet in the 1990s revolutionized how data was created, stored, and shared. Many applications, such as e-commerce platforms, required *nested data structures* to efficiently manage customer details, order histories, and product listings. However, existing formats like CSV, plain text, dBASE's `.dbf`, and Microsoft Excel's `.xls` files proved inadequate, as they lacked features, like support for hierarchical relationships; or were not open standards.

To address these limitations, **XML (Extensible Markup Language)** and **JSON (JavaScript Object Notation)** emerged as dominant solutions [10, 11]. XML, introduced in the late 1990s, provided a *self-descriptive, hierarchical format* for encoding data, but it was often criticized for its verbosity and parsing complexity, leading to performance issues in large-scale applications [12]. JSON, introduced in the early 2000s, became the preferred alternative due to its *lightweight, human-readable structure* and seamless integration with JavaScript-based environments. Unlike XML, JSON's simplicity and efficiency made it ideal for modern web applications [13].

By the early 2000s, data processing needs continued to evolve as organizations embraced *web-scale applications*, *cluster technologies*, and early *cloud-based services*. This big data era marked a transitional period that introduced new challenges in *interoperability*, *schema evolution*, and *binary data serialization* across distributed systems. To meet these demands, new formats emerged—bridging the gap between human-readable formats and scalable, machine-friendly storage solutions.

Two early technologies from this era were **Apache Avro** [14] and **RCFile (Record Columnar File)** [15]. Avro provided *row-based serialization* with built-in support for schemas, schema evolution, and compact binary representation, making it a strong fit for messaging and archival use cases within the Hadoop ecosystem. RCFile, on the other hand, pioneered *columnar storage within row groups*, introducing performance optimizations tailored for analytical workloads and influencing future formats such as ORC and Parquet [15]. These innovations laid the foundation for scalable data infrastructure in the big data and cloud computing era.

Cloud Computing Era (2010s–2020s) With the rise of big data and cloud computing, specialized file formats such as Parquet [1] and ORC [16] have emerged to efficiently manage massive datasets by optimizing storage, retrieval, and processing [17]. **Parquet** has become the standard and offers a range of features missing in traditional formats like CSV, JSON, and XML—namely, columnar storage, high compression ratios enabled by heavyweight algorithms such as ZSTD, and row group skipping. A row group is a partition of tabular data, and skipping enables filtering of unrelated data at the row group level using

min/max statistics.

AI + the End of Moore's Law (2020s–Present) Over the past 25 years, hardware evolution has slowed at the transistor level, marking the gradual end of Moore's Law [18]. In response, Compute architecture has become more diverse due to hardware specialization to support specific workloads. CPUs have adopted increasingly wide SIMD instruction sets—from 128-bit SSE to 256-bit AVX, AVX2, and eventually 512-bit AVX-512. Meanwhile, GPUs have become central to modern computing—not only due to the rise of machine learning and AI, but also because of their suitability for accelerating OLAP workloads, real-time analytics, and large-scale query processing [19, 20]. GPUs offer massive parallelism, high memory bandwidth, and the ability to process large volumes of data concurrently [21].

Despite these advances, Parquet, the state of the art, remains rooted in design decisions from an earlier era. Influenced by the Java Virtual Machine (JVM)—which historically lacked native SIMD support[1]—Parquet cannot fully exploit SIMD acceleration [24]. Moreover, the evolution of Parquet through versions 2 [25], 2.4.0 [26], and now early proposals for 3.0 [27] has not—and does not plan to—incorporate optimizations for modern hardware. Even variants like Grafana Tempo's `vParquet3` prioritize traceability and integration with distributed tracing tools over hardware-level performance tuning [28]. Parquet remains poorly suited to GPUs [29]. Its complex, branch-heavy decoding logic and layered encodings—such as dictionary encoding combined with run-length encoding (RLE), followed by general-purpose compression—introduce performance bottlenecks, including warp divergence on GPUs and branch mispredictions on CPUs [24, 29]. Additionally, it lacks support for fine-grained, vector-at-a-time decoding—essential for modern CPU vectorized compressed execution [30]—as well as tile-based execution, the dominant model in GPU-optimized academic systems [31].

These pressures—namely, better utilization of existing compute power in modern CPUs, SIMD instructions, adaptation to the rise of AI and GPUs, and the need for a file format that aligns with current execution models—create a design opportunity for a new file format. In our view next-generation formats must be architected to align with modern CPU and GPU capabilities, leveraging the parallelism of today's hardware. They should incorporate research advances like Multi-Column Compression (MCC) [32], which enables efficient compression across multiple columns by exploiting inter-column relationships to further compress data—overcoming historic weaknesses of columnar storage compared to row-based formats. It should also adopt approaches like the Whitebox Compression Model [33], that can address storage and processing issues by poor data design. For example, it can split a column with values such as `FastLanes25071994` into two columns—one with `FastLanes` and one with `25071994`—separating the string and numeric parts. This allows the numeric portion to be fed into an integer compressor, improving compression efficiency, and reducing entropy in the string part, e.g. enabling dictionary compression. Finally, they should natively support novel compression schemes developed over the past decade, enabling efficient encoding, decoding, and execution in the heterogeneous, AI-driven workloads of today.

[1]The Vector API, offering an almost-explicit SIMD interface, was only introduced in 2016 [22], and stable auto-vectorization did not appear until 2012 [23].

1

1.1 Research Questions and Contributions

FastLanes is a project initiated at CWI, intended as a foundation for the next generation of big data file formats. In this thesis, we explore the design of this new file format, with a particular focus on addressing the limitations of Parquet in the context of modern hardware—not just by proposing new solutions, but by thoroughly investigating their practical implications, including aspects such as code maintainability, which is often overlooked in research. To achieve this, we implement all our algorithms in multiple ways—purely from an engineering perspective—measuring the trade-offs between performance and maintainability. Furthermore, we provide high-quality implementations and open-source our entire codebase, making the results fully reproducible and accessible to the community.

Arguably, the most important component of Parquet—and indeed any modern big data file format—is the set of compression schemes used to compress the data, as they directly impact both read (decompression) speed and storage size (compression ratio). Any compressed data that is relevant to a query (i.e., not filtered out) must be decompressed—either fully or partially, in the case of compressed execution—before it can be processed, making decompression (almost[2]) an unavoidable step in query execution. Therefore, our first objective is to investigate how the decompression speed of compression schemes can be improved.

There are two broad categories of encodings to consider: lightweight compression (LWC) schemes [34] and heavyweight compression (HWC) schemes. While HWCs are effective at improving I/O throughput through high compression ratios, they are not well-optimized for modern CPUs [35]. This is due to their block-based nature—these schemes often require decoding large blocks at once, resulting in repeated movement of data between cache and RAM: data is brought into cache, decoded, and then written back to RAM. In contrast, LWC schemes can be decoded at a much finer granularity, following a decompression model known as *vectorized decoding* [34]. Inspired by vectorized processing, this model operates on small batches of data that are loaded into cache, decoded, and immediately processed without unnecessary memory traffic.

Consequently, our focus is on LWCs. On CPUs, these schemes can be accelerated by designing decoders that are SIMD-friendly, capable of decoding many values at once. Given the importance of achieving fast decoding using SIMD instructions, we posed our first research question:

Research Question 1: *How can SIMD instructions be leveraged to accelerate the decompression of lightweight compression schemes (LWCs)?*

[2] Systems such as DuckDB, Procella, and Velox do not always decompress data in the scan operator. They support compressed execution, where decompression is delayed until necessary. This is achieved using compressed vectors, which, unlike regular vectors that hold batches of decoded data, store values in compressed form. For example, a dictionary-compressed vector holds a pointer to the dictionary and a sequence of codes referencing its entries. The dictionary indexes themselves are typically encoded and must be decoded, but full materialization of values is avoided for as long as possible by pushing compressed vectors through the pipeline and operating directly on the indexes.

Research Question 1 is investigated in **Chapter 2**.

Chapter 2 investigates how lightweight encodings—such as bit-packing, run-length encoding (RLE), frame-of-reference (FOR), and delta encoding—can be accelerated using SIMD instructions. It explores key challenges, including the dependency chains inherent in delta encoding and the heterogeneity of SIMD instruction sets across different architectures. To address these challenges, the encodings are redesigned to be fully data-parallel, enabling efficient use of SIMD instructions and significantly improving decompression performance. Fully data-parallel encodings can be implemented in four different ways: (1) scalar code with auto-vectorization, (2) compiler intrinsics, (3) explicit SIMD instructions, and (4) third-party libraries such as XSIMD [36]. Scalar code relies on simple loops and readable constructs, delegating vectorization to the compiler's auto-vectorization mechanisms. While this approach is the most portable, it heavily depends on compiler heuristics and may not always yield optimal performance [37, 38]. Compiler intrinsics expose low-level SIMD instructions through C/C++ functions that map directly to hardware operations (e.g., AVX or NEON), offering more control than scalar code while maintaining some degree of portability [39]. Explicit SIMD programming involves writing platform-specific instructions—often through intrinsics or even assembly—enabling maximum performance at the expense of portability and maintainability [40]. Finally, third-party libraries such as XSIMD abstract away low-level complexities while still leveraging SIMD optimizations under the hood, providing a balanced trade-off between performance, readability, and cross-platform support [36].
Having considered these SIMD programming paradigms, we now pose the next question:
Research Question 2: *Can data-parallel encodings be implemented in the most maintainable way—namely, scalar code with auto-vectorization—by relying on compilers to generate SIMD instructions, while still achieving maximum performance?*

Research Question 2 is investigated in **Chapter 2**.

Chapter 2 investigates how our data-parallel layouts can be implemented. This chapter demonstrates that it is entirely *possible* to implement fully data-parallel encodings using scalar code with the auto-vectorization paradigm and still achieve the performance of explicit SIMD, by redesigning the encodings to be extremely simple—consisting only of straightforward instructions with no control flow or data dependencies. To validate this claim, the chapter also presents an implementation of the same encodings using explicit SIMD instructions and compares the two approaches. The results show that there is **no performance gap** between scalar code with auto-vectorization and explicit SIMD. This highlights that we can achieve the best of both worlds: maximum performance and high maintainability using only scalar code.
There has been significant focus on data-parallelizing and implementing encoding schemes such as Run-Length Encoding (RLE), Frame-of-Reference (FOR), Bit-Packing, Dictionary Encoding, and Delta Coding for integer data types. However, floating-point data is also becoming increasingly important in modern workloads [41]. Despite this, encoding schemes tailored specifically for floating-point data have only recently begun to receive serious attention. Existing techniques—starting with Gorilla [42] and continuing with the more recent ELF [43]—have shown promising compression ratios on certain datasets. However,

they fall significantly short in performance when compared to the lightweight schemes integrated into FastLanes. Motivated by this gap, we decided to design a new compression scheme for floating-point data from the ground up. We began by analyzing the properties of real-world floating-point datasets, with the goal of identifying patterns in their complex binary structure that could be exploited for both compression and high-performance, data-parallel decoding. This leads us to our next research question:

Research Question 3: *Is it possible to design a data-parallel encoding for floating-point numbers that uses SIMD instructions to decode many values in parallel while achieving a compression ratio comparable to heavyweight compressors (HWCs)?*

Research Question 3 is addressed in **Chapter 3**.

Chapter 3 investigates the bit-level properties of floating-point values in real-world datasets and how these properties can inform the design of a new SIMD-friendly encoding scheme for floats. The chapter explores how vectorized decoding can be utilized to accelerate decompression. It introduces a novel encoding format, **ALP**, which delivers superior performance across all three key metrics of a compression scheme: encoding speed, decoding speed, and compression ratio.
With a complete pool of data-parallelized lightweight compression schemes (LWCs) implemented on the CPU, it becomes compelling to evaluate their performance on GPUs and explore how they can be further optimized—especially given the fundamentally different nature of GPU architectures. GPUs offer significantly higher parallelism but are constrained by much smaller fast scratchpad memory (including registers and shared memory), in contrast to the large caches available on CPUs. These architectural differences necessitate rethinking how compression schemes are designed and executed. Given that one of the primary goals of FastLanes is to support AI workloads—where GPU execution is essential—this evaluation becomes even more critical. Therefore, we pose our next research question:

Research Question 4: *Do data-parallelized encodings, originally tailored for CPUs, remain efficient on GPUs? What is their impact when integrated into query execution engines on GPUs, and can they be further optimized?*

Research Question 4 is addressed in **Chapter 4**.

Chapter 4 investigates how the data-parallel encodings proposed in Chapter 2 perform on GPUs, leveraging warp-level parallelism. The chapter evaluates their performance both when used solely for decoding and when integrated into a full query engine. It explores how this baseline GPU implementation can be further optimized, particularly to address bottlenecks unique to GPUs—most notably, limited local memory. To mitigate this, the FastLanes API is redesigned to support more fine-grained decoding, enabling query engines to maintain higher occupancy, even when executing multi-column queries that exert significant pressure on local memory.
From our initial work on GPU, described in Chapter 4, we learned two key lessons. First, data-parallel layouts that expose at least 32 independent tasks align well with the GPU's warp-based execution model, where each warp consists of 32 threads. Second, the original

FastLanes API, which delivers 1024 values at a time, becomes a bottleneck on GPUs and must be redesigned for finer granularity. With these insights, we extended our evaluation of lightweight encodings on GPUs, focusing particularly on ALP. Our goal was to completely rethink how such encodings should be optimized for GPU execution and investigate whether the same encoding could be made performant on both CPU and GPU. This led us to pose the following research question:

Research Question 5: *How should LWCs be implemented on GPUs? What should their API look like?*

Research Question 5 is addressed in **Chapter 5**.

Chapter 5 builds upon the insights presented in Chapter 4 and explores the GPU performance of ALP. After identifying its limitations, the chapter proposes two core principles for GPU optimization: (1) all parts of the decoding process must be fully data-parallelized, and (2) the decoding API should deliver one value at a time per thread to minimize local memory pressure. The benefits of these ideas are demonstrated through multi-column queries, where increased memory pressure on GPUs highlights the importance of fine-grained and efficient decoding. These optimizations are presented as guidelines for designing future GPU-friendly file formats.
While LWCs are highly efficient in terms of decompression speed they fall significantly behind heavyweight compression schemes (HWCs) when it comes to compression ratio [44]. This observation motivates our next research question:

Research Question 6: *Can data-parallel lightweight encodings be used to achieve better compression ratios than heavyweight compressors (HWCs) while maintaining the key advantages of lightweight encodings, such as support for compressed execution, fast decoding, and vectorized processing?*

Research Question 6 is addressed in **Chapter 6**.

Chapter 6 explores how the data-parallel LWCs proposed in Chapter 2 can be composed to outperform heavyweight compression schemes (HWCs) in terms of compression ratio. To achieve this, the chapter redesigns the overall compression model by introducing *Expression Encoding*, which uses a small interpreted expression language to represent arbitrary combinations of encodings. It further examines which combinations are necessary to surpass HWCs in compression ratio on the `PUBLIC_BI` dataset.
With *Expression Encoding* established as the dominant compression model, the next natural question is how to design a file format around it. This leads us to our final research question—the last piece of the puzzle in building the file formats of the future:

Research Question 7: ***What could a file format built on the ideas from this thesis look like?***

Research Question 7 is addressed in **Chapter 6**.

Chapter 6 explores how a new compression model, *expression encoding*, can serve as the

1

foundation for designing a future-ready file format that incorporates a wide range of advanced, research-driven features such as MCC [32] into a single, cohesive file format.

1.2 Thesis Outline & Publications

Chapter 2 investigates how lightweight compression schemes (LWCs) can be fully SIMDized through the careful design of data-parallelized layouts. This chapter is based on the work presented in the following paper:

> ***The FastLanes Compression Layout: Decoding 100 Billion Integers per Second with Scalar Code***
> Azim Afroozeh and Peter Boncz
> Published in the *Proceedings of the VLDB Endowment*, Vol. 16, No. 9, pp. 2132–2144, 2023.
> Presented at the International Conference on Very Large Data Bases (VLDB), 2023.

Chapter 3 investigates the design of a novel lightweight compression scheme (LWC) for floating-point numbers that is fully data-parallelized and achieves a high compression ratio. This chapter is based on the work presented in the following paper:

> ***ALP: Adaptive Lossless Floating-Point Compression***
> Azim Afroozeh, Leonardo Kuffo, Peter Boncz
> Published in the *Proceedings of the ACM on Management of Data*, Vol. 1, No. 4, pp. 1–26.
> Presented at the 2024 ACM SIGMOD/PODS Conference, Santiago, Chile

Chapter 4 investigates the performance of FastLanes data-parallel LWCs on GPUs and their integration into Crystal [31], the state-of-the-art academic GPU database system. This chapter is based on the work presented in the following paper:

> ***Accelerating GPU Data Processing using FastLanes Compression***
> Azim Afroozeh, Charlotte Felius, Peter Boncz
> Presented at ACM SIGMOD/PODS 2024, Santiago, Chile – Monday, June 10, 2024.
> Published in the *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN)*, 2024.

Chapter 5 investigates the performance of FastLanes ALP on GPUs and how it can be optimized, establishing guidelines for the design of other LWCs on GPU. This chapter is based on the work presented in the following paper:

> ***G-ALP: Rethinking Lightweight Encodings for GPUs***
> Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, Stefan Manegold
> Submitted to the *21st International Workshop on Data Management on New Hardware (DaMoN)*, ACM SIGMOD/PODS 2025, Berlin, Germany – June 23, 2025.

Chapter 6 investigates how lightweight compression schemes (LWCs) can be combined to achieve higher compression ratios while preserving their performance benefits through a novel compression model called *expression encoding*. It further presents the design and implementation of the *FastLanes File Format*, which is built around this model and serves as a foundation for the next generation of efficient and extensible big data file formats.

> ***The FastLanes File Format***
> Azim Afroozeh, Peter Boncz
> Submitted to the *51st International Conference on Very Large Data Bases (VLDB)*, London, United Kingdom – September 1–5, 2025.

Chapter 7 concludes the thesis and shares our vision for the future of the FastLanes file format.

2

Data Parallelized Encodings

The open-source FastLanes project aims to improve big data formats, such as Parquet, ORC and columnar database formats, in multiple ways. In this chapter, we significantly accelerate decoding of all common Light-Weight Compression (LWC) schemes: DICT, FOR, DELTA and RLE through better data-parallelism. We do so by re-designing the compression layout using two main ideas: (i) generalizing the value interleaving *technique in the basic operation of bit-(un)packing by targeting a virtual 1024-bits SIMD register, (ii) reordering the tuples in all columns of a table in the same Unified Transposed Layout that puts tuple chunks in a common "04261537" order (explained in the chapter); allowing for maximum independent work for all possible basic SIMD lane widths: 8, 16, 32, and 64 bits.*

We address the software development, maintenance and future-proofness challenges of increasing hardware diversity, by defining a virtual 1024-bits instruction set that consists of simple operators supported by all SIMD dialects; and also, importantly, by scalar code. The interleaved and tuple-reordered layout actually makes scalar decoding faster, extracting more data-parallelism from today's wide-issue CPUs. Importantly, the scalar version can be fully auto-vectorized by modern compilers, eliminating technical debt in software caused by platform-specific SIMD intrinsics.

Micro-benchmarks on Intel, AMD, Apple and AWS CPUs show that FastLanes accelerates decoding by factors (decoding 40 values per CPU cycle). FastLanes can make queries faster, as compressing the data reduces bandwidth needs, while decoding is almost free.

2.1 Introduction

Analytical data systems routinely employ columnar storage. This allows queries to skip columns that they do not need, saving network, disk and memory bandwidth. Further, columnar storage tends to be more compact than row storage, thanks to *compression*.

Vectorized execution is a broadly adopted design for query execution where computational work in query expressions is performed on chunks of e.g., 1024 values called "vectors", by an expression interpreter that invokes pre-compiled functions that perform simple actions in loops over these vectors (arrays), thus amortizing function call overhead over 1024 tuples

and allowing compilers to optimize these functions using techniques like loop-pipelining, code motion and auto-vectorization: generation of SIMD instructions [45].

Vectorized decoding carries over these efficient properties when applied to decoding compressed data. We focus on FOR, DICT, DELTA and RLE (resp. the Frame Of Reference [46], Dictionary, Delta and Run Length encodings). Also, when a vectorized table scan decompresses a vector, (compact) compressed data in RAM gets decompressed into an uncompressed vector, which is a small array of 1024 values, that fits the CPU L1/L2 caches and is immediately processed by the query pipeline, so it typically does not spill to RAM. As such, decompression happens between RAM and CPU, reducing memory, network and disk bandwidth consumption [47].

Parquet [1] also uses columnar encodings, albeit using a scheme that always applies DICT and represents the dictionary codes in variable-sized runs using bit-packing or RLE. Such variable-sized adaptivity hinders fast vectorized decoding [48], and the non-interleaved bit-packing and classic RLE it uses do not expose the opportunities for data-parallelism introduced by our techniques.

Compressed execution. We think scans in next-gen database systems should not decompress columns eagerly to their SQL type, which often is a wide integer (e.g., a decimal stored in 64-bits), but rather to the smallest type that makes the values processable by query operators. Modern systems like Procella [49], Velox [50] and DuckDB [51] support *compressed vector*s, where data is both randomly accessible yet still partially compressed: e.g., a FOR-vector or a DICT-vector, where 1024 values are represented as `uint8[1024]`, accompanied by one `uint64` base (FOR), resp. a pointer to a Dictionary. Such tight representations unlock optimizations (e.g., SIMD) for operators higher in a pipeline, and reduce the size of data structures, lessening (cache) memory pressure. It also causes *best case* scan decoding performance, where one decompresses a vector to its smallest possible lane-width, to become the *common case*.

FastLanes is a project initiated at CWI, intended as a foundation for next-generation big data formats. It introduces a new layout for compressed columnar data that increases the opportunities for data-parallel decoding, improving performance by factors. It does so in a way that works across the heterogeneous and evolving Instruction Set Architectures (ISAs) landscape, is future-proof, and minimizes technical debt by relying on scalar-only code.

2.1.1 Challenges and Contributions

In the FastLanes project we are re-designing columnar storage to expose more independence in data decoding, to make future query engines better at exploiting data-parallelism present in modern hardware. We contribute solutions to six challenges in Table 1:

Many SIMD widths. In the course of 25 years, SIMD ISAs have widened by a factor 8. Rather than taking the current widest SIMD ISA and proposing a data layout optimized for it, we preempt further widening of SIMD registers and propose a layout optimized for a **virtual 1024-bits register** `FLMM1024` that gets the best performance out of any existing ISA, and even from scalar code. At the lowest level of bits, this means FastLanes applies an *interleaved* bit-packed layout to 1024 bits; which distributes all logically subsequent e.g., 3-bit values round-robin over 128 separate 8-bit lanes. On the implementation level, it leads to

vectorized decoding functions that deliver a vector of 1024 tuples at-a-time, in sometimes as little as 17 CPU cycles (an astonishing 70 values per CPU core cycle).

Heterogeneous ISAs. In order to deal with concurrently existing generations of x86 SIMD hardware, as well as ARM, where AWS Graviton1-3 and Apple M1-2 support 128-bits NEON, and Graviton3 also supports SVE; and other ISAs for POWER and RISC-V, we define a **simple instruction set**[1] on `FLMM1024` that is easily supported by the common denominator of all SIMD instruction sets. While it is out of scope in this chapter, we think `FLMM1024` instructions on the FastLanes layout can also map efficiently to GPUs and other future data-parallel hardware (such as TPUs).

Decoding dependencies. Decoding RLE has an intrinsic control-dependency, as it needs a loop for emitting repeated values; but SIMD does not support control-instructions. DELTA decoding has an intrinsic data-dependency between subsequent values, which in SIMD are located in adjacent lanes; yet instructions with lane-dependencies are much slower. We tackle the latter problem by **reordering the column** using a technique we call "transposing", such that all lanes handle completely independent DELTA sequences. We then remap RLE to a combination of DELTA and DICT encoding, that leverages this very efficient DELTA decoding kernel.

Layouts that depend on lane-width. Previous work [53–60] studied data encodings in isolation, but here we also look at the system context, i.e. table scans of multiple columns. When the optimal layout depends on a specific lane-width (8, 16, 32, 64 bits), this is problematic in that context. In table formats, different columns will store different value distributions which get bit-packed using different bit-widths and get decoded into types that fit different lane-widths. Our idea of transposing also runs into problems in this regard. Naively applied, it would lead to different column reorderings inside the same table. Therefore, we invented a very specific reordering of 1024 tuples that suits all possible lane-widths. This we call the **Unified Transposed Layout**. The gist of this reordering is to organize 1024 values in eight 8x16 transposed blocks, and to put these eight blocks in the order “04261537”. We will explain why this order works well with any column-width.

Table 2.1: Challenges to efficient data-parallel decompression in big data formats, and how **FastLanes** tackles them.

Challenge	FastLanes Solution
many SIMD widths	target a virtual FastLanes `FLMM1024` SIMD register
heterogenous ISAs	`FLMM1024` uses simple operators, present in all ISAs
decoding dependencies	reorder (transpose) columns to break dependencies
1 layout per lane-width	same Unified Transposed Layout for all lane-widths
keeping code portable	no intrinsics: use scalar code & auto-vectorization
LOAD/STORE-bound	vectorized execution & fused unpacking+decoding

Keeping code portable. The simple design of the `FLMM1024` Fastlanes 1024-bits instruction

[1]The idea is similar to [52] but as SIMD width interacts with data layout, we design for a concrete 1024-bits width. Rather than trying to cover all ISAs in intrinsics, our simple `FLMM1024` instruction set has a scalar implementation that gets auto-vectorized.

set allows to implement it in scalar code that uses `uint64` registers and operations. This portability also allows low-end CPUs that do not support any SIMD and that may even have 32-bits registers and memory addressing (but where compilers emulate 64-bits arithmetic) to also run FastLanes rather *efficiently* to their standard. On 64-bits CPUs, scalar FastLanes code achieves SIMD-like acceleration when handling small lane-widths (i.e. 8-bits gets 8x faster using 64-bits scalar). We find it remarkable that SIMD-friendly ideas like interleaving and transposing accelerate our scalar code, rather than slow it down. Last but not least, modern compilers can auto-vectorize our scalar code-path without loss of performance, **avoiding the need for SIMD intrinsics**, thus reducing technical debt and further making FastLanes future-proof.

Avoid getting LOAD/STORE-bound. We propose to use FastLanes decoding in vectorized execution, where the compressed data is read from RAM and gets decoded into 1024-value arrays, which are then processed from the CPU caches by the query pipeline. This reduces memory traffic by the compression ratio (often 2-3x). Further, most CPU time will be spent on the operators in the query pipeline, so scans run at much lower than the maximum decoding speed, further reducing bandwidth pressure. Sequential scans will trigger memory hardware prefetching, so good throughput can be reached. All this reduces the probability to be LOAD bound.
However, as FastLanes decoding is much faster than previous LWC schemes, and can achieve astonishing speeds, the decoding functions can become STORE bound, even when storing just into L1 cache. We show that **fusing our bit-unpacking kernels with the decoding kernels** for FOR/DELTA/RLE/DICT benefits performance, as this saves an intermediate STORE+LOAD.

2.1.2 Outline

The remainder of the chapter is organized as follows. In Section 2 we explain these contributions in more detail, helped by a series of figures in visual language. First we explain 1024-bits interleaved bit-unpacking. The Unified Transposed Layout of FastLanes is motivated and explained around DELTA decoding. We further discuss efficient decoding of RLE exploiting this foundation. We follow-up in Section 3 with an evaluation of decompression performance of FastLanes bit-unpacking and DELTA and RLE decoding on all major hardware platforms. We also perform an end-to-end query execution benchmark based on Tectorwise [61] showing that using FastLanes decoding, instead of just an uncompressed in-memory array scan, can make a query faster. In Section 4, we discuss related work, covering the main differences between FastLanes and the state-of-the-art using both explanatory figures and micro-benchmarks. We conclude the chapter and discuss future work in Section 5.

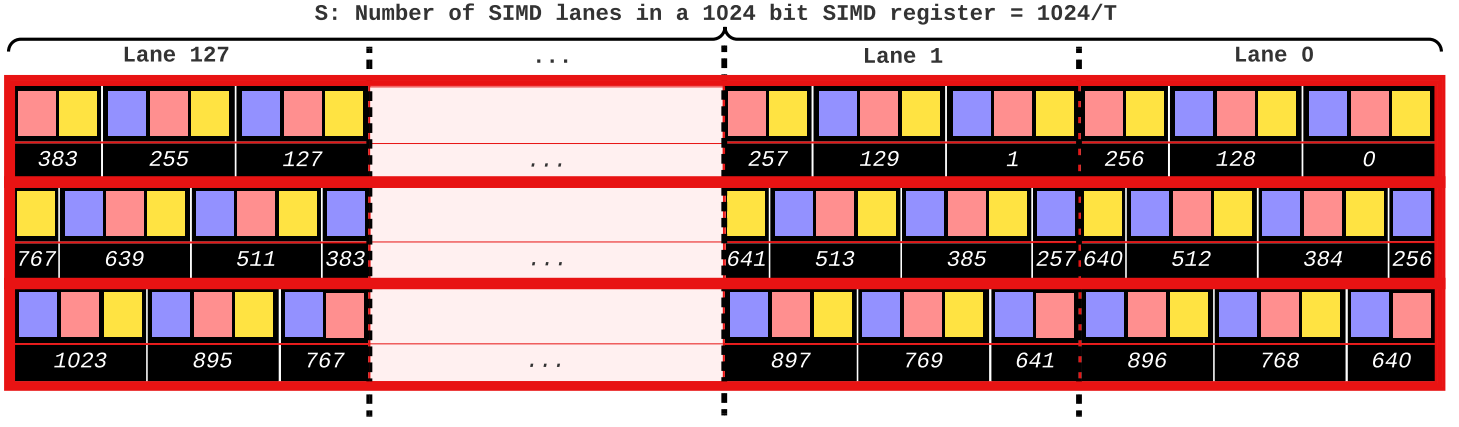


Figure 2.1: The 1024-bit interleaved layout. B 3 adjacent FLMM1024 words (red boxes, shown top-down) store 1024 values. Black bars indicate bit-packed values with their logical positions in the column: logically subsequent W 3-bit encoded values are round-robin spread into S 128 lanes of T 8-bits. In the first word, only the first two bits (yellow,pink) of the value at position 256 fit, so it is continued in the second word (blue bit). The value at position 640 is also split. This happens in all lanes.

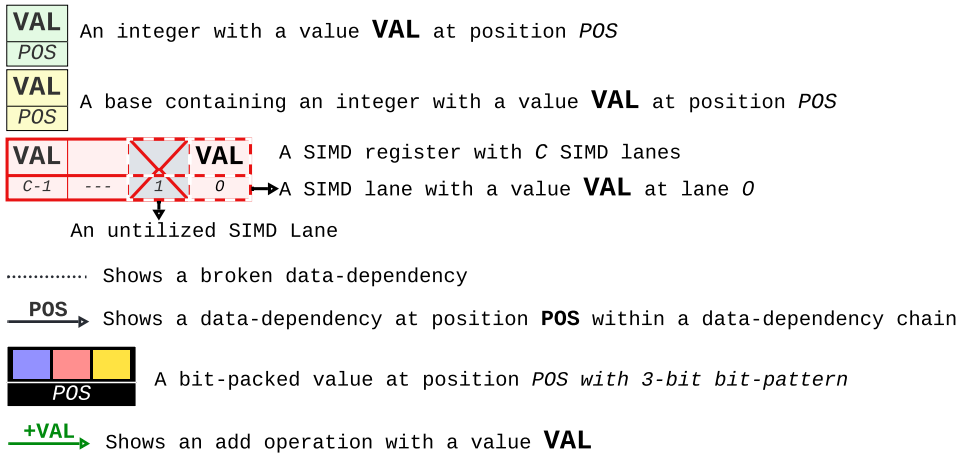


Figure 2.2: Legenda for our visual explanations.

2.2 FastLanes

In order to explain the FastLanes compressed data layout, we make extensive use of drawings in the visual language introduced in Figure 2.2. We now explain the main FastLanes features in detail.

2.2.1 Many SIMD widths

Over the past three decades, SIMD register widths in x86 CPUs have doubled three times from MMX (64-bits) to SSE1-4 (128-bits 1999), AVX/AVX2 (256-bits, 2008) and AVX512 (512-bits, 2015). A next doubling is not imminent, but we do see GPUs - and Apple CPUs - adopting a 1024-bit cache-line, which facilitates such a move.

Existing SIMD decoding algorithms and their data layouts typically target a specific register width. Consider the *4-way* interleaved layout [54], which distributes bit-packed tuples among 4 SIMD lanes. This layout avoids expensive cross-lane `PERMUTE` or `BITSHUFFLE` instructions, needed if bits would be packed consecutively. While being efficient for unpacking four 32-bits values CPUs on 128-bit SIMD registers, this layout does not have enough parallelism for 256-bits or 512-bits registers. In response, the 8-way and 16-way interleaved formats were proposed [62], which are all different.

To preempt changing data formats when some ISA starts to support a wider SIMD register, FastLanes targets a still-not-existent register width, concretely 1024-bits.[2] One should note that as long as – expensive – lane-crossing operations are avoided, it is trivial to support data layouts designed for a wider register without performance penalty on a thinner SIMD register; just by using multiple identical thinner instructions working on adjacent data. The

[2]We could have picked 2048 or 4096 as well; we chose to be conservative as the layout chunk-size grows with it: a chunk of 1024 W (bit-width) encoded values fit in exactly W `FLMM1024` registers. Larger chunk-sizes lead to worse compression ratios since the bit-width for bit-packing depends on the value-domain of a chunk (an exception mechanism to remove outliers can help to contain this problem). They also lead to an increased minimum vector-size, i.e. access granularity, imposed to the scan subsystem.

reverse is not true: supporting thin layouts on wide registers typically leads to lack of parallel work and unused lanes or expensive compensating actions such as `PERMUTE` and `BITSHUFFLE`. Figure 2.1 shows the interleaved bit-packed layout in the example case of integers that can be encoded in 3 bits (W=3). To maximize decoding performance we use the smallest lane-width that fits that, i.e. 8-bits (T=8), and therefore we have 128 (S=1024/T=128) lanes in our `FLMM1024` word. Note that bit-packing is a building block that is used in all encodings and can optionally be combined with an exception-handling technique (such as "Patching" [47]), to handle - in this case – infrequently occurring values that do not fit 3 bits.


```
FLMM1024* // A pointer to 1024-bit word memory.
FLMM1024  // A variable of size 1024-bit

// Load 1024 bits from memory address ADR
FLMM1024 LOAD<T>(FLMM1024* ADR);

// Store 1024 bits from REG into memory address ADR
void STORE<T>(FLMM1024* ADR, FLMM1024 REG);

// For all T-bit lanes i in REG, return (i & MASK) << N
FLMM1024 AND_LSHIFT<T>(FLMM1024 REG, uint<T> MASK, uint8 N);

// For all T-bit lanes i in REG, return (i & (MASK << N)) >> N
FLMM1024 AND_RSHIFT<T>(FLMM1024 REG, uint<T> MASK, uint8 N);

// For all T-bit lanes (a,b) in (A,B), return (a & b)
FLMM1024 AND<T>(FLMM1024 A, FLMM1024 B);

// For all T-bit lanes (a,b) in (A,B), return (a | b)
FLMM1024 OR<T>(FLMM1024 A, FLMM1024 B);

// For all T-bit lanes (a,b) in (A,B), return (a ^ b)
FLMM1024 XOR<T>(FLMM1024 A, FLMM1024 B);

// For all T-bit lanes (a,b) in (A,B), return (a + b)
FLMM1024 ADD<T>(FLMM1024 A, FLMM1024 B);

// For all T-bit lanes, return VAL
FLMM1024 SET<T>(uint<T> VAL);
```

Listing 1: FastLanes simple SIMD instruction set, with `FLMM1024` 1024-bit registers and `T`-bit lanes; `T` $\in \{8, 16, 32, 64\}$. It can be trivially mapped onto any existing SIMD ISA, or to scalar code using `uint64`. ISAs with narrower registers use multiple identical instructions over multiple registers and adjacent memory to simulate 1024-bit width.

2.2.2 Heterogeneous ISAs

When new SIMD ISAs are introduced, we often see two kinds of asymmetries: (i) new operators that did not exist in a thinner ISA are introduced, or (ii) a wider register is introduced, but not all operators existing on thinner registers are (initially) supported on the wider register. Data layouts that depend on these operators are then problematic to support

efficiently on all plausibly in-use hardware platforms, certainly for data systems that are distributed as binaries (pre-compiled).
Recently, ISA heterogeneity has significantly increased as ARM CPUs have become popular both on servers (AWS Graviton2,3) and with end-users such as data scientists (Apple M1,2); which bring their own subsets of NEON as well as SVE.
In order to support heterogeneous ISAs, FastLanes only uses simple operators, such as load/store, left/right-shift, and/or/xor, addition and set instructions; supported for all lane-widths, $T \in \{8, 16, 32, 64\}$ as shown in Listing 1. This instruction set can be trivially mapped to intrinsics in all previously mentioned thinner ISAs, just by using multiple identical instructions on independent registers or adjacent memory locations, to reach the 1024-bit width of our virtual `FLMM1024` register. The extreme example of this is our Scalar_T64 code-path, which relies on 64-bits integers (`uint64`):

```
struct { uint64 val[16]; } FLMM1024; // 16*uint64 = FLMM1024

FLMM1024 AND<8>(FLMM1024 A, FLMM1024 B) {
  FLMM1024 R;
  for (int i = 0; i < 16; i++) R.val[i] = A.val[i] & B.val[i];
  return R;
}
```

Listing 2: Example of a vectorized AND operation using `FLMM1024`. Each of the 16 `uint64` values is processed independently, modeling a 1024-bit SIMD register.

As a detail, we note that we combined the shift instructions with AND functionality. In bit-packing, these two operations are typically followed by each other anyway, so in those cases, the combined instruction is a shorthand. Another reason to introduce this shorthand is our Scalar_T64 code-path that manipulates `uint64` values. As shown above, we can support for instance eight 8-bits lanes using instructions on `uint64`. However, shift instructions on `uint64` could transport bits from one lane into another, something that is guaranteed not to happen in SIMD instructions. But, by performing the AND before shifting in such a way that bits that would cross a lane are masked out, this problem can be prevented by manipulating the (constant) mask value, at no additional cost.[3]
Figure 2.3 shows the implementation for unpacking 3-bit (W=3) codes into 8-bit (T=8) integers. Rather than writing such code by hand, we generate it statically for all $1 \leq W \leq 64$, $T \in$ {8,16,32,64} where W T (116 pre-compiled functions that each deliver a vector of 1024 values). Figure 2.4 shows the algorithm in action: in 10 instructions, 384 values are unpacked. On this unpack kernel, Intel AVX512 CPUs get to the astonishing speed of 70 values per cycle = 140 billion values per second on one 2GHz core. Given 3-bits per value this requires 52GB/s - close to RAM bandwidth limit. In reality, however, a query pipeline spends at least a few cycles per value in its operators, so the pipeline runs 100x slower; but

[3]Note that cross-lane bit-spilling is also a risk in the ADD operator. However, as SIMD ISAs do not support overflow detection, usage of SIMD ISAs for summations already requires the use of overflow prevention techniques in order to ensure correctness. Hence for ADD we can assume that overflow does not happen.

```
uint<8> MASK1 = (1<<1)-1, MASK2 = (1<<2)-1, MASK3 = (1<<3)-1;
FLMM1024 r1, r0;
r0 = LOAD<8>(in+0);
r1 = AND_RSHIFT<8>(r0,0,MASK3); STORE<8>(out+0,r1);
r1 = AND_RSHIFT<8>(r0,3,MASK3); STORE<8>(out+1,r1);
r1 = AND_RSHIFT<8>(r0,6,MASK2);
r0 = LOAD<8>(in+1);                          STORE(out+2,OR<8>(r1,
       AND_LSHIFT<8>(r0,2,MASK1)));
r1 = AND_RSHIFT<8>(r0,1,MASK3); STORE<8>(out+3,r1);
r1 = AND_RSHIFT<8>(r0,4,MASK3); STORE<8>(out+4,r1);
r1 = AND_RSHIFT<8>(r0,7,MASK1);
r0 = LOAD<8>(in+2);                          STORE(out+5,OR<8>(r1,
       AND_LSHIFT<8>(r0,1,MASK2)));
r1 = AND_RSHIFT<8>(r0,2,MASK3); STORE<8>(out+6,r1);
r1 = AND_RSHIFT<8>(r0,5,MASK3); STORE<8>(out+7,r1);
```

Figure 2.3: Interleaved bit-unpacking kernel in FLMM1024 SIMD for $T8$ and $W3$. We use code-generation to create such implementations for all combinations of T and W (W T).

with this unpacking speed the decompressing scan is practically free.

2.2.3 Dealing with Sequential Data Dependencies

Dependencies between subsequent values are SIMD-unfriendly since adjacent values end up in adjacent lanes. Figure 2.5a shows that the default layout (one value after the other) has this problem. The additions needed for DELTA decoding are lane-crossing operators: suppose the values in in Figure 2.5b are 32-bits, then adding the values at position 0 and position 1 correspond to different lanes (if e.g., positions 0-3 were loaded in a 128-bit SIMD register). In these figures, the yellow boxes indicate *base* values. These bases provide entry-points to start DELTA decoding. In FastLanes, we allow to start decoding with a granularity of 1024 tuples. Base values would be found in the header of a compressed columnar block. But, rather than having one base per vector, Figure 2.5c shows the idea of having four bases. This allows to start decoding at positions 0,4,8 and 12. It still does not solve the lane-crossing problem, though. Figure 2.5d shows the "transposed" layout, that **stores the values out-of-order**. The order for the first 16 values here is 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15. Figure 2.5e show this leads to optimal 128-bits SIMD processing: only 4 additions are needed.

We call this re-ordering a transposition because the idea is to cut up the value column in SIMD register-sized chunks and put these chunks vertically under each other, as shown in Figure 2.5f. In case of our 1024-bits FLMM1024 register, this means that this matrix has exactly T rows and S columns; where T is the value (=lane) bit-width and S is the amount of such values in a register.

We argue that changing the tuple order is not problematic in the database scan context. Relational algebra is set-based and query operator semantics typically do not depend on order, so if the tuples arrive perturbed from insertion order, they can usually be processed in whatever order they arrive. Even if the order matters for the query result or operator

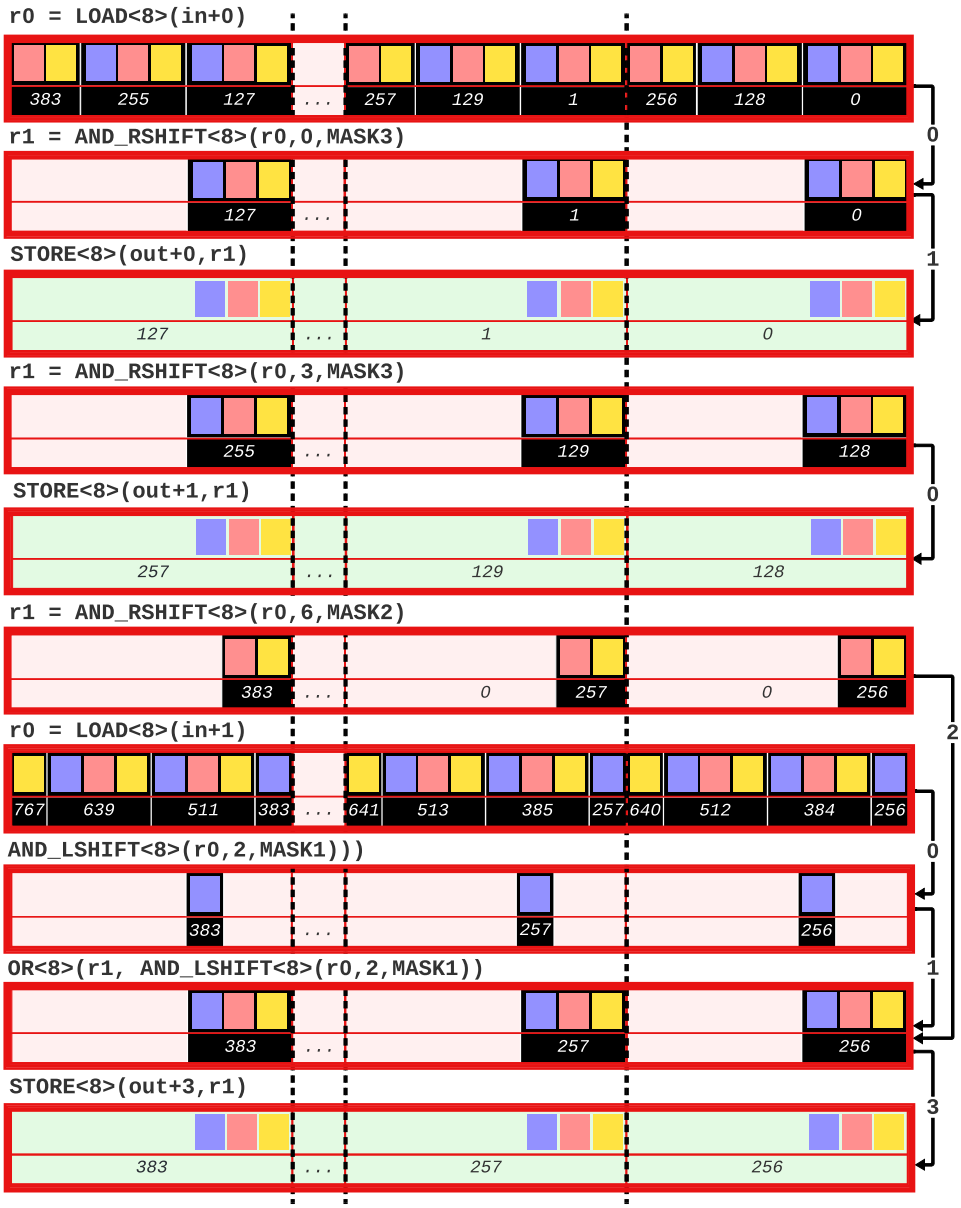
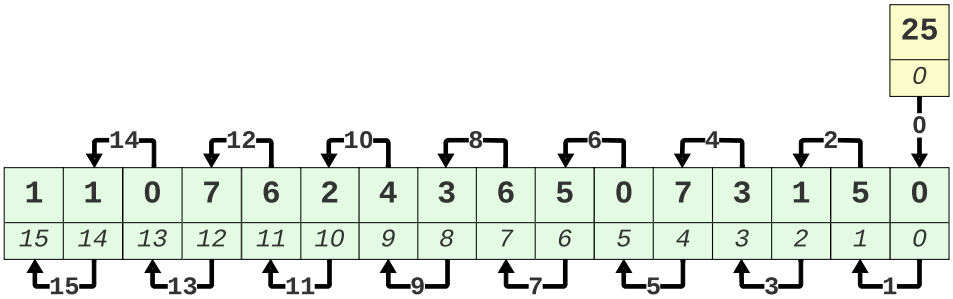
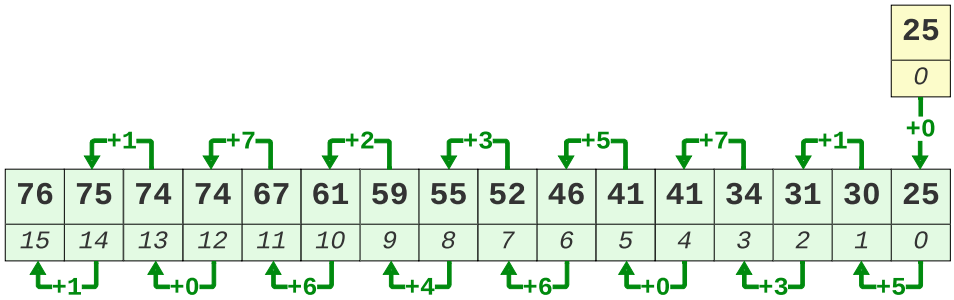


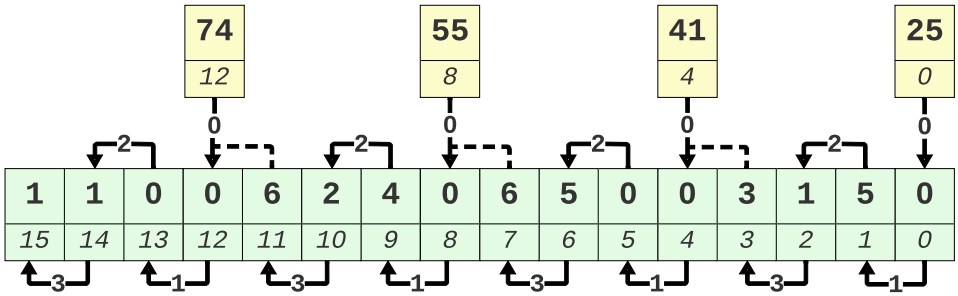
Figure 2.4: Lines 3-8 of Listing 2 in action: ten `FLMM1024` instructions bit-unpack the first 384 3-bits codes into 8-bit integers. The investment in interleaving of bits leads to perfectly sequential unpacked integers using few simple instructions.



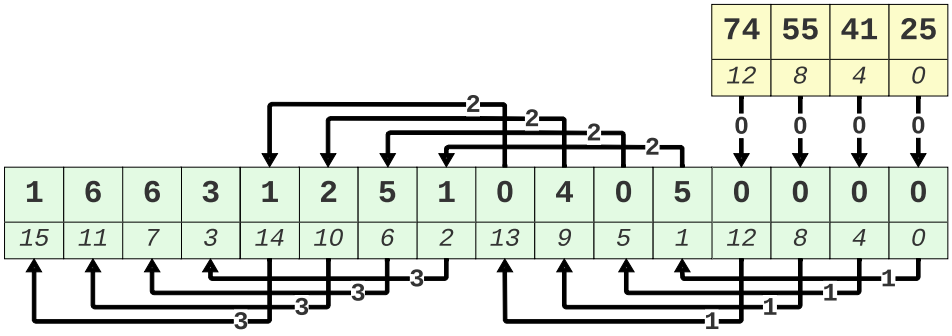
(a) Default DELTA layout with data dependencies on arrows.



(b) The process of decoding DELTA-encoded data (green arrows).

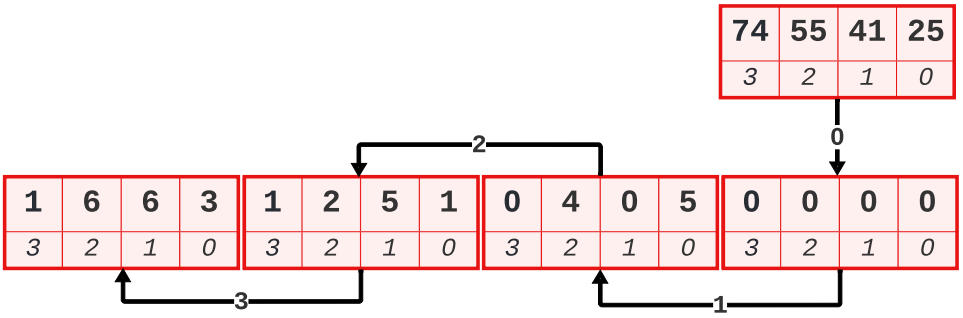


(c) Example DELTA layout with multiple bases.

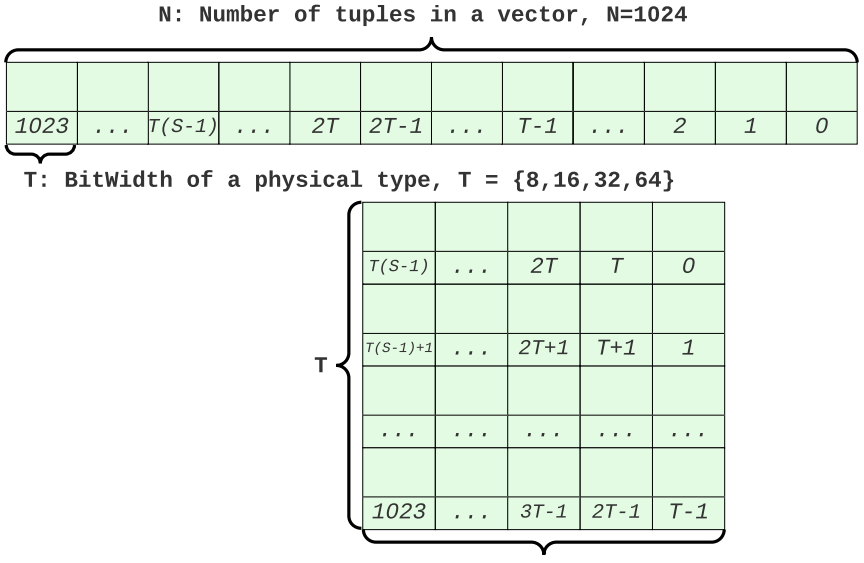


(d) Transposed data layout.

Figure 2.5: (a)–(d): Part 1 of the Transposed Data Layout illustrations.



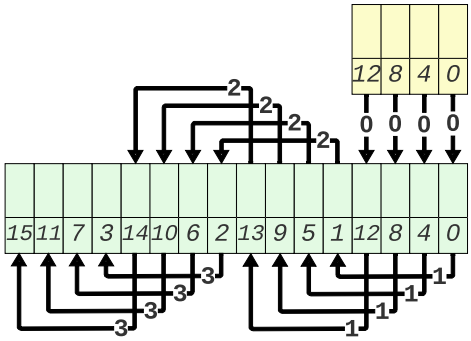
(e) SIMD-friendly DELTA decoding on the transposed layout.



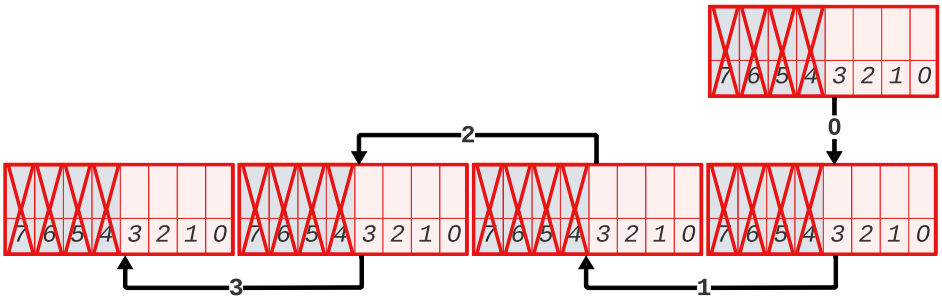
(f) Transposed Layout: value order depends on widths S & T.

Figure 2.5: (e)–(f): Remaining Transposed Data Layout illustrations. Idea: reorder column values to make data dependencies SIMD-friendly.

semantics, the original order could be restored or encoded in a *selection vector*. While the presence of a selection vector can slow down operations, it can often be avoided: vectorized query executors typically have an optimization where simple arithmetic operators (that cannot raise errors) will ignore (identical) selection vectors on all parameters, if many tuples are still in play, executing the operation on all values, at much lower per-value cost thanks to full sequential access (and SIMD).



(a) Reordering from Figurefigure 2.5d applied to a half-width column.



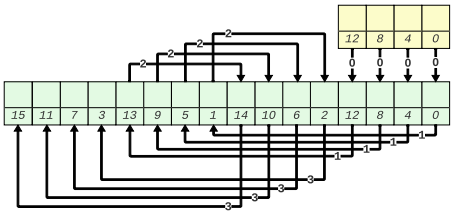
(b) Eight independent operations are needed, but this layout provides only four—resulting in underutilized SIMD lanes.

Figure 2.6: The transposed layout and resulting value reordering, while effective for one data type, are not suited for narrower types.

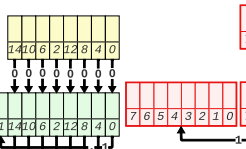
2.2.4 The Unified Transposed Layout

In our Transposed Layout, the order of the tuples depends on T. This creates a problem for database scans: relational tables consist of *multiple* columns and different columns will have different widths. However, when we reorder tuples, we should use the same order for all columns, because a scan needs to create a consistent stream of tuples.[4] Figure 2.6 shows that when we apply the reordering from Figure 2.5d to a data type of half the width, there is not enough independent work for the thinner type. In our example, the wide data-type was

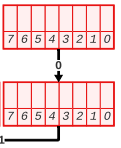
[4]Even if a query processor would be able to work with column vectors that each have a different value order, e.g., by accompanying each with their own selection vector that restores order; this would likely carry performance penalties due to the indirect memory access needed and reduce the applicability of our format to systems that could do this. Therefore we enforce the ability to retrieve all column data in the same order.



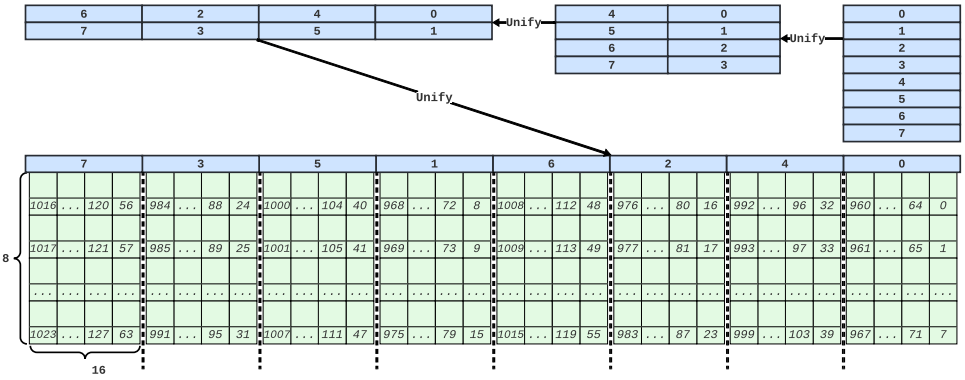
(a) This reordering is also SIMD-friendly for the wide type, but..



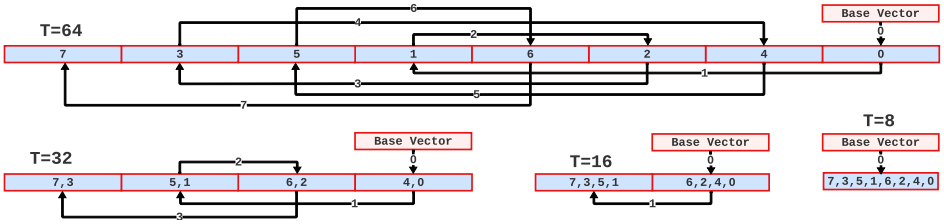
(b) ..applied to the thin type..



(c) ..is also SIMD-friendly for that.



(d) From the 64-bits transposed layout with 8x1 tiles of 8x16 values (upper right), we unify to 4x2 tiles for 32-bits types, then to 2x4 for 16-bits and 1x8 for 8-bits, in 04261537 tile order. The final value order is 0,64,..,960,32,96,..,992,..,..,63,127,..,1023 (top-to-bottom, repeated right-to-left)



(e) The 04261537 tile order is SIMD-friendly for all lane-widths. Note: the numbered blue-red boxes here are abbreviations for a 8x16 tile, and imply 8 SIMD ADD operations each during DELTA decoding. This layout benefits all encodings with dependencies (i.e., also RLE).

Figure 2.7: Unified Transposed Layout: (a)-(c) idea of order unification, (d) how our unified approach arrives at the 04261537 order (blue) of 8x16 tiles (green) and the final value order (green), (e) how it provides data-parallelism for all possible lane-widths. Notably, FastLanes does not only store each sequence of 1024 tuples permuted in this reordering, but the individual columns are usually also encoded with some LWC scheme (DELTA, FOR, DICT, RLE), which involves bit-packing using 1024-bit interleaving (Figure 1). So the eventual bit-sequences stored are humanly hard to grasp. However, decoding the values requires only regular and astonishingly fast calculations that are completely data-parallel.

32-bits such that 4 values fit a 128-bits SIMD register. So when putting a column of 16-bits integers in that order, we see that we only can take advantage of four lanes, instead of 8. In this case, the problem can be solved by just using a different ordering, shown in Figure 2.7a-c, that works well with columns of both widths.

Our Unified Transposed Layout provides a generic solution to this problem for all lane-widths. The basic building block are transposed tiles of 8x16 values. We have eight such tiles for each vector of 1024 tuples. For the widest 64-bits type, each row in the tile is one `FLMM1024` register, making it a suitable format to process one tile-at-a-time: for DELTA decoding, the 8 rows are processed using 8 `FLMM1024` ADD64. In case of 32-bits values, however, one row occupies half a register, so we need to group two independently processable tiles together in one register. This is done by taking the lower half of tiles 0-7 and placing them to the left, arriving at 4 rows of 2 tiles. This process repeats for 16-bits and 8-bits, arriving at a single row of 8 tiles in the 04261357 ordering (blue). The complete value ordering for all 1024 tuples is shown in green.

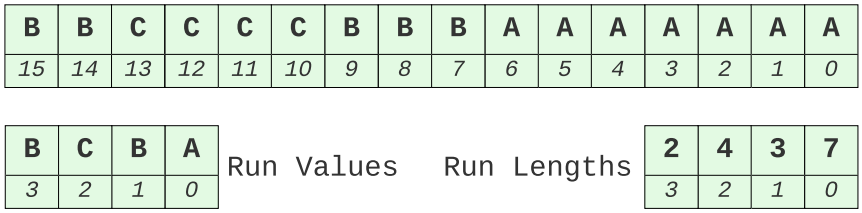
One can ask if 04261357 is the only ordering (starting at 0) that is suitable for DELTA decoding. We want to start at 0, because for 64-bits values we compute on data from one tile at-a-time, starting at tile 0; and for 64-bits data, the header thus holds bases for tile 0 only (see Figure 2.7a-b with base values in yellow). Beyond starting at 0, the second desirable property is that for processing tiles in SIMD operations, we need the subsequent operations to touch *directly subsequent* tile numbers in the same SIMD lane position.

Now the proof. Considering 16-bits values, where four tiles fit the SIMD register width, and given that 0 is first; we see that 1 must be in fourth position (as it must be subsequent in 0xxx→1xxx). In fact, the only way to get subsequent numbers in the two halves of the ordering is to have all even numbers first, and the odd numbers later. Now, considering 32-bits data types, where data from two tiles is processed at-a-time, the ordering should start with 04. Because, if we would start with 02, then after 02→13, the next SIMD operation should be on 24, but tile 2 was already processed. The other even choice 06 runs out of work, as after 06→17 there is no tile 8. As the first pair is 04, the third pair must be 15, and this fixes the second pair to 26 and the final pair to 37; so we arrive at 04261537 as the only ordering with the desired properties. Figure 2.7e shows that for 8-bits types, DELTA decoding processes: *bases* → 04261537 (drawn, as all layouts, right-to-left in our Figures). For 16-bits types the processing order is: *bases* → 0426 → 1537. For 32-bits it is: *bases* → 04 → 15 → 26 → 37. For 64-bits: *bases* → 0 → 1 → .. → 7.

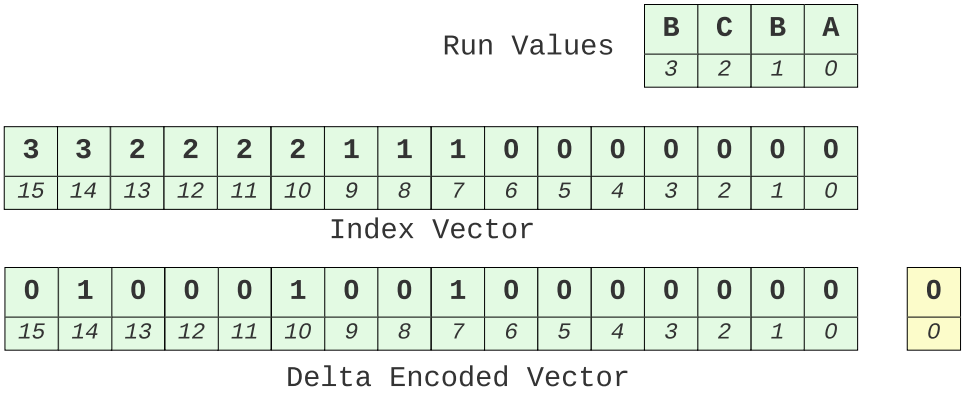
FastLanes-RLE. Value sequences get Run Length Encoded in classic RLE as *(value,length)* tuples. Decoding requires two nested loops: one that iterates over the tuples, and inside, one that iterates over *length*; while writing out the *value*-s. A loop is by definition scalar, and the inner loop will suffer from branch mispredictions on short lengths. The best SIMD acceleration so far for RLE works when run-lengths are large, such that the uncompressed run is very significantly larger than the SIMD register. In this case, one can set all lanes of a SIMD register to the constant *value*, and reduce the amount of STORE instructions by the amount of lanes [63].

We propose a new scheme called Fastlanes-RLE, that maps RLE to DELTA and supports storage reordered in the Unified Transposed Layout. It targets systems like Velox [50] and DuckDB [51], that prefer to represent decoded RLE as compact in-flight Dictionary vectors; rather than full/eager decompressed vectors. The twist here is that the Dictionary is the Run

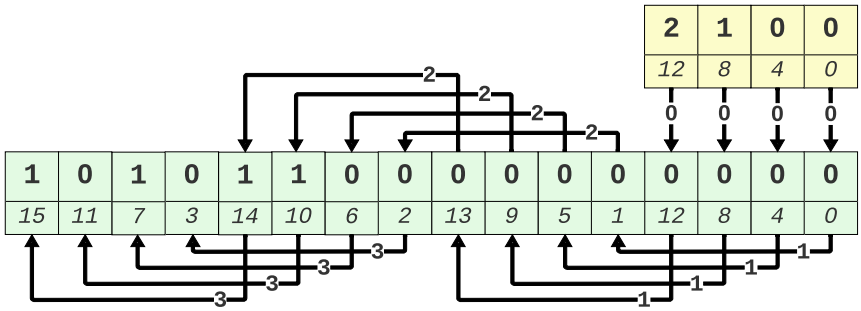
Value vector from RLE, and hence may contain duplicates. The Index Vector monotonically increases by one, whenever a new run starts. FastLanes-RLE uses 16-bit indexes for vectors with many short runs and 8-bits otherwise. These Index Vectors are DELTA encoded using only 1-bit per value. Base storage in the 8-bit case can use 3-bit bit-packing, adding .375 bits of storage per value, making the compression ratio better than classic RLE, up to average run-lengths of 12. For longer average run-lengths, we should use 0-bit DELTA encoding, that memsets the Index Vector to 0, and where the 1-s are inserted by an exception mechanism (we will cover such mechanisms in follow-up work).



(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 2.8: FastLanes-RLE: a fast and compact encoding scheme targeting in-flight partially compressed vectors [50, 64]

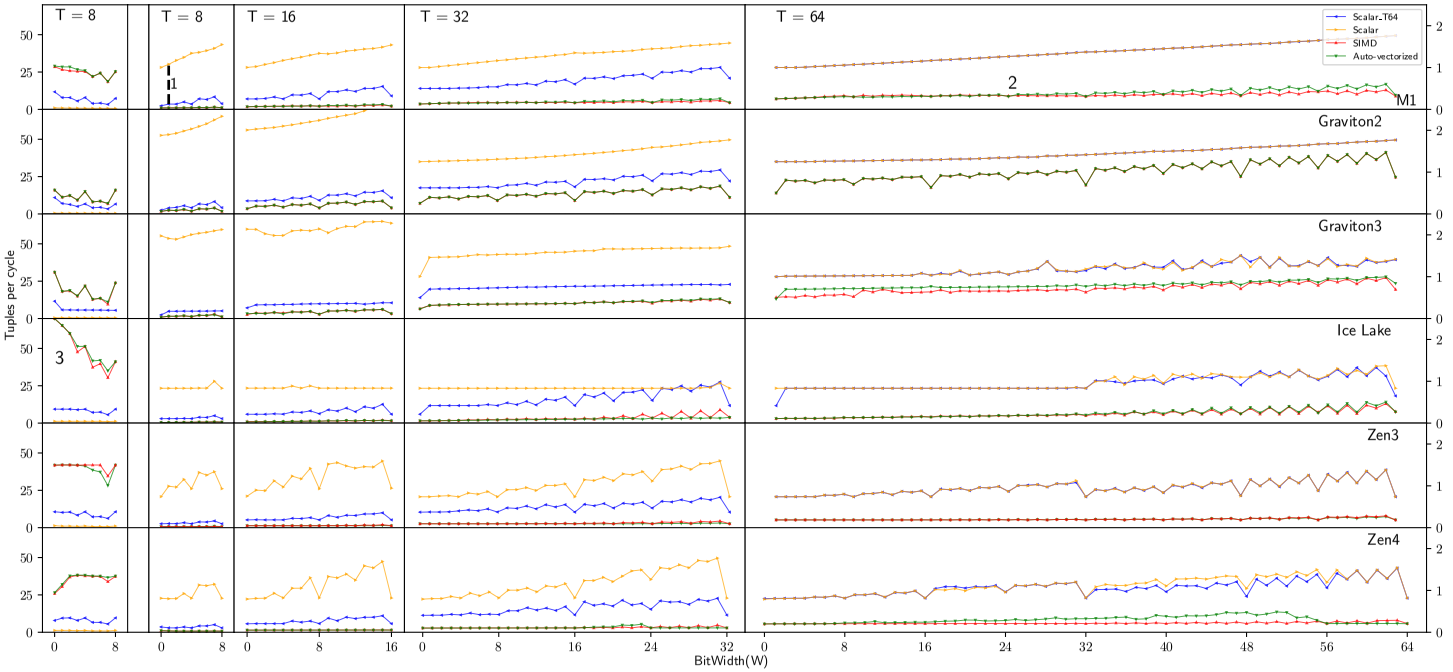


Figure 2.9: Bit-unpacking performance of the 1024-bit interleaved layout. (1) `Scalar_T64` uses 64-bit scalar registers as quasi-SIMD and beats naive `Scalar` up to 8x. (2) `clang++` auto-vectorizes `Scalar` perfectly, matching performance of explicit SIMD intrinsics. (3) Decoding can reach 70 tuples/cycle (T=8, W=1). Except in the leftmost box here (tuples/cycle), lower is better in all Figures (cycles/tuple).

2.3 EVALUATION

The C++ FastLanes library is released under a MIT license in open source and will be put in `github.com/cwida/FastLanes` on Jan 7.
We now experimentally evaluate the following questions:

(Q1) What is the absolute speed of the proposed FastLanes 1024-bit interleaved bit-unpacking?

(Q2) Does decoding performance scale with SIMD width, and how does it vary between the platforms listed in Table 2.2?

(Q3) Can scalar code profit from 1024-bits interleaving and the Unified Transposed Layout?

(Q4) What is the performance of the scalar implementation, and how well does compiler auto-vectorization compare with the use of explicit SIMD intrinsics?

(Q5) How does the proposed Unified Transposed Layout influence decoding performance, specifically for LWC schemes with sequential dependencies, such as DELTA?

(Q6) What effect on end-to-end query performance could the adoption of FastLanes have?

We also investigate the performance benefits of potentially fusing the implementations of bit-unpacking and decoding kernels. Note that in Section 2.4, we present additional micro-benchmarks while comparing FastLanes with related work.

2.3.1 MICRO-BENCHMARKS

We implemented bit-unpacking and decoding into T $\{8, 16, 32, 64\}$ result columns in 4 different ways: `Scalar`, `Scalar_T64`, `SIMD`, and `Auto-vectorized`. The `Scalar` code unpacks/decodes one `uint`T value at-a-time. The `Scalar_T64` implementation treats a `uint64` variable as a quasi-SIMD register consisting of 64/T lanes of T-bits.

Table 2.2: Hardware Platforms Used

Architecture	Scalar ISA	Best SIMD ISA	CPU Model	Frequency
Intel Ice Lake	x86_64	AVX512	8375C	3.5 GHz
AMD Zen3	x86_64	AVX2 (256-bits)	EPYC 7R13	3.6 GHz
AMD Zen4	x86_64	AVX512	Ryzen9 7950X	4.5 GHz
Apple M1	ARM64	NEON (128-bits)	Apple M1	3.2 GHz
AWS Graviton2	ARM64	NEON (128-bits)	Neoverse-N1	2.5 GHz
AWS Graviton3	ARM64	NEON (128-bits) SVE (variable)	modified Neoverse-V1	2.6 GHz

We used clang++ for our experiments. To make sure that our scalar code is not auto-vectorized, we explicitly disabled the auto-vectorizer for the `Scalar` and `Scalar_T` implementations by using: `-O3 -mno-sse -fno-slp-vectorize -fno-vectorize`.
The `SIMD` implementations use explicit SIMD intrinsics. Note that for ARM64, all SIMD implementations are based on NEON instructions. This is because our experiments on Graviton3 showed that SVE [65] is slower than NEON. Finally, the `Auto-vectorized`

implementation is the `Scalar` implementation, with the difference that auto-vectorization is not disabled.
These micro-benchmarks aim to characterize pure CPU cost and decompress a single vector 30M times; hence all data is L1 resident. We report CPU cycles per value (**lower is better!**), but for T=8 bit-unpacking also the reverse: values per cycle (cycles per value there get close to 0 and hard to discern). These measures make the results more meaningful to compare across platforms than elapsed time, as our hardware comes from different frequency classes (hi/mid/low end, consumer vs. server). We disabled CPU turbo scaling features where present to make clock normalization stable.

Bit-unpacking. Figure 2.10 we see that the 1024-bits interleaving of packed data does not even hinder `Scalar` decoding: performance is equal to the naive "horizontal" (non-interleaved) bit-packed layout. But, only the interleaved layout provides the opportunity of decoding multiple lanes in parallel seized by `Scalar_T64`, making it 8x faster than `Scalar` on 8-bits values. As for **(Q1)**, Figure 2.9 shows the high speed of FastLanes decoding: thanks to SIMD it significantly outperforms `Scalar` across all platforms: 40x-70x for 8-bits, to 3x-4x for 64-bits types. Regarding **(Q2)**: we do see that Gravitons have weaker SIMD; which especially shows for 64-bits types. Apple M1 also has just 128-bit NEON, but clearly has more instruction level paralellism (ILP). Wider SIMD does not always equate more performance: despite supporting AVX512, Zen4 is not faster than Zen3. This is expected if the CPU executes one AVX512 instruction using two AVX2 (256-bits) units. The absence of dependencies and the opportunities for data-parallelism that FastLanes code exposes, make it profit from total CPU execution capability, which is the product of ILP and register width. Figure 2.9 highlights that (1) `Scalar_T64` is indeed $\frac{64}{T}$ times faster than Scalar for different Ts **(Q3)**; (2) clang++ can auto-vectorize our `Scalar` code, matching the performance of explicit intrinsics — denoted `SIMD` **(Q4)**; (3) FastLanes can decompress 70 tuples per cycle for 8-bits types **(Q1)**, where SIMD paralellism is maximal. Point (2) means that when incorporating FastLanes in future systems, we recommend just using the Scalar code paths; in fact for the kernels described in this chapter, just the `Scalar_64` code is enough. This result significantly enhances the future-proofness of FastLanes.

Unified Transposed Layout. We performed experiments for **(Q5)** regarding DELTA decoding for all six hardware platforms. Figure 2.11 shows that the Unified Transposed layout – the idea to reorder the tuples in order to break sequential dependencies – also benefits our `Scalar_T64` code-paths, that uses `uint64` scalar registers as if they were 8x8-bits, 4x16-bits or 2x32-bits SIMD registers. In terms of scalar performance, M1 tops Ice Lake clock-for-clock. Remarkably, Graviton and Zen3 are slower in scalar additions on 8- and 16-bits numbers than on 32- and 64-bits. The Gravitons again show weak SIMD. Performance can again be very high, like 40 tuples per cycle on the faster platforms for 8-bits DELTA. Most DELTA decoding will be on the larger datatypes (32-, 64-bits), but FastLanes-RLE (evaluated later) uses very fast on 1-bit decoding in a 16-bits lane.
As bit-unpacking and FastLanes decoding use dependency-free instructions, column contents do not influence performance at all. Only the bit-width matters, hence we evaluate all bit-widths.[5]

[5]Regarding (ordered) DELTA columns, we finally argue that subsequent query performance after decompression is not likely to be affected even if the tuple order is left transposed, since the permutation caused by transposing is

Fusing Bit-packing and Decoding. The 116 bit-unpacking kernels we generate for all bit-packing widths W and unpacked type-widths $T \leq W$ could possibly be fused with the decoding kernel for DELTA, FOR, DICT and FastLanes-RLE in a single kernels that do both unpacking and decoding. The benefit of fusing is that the STORE instructions that bit-unpacking ends with, and the LOAD instructions that decoding starts with, are saved. Figure 2.12 shows that fusing indeed improves the decompression speed.
In case of decoding into compressed vectors, fusing is not needed for DICT and FOR (decoding is just bit-unpacking in that case – therefore we do not micro-benchmark these schemes separately). For decoding DELTA into a compressed FOR vector, we can use fusing; what is then needed is to keep MinMax stats per vector, and subtract Min from the bases before decoding.

2.3.2 End-to-End Query Performance

We also ran a complete query pipeline, by integrating FastLanes in the experimental Tectorwise [61] vectorized query processor. We created a table `TAB` with a single column `COL` that has $10 * 2^{28}$ `uint32` integer values (10GB), and benchmarked the query `SELECT SUM(COL) FROM TAB` on our IceLake platform.
Figure 2.13 shows the performance of this query, depending on the domain of the values in the column, which is uniform-randomly generated from the domain [0-$2^W\rangle$. We run this unmodified Tectorwise query, that reads COL from an `uint32` array, and two modified versions (FastLanes and `Scalar`) that scan a compressed `COL` – which gets bit-packed in W bits per value. In all cases the data is RAM-resident. As for **(Q6)**, we thus see that reading from FastLanes typically makes a query **faster**, despite the decompression, because the query needs less RAM-bandwidth. Parallel execution increases the RAM bottleneck: with 8 threads we see up to 7x end-to-end performance improvement vs. uncompressed (and 4x vs. `Scalar`). FastLanes shifts the crossover point where queries get faster from data with a 4x compression ratio (`Scalar`) to almost any data.

within a 1024-vector only, and hence localized, such that any column order is largely preserved.

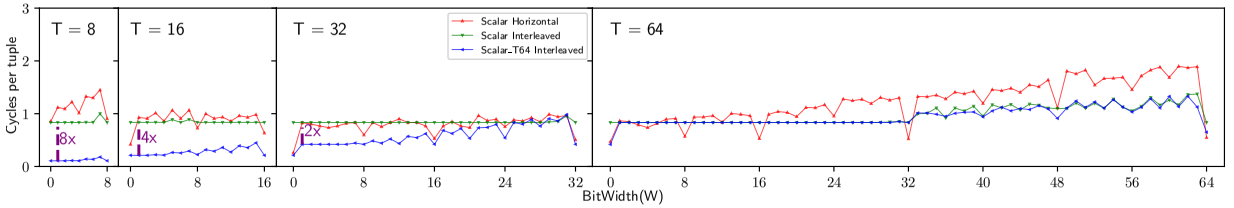


Figure 2.10: Horizontal vs. 1024-bit interleaved. `Scalar` bit-unpacking performance with 1024-bit interleaving is equal to the naive horizontal layout (red = blue). The bit-interleaving approach allows `Scalar_T64` (green) to get up to 8x faster (Ice Lake).

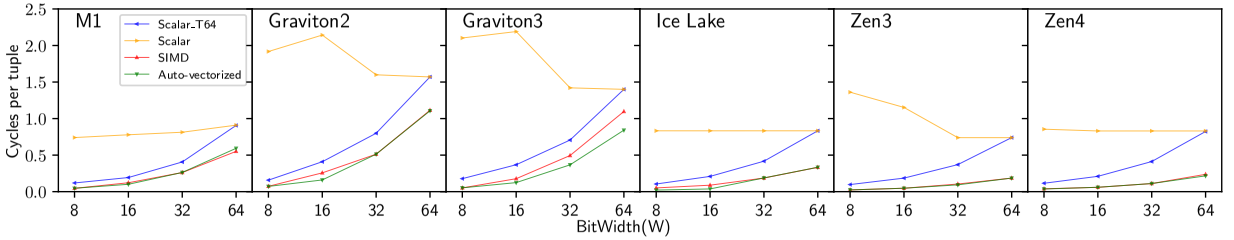


Figure 2.11: FastLanes DELTA decoding, for all bit-widths & platforms: very high performance for `Auto-vectorized`. Also, `Scalar_T64` profits from data-parallelism in the Unified Transposed Layout, whereas `Scalar` cannot and can be 40x slower than `SIMD`.

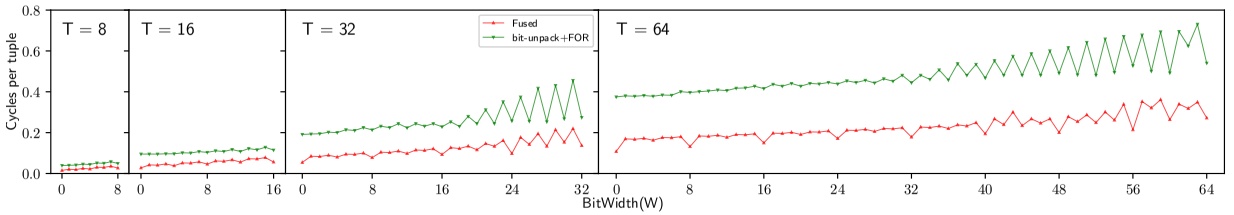


Figure 2.12: Fusing 1024-bit interleaved bit-unpacking with decoding (FOR) improves performance (Ice Lake).

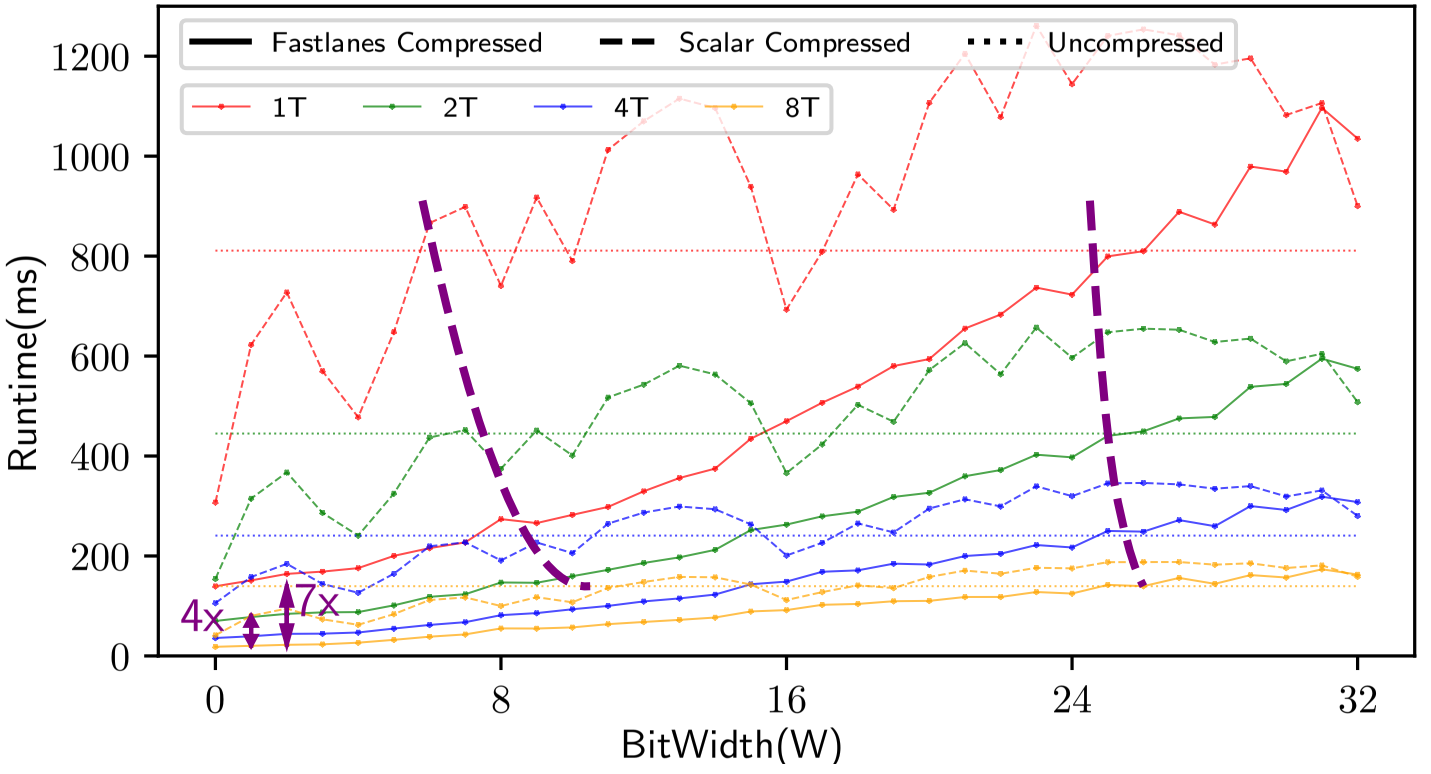


Figure 2.13: `SELECT SUM(COL) FROM TAB` runtime for various `COL` bit-widths and threads (T) on Ice Lake. The crossover point where decompressing scans (plots) outperform plain array scans (horizontal lines), moves from a minimal compression ratio of 4x (≈8bits) with `Scalar` decoding to just 25% compression (≈24bits) with FastLanes. Note that with higher thread counts, the crossover point (thick stripes) moves right a bit, as RAM bandwidth gets scarcer. FastLanes can then improve end-to-end performance up to 7x vs. uncompressed and 4x vs. scalar.

2.4 Related Work

For more than two decades, researchers have been trying to use SIMD instructions to improve the performance of database systems [66, 67]. Much of this effort has been made on SIMDizing the compression and decompression of data [53–60]. Surveys of these SIMDized compression schemes are [48, 63].

Bit-packing. Zukowski *et al.* propose to bit-pack 128 integers sequentially using the same bit-width [47]. Schlegel *et al.* call this layout horizontal [57]. Willhalm *et al.* propose a SIMDized bit-unpacking for the horizontal layout [68]. In addition to the horizontal layout, Schlegel *et al.* propose the k-way vertical layout [57], where each of the k consecutive bit-packed values are distributed among consecutive memory words. This vertical idea is also called interleaved layout, and we use that terminology in this chapter. This distribution allows to have bit-packed values in different SIMD lanes and avoids the extra `PERMUTE` instruction, required in the horizontal layout. Lemire *et al.* use the 4-way vertical layout (k=4) to SIMDize the bit-unpacking for 32-bit integers on CPUs with `SSE` registers [54]. Also, Habich *et al.* use 8-way and 16-way vertical layouts for AVX2 and AVX512 registers [62]. However, these layouts do not cover all challenges that have been discussed earlier in Table 2.1: these layouts are tied to a specific SIMD-width, they do not address the problem of sequential data dependencies in LWCs that work on the decoded data (such as DELTA), and do not address the issue of different data type widths in relation to that. Figure 2.14 shows that the 4-way layout becomes only slightly faster on AVX2 and AVX512 ISAs. On the other hand, the interleaved layout becomes respectively 2x and 4x faster on AVX2 and AVX512. This confirms that the 4-way layout cannot take advantage of wider registers, while the 1024-bit interleaved layout can.
In addition to the bit-packed layouts that focus on decompression speed, there are other bit-packed layouts that focus more on the filter scan. BitWeaving [69] and ByteSlice [70] are two examples of such layouts. BitWeaving proposes two novel bit-packed data layouts: HBP and VBP. These layouts allow using all the bit-parallelism of a SIMD register during the filter scan. HBP is more focused on supporting efficient lookup operations, while VBP provides a faster filter scan. ByteSlice tries to achieve both fast lookup and fast filter scan by applying all the BitWeaving techniques in the byte-by-byte manner instead of bit-by-bit. However, neither BitWeaving nor BitSlice provides a fast and efficient way to actually decompress data. Polychroniou *et al.* propose a SIMDized bit-unpacking for the VBP layout [71]. However, the reported performance of this layout is roughly 30x slower than our 1024-bit interleaved layout.

DELTA coding is an LWC that encodes a sequence of integers by replacing each integer with its difference to its preceding integer [72]. DELTA is typically used on top of bit-packing to reduce the number of bits required to represent values. While improving the compression ratio, DELTA decoding becomes a bottleneck in combination with bit-unpacking. Three approaches have been proposed to data-parallelize DELTA decoding: vertical computation [73], horizontal computation [74] [75], and the SIMDized tree computation [73]. Vertical computation is based on the SIMD `SCATTER/GATHER` instructions with non-sequential access pattern. Unfortunately, these instructions are costly and do not make decoding faster [73]. Horizontal computation reduces the complexity of DELTA decoding from $O(n)$ to $log(n)$. This is achieved by using the SIMD `SHIFT` instructions.

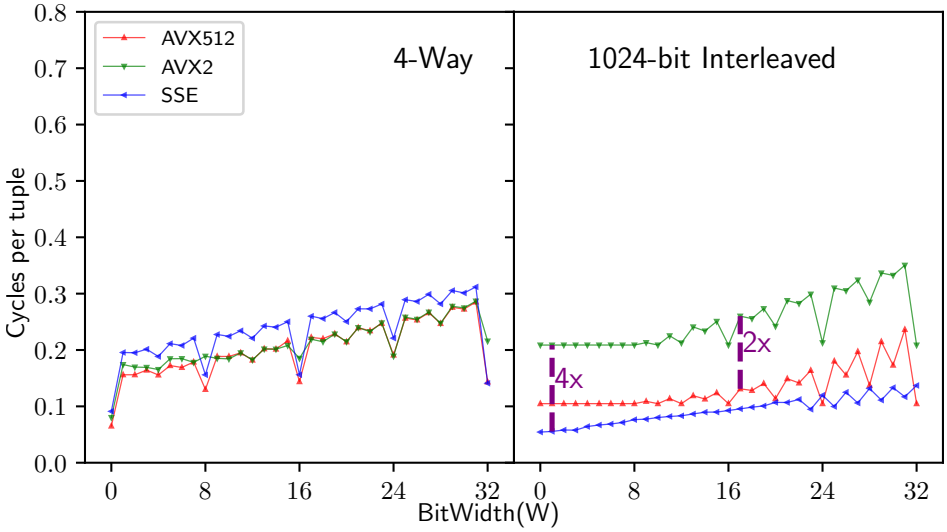


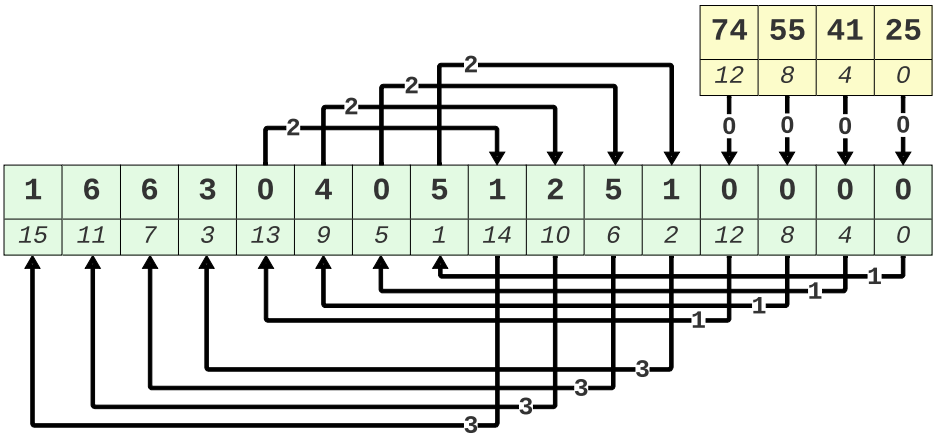
Figure 2.14: Bit-unpacking using the 4-way layout vs. 1024-bit interleaved layout, where T 32 (Ice Lake). The 4-way layout cannot take advantage of wide SIMD registers, with a performance penalty of 2x resp. 4x for AVX2 resp. AVX512.

However, these instructions do not exist in all ISAs, and it is costly to simulate them. Finally, the tree approach is based on Guy *et al.*'s work [76] and also relies on `SCATTER/GATHER` instructions [73].

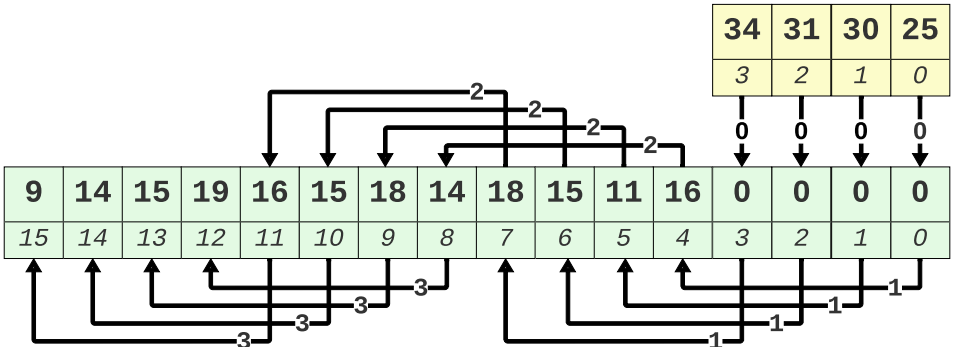
The SIMD implementation of horizontal computation can be considered state-of-the-art [73]. This implementation depends on the `SHIFT` instruction that shifts bits together arbitrarily times to the right. However, this instruction only exists for SSE registers. Zhang *et al.* propose to extend this implementation to AVX-512 by simulating the `SHIFT` instruction with two `SET`, and `ALIGNR` instructions [73]. This implementation needs 12 instructions for every 16 integers. Compared to FastLanes, we can see that this SIMDization does not address all the challenges mentioned earlier. First, data dependency still exists. Second, these implementations are not designed to support all SIMD ISAs.

Rather than SIMDizing the decoding part of the naive DELTA layout, several studies have focused on changing the data layout of DELTA. Lemire *et al.* [54] has proposed two approaches: `DM` and `D4`. The key idea behind these two approaches is to keep deltas between adjacent batches of values instead of adjacent values. As shown in Figure 2.15b, `D4` subtracts the values batch-wise, while `DM` (Figure 2.15c) subtracts the last value of the previous batch with the next batch. Although `D4` provides more data parallelization, the problem here is that the DELTAs are bigger because they are the difference between more distant values. In D4, the differences are 4x bigger, which reduces the compression factor typically by $\log_2(4)$, hence a factor 2. Unfortunately, to support ever wider SIMD registers, ever larger batches are necessary, increasing this overhead.

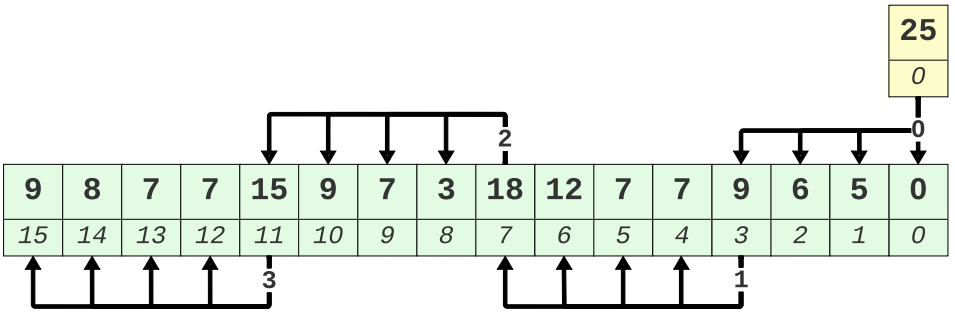
Another layout proposed to mitigate the issue of data dependency is the four cursors



(a) Unified Transposed Layout. The vector can be bit-packed using 3 bits per value as the maximum delta is 6.



(b) D4 data layout. The maximum delta is now 19. Therefore, 5 bits are required to bit-pack each value.



(c) DM data layout. The maximum delta is now 18. Therefore, 5 bits are required to bit-pack each value.

Figure 2.15: The Unified Transposed layout needs fewer bits than D4 and DM as it keeps DELTAs between subsequent values.

Table 2.3: Summary of all proposed approaches for SIMD DELTA decoding. Decompression Cost is the number of ADD instructions required to decode S values, while the Compression Overhead is the number of extra bits required.

Approach	**Decompression Cost**	**Compression Overhead**	**Shortcoming**
Scalar [72]	S	0	Data dependent
Four Cursor [48]	S	$\frac{4 \cdot T}{N}$	Data dependent
Vertical [73]	2	0	Random access
Horizontal [74]	$\log S$	0	Not efficient
Tree [73]	2	0	Random access
D4 [54]	1	$\log S$	Compression ratio
DM [54]	2	$\log \frac{(SS-1\cdots1)}{S}$	Compression ratio
Unified Transposed Layout	1	$\frac{1024}{S}$	-

layout [48]. The key idea is to keep more base values, so we can decode more values in parallel without dependencies. This layout was already shown in Figure 4c. Note that although we cannot use SIMD instructions to decode these four values simultaneously, it allows a wide-issue scalar CPU to achieve better ILP by working on four cursors inside one same scalar loop.
Figure 2.16 shows the performance of the DELTA decoding methods summarized in table 2.3. The performance of the horizontal methods is inconsistent, as important SIMD instructions are not available for all register- and lane-width combinations. Four-cursor improves Scalar a little. The Unified Transposed layout is by far fastest. It does increase the amount of base values per vector: from 1 to S (the amount of lanes, 1024/T). The bit-packed vector with deltas takes W*1024, and each base W bits, so the overhead is 1 bit per value. But bases are ascending, so one could DELTA-encode all bases of consecutive vectors in a row-group header. As each vector has T values per lane, and the sum of T W-bit values needs W+log(T) bits, a DELTA-encoded base can be stored in W+log(T)+1 bits, where the +1 is because these bases also need (uncompressed) bases. As 1024 main values need 1024/T bases, DELTA-encoding bases reduces base-overhead from 1 to (B+log(T)+1)/T bits per value. For example, for the T=64-bit data type, and DELTAs that fit W=7 bits, the extra cost is:((7+log(64)+1)/64)=0.21 bit per value. So that turns W=7 bits per value into 7.21 bits per value (3% overhead).

RLE has been shown to be useful in column-oriented databases [77]. Compared to other LWCs, RLE is fundamentally different: While other LWCs represent the original data as a sequence of small integers, RLE reduces the number of values required to represent the original data. This makes it very challenging to data-parallelize RLE, as we are dealing with a variable number of values. Nonetheless, there were several attempts to SIMDize RLE. The encoding part of RLE has been SIMDized in [58–60]. For the decoding part of RLE, Damme *et al.* propose a new implementation that could be considered the state-of-the-art [63]. We discussed this scheme when we introduced FastLanes-RLE and call it SIMDized RLE here.
Figure 2.17 shows that FastLanes-RLE is significantly faster than the other solutions, when

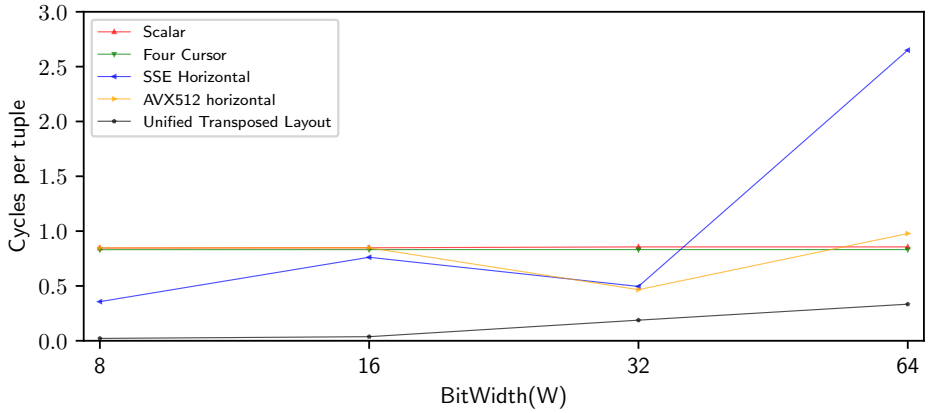


Figure 2.16: DELTA decoding on the Unified Transposed layout is 3x-40x faster than the alternatives (Ice Lake). Note the AVX512 horizontal computation falls back to scalar for T 8 and T 16 as it requires the `_mm512_alignr_epi` instruction.

runs are shorter than 333 (i.e. more than 3 runs in the 1024-value vectors we test on). This is because of two reasons. First, the SIMDized RLE and Scalar suffer from branch miss predictions. This happens in case of storing a new run, as there is a need to take another path to load the new value, and the branch happens more frequently as there are more runs. Second, the SIMDized RLE approach does not profit from the full width of a SIMD register. This is because the next `STORE` instruction may overwrite most of the values stored by the previous `STORE` instruction.

When introducing FastLanes-RLE, we already mentioned its compression ratio is better for runs with an average length $\leq$12 (in Figure 2.17, for more than 80 runs in a vector), but starts suffering for longer runs, as its Run Lengths require 1.375 bits per value (W=1 + (1+log(16)+1)/16 for bases, since FastLanes-RLE relies on W=1, T=16 FastLanes-DELTA). However, RLE compression ratio typically does not depend so much on Run Lengths as on Run Values, certainly if these are strings. Also, our future work on cascading encodings (i.e. compressing Run Values, and DELTA-bases) and exception handling schemes, will improve the compression ratio of FastLanes-RLE, by moving to 0-bit DELTA storage with the 1-bits as exceptions, for vectors with long runs.

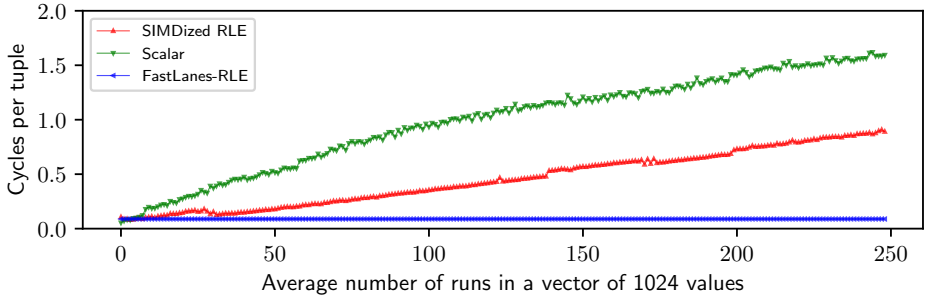


Figure 2.17: RLE decoding: Scalar, vs SIMDized vs FastLanes-RLE (Ice Lake). FastLanes-RLE is much faster except whith run lengths 333, i.e. at avg 3 runs in a 1024-value vector.

2.5 CONCLUSION AND FUTURE WORK

Current database systems only profit to a limited extent from what SIMD could bring [67, 71, 78]. With stalling progress in CPU frequency and core counts, this is still an opportunity for performance gains. In our vision, one needs to start by redesigning the basis – data storage – to seize this opportunity. This is why FastLanes proposes a new data layout, that creates opportunities for independent work on data-parallel hardware. Besides SIMD, we remark that other popular data-parallel hardware includes GPUs and TPUs and that we are in an age of further hardware innovation. The gist of FastLanes is that this age needs a data format that takes away sequential decoding dependencies and that is why its key idea is to reorder tuples in the special "04261357" 8x16 tiling order.

FastLanes can express all common LWC decoding methods in simple operations on a virtual (and future-proof) 1024-bits register that can efficiently map to existing SIMD instruction sets, as shown by our experiments on Intel, AMD, Apple and AWS hardware.

Rather than looking at value decoding in isolation, we look at it from a database systems context, where decompression is part of a pipeline that should be in balance with hardware resource limits, and where a column is not decoded fully in isolation, but incrementally (vector-at-a-time), as the source of a query pipeline, that processes the data further, and where the scan decodes multiple different columns. And, where decoding infrastructure is part of a (vectorized) software subsystem [79], where code portability in an ever more heterogeneous hardware environment is of paramount importance, to limit development effort and technical debt.

FastLanes also has a scalar code-path, and the data-paralellism on compact data-types that it exposes, even accelerates scalar decoding in comparison with naive bit-packed sequentially stored data. A key result is that modern compilers can completely auto-vectorize this scalar code-path, with no performance penalty compared to explicit SIMD intrinsics. This makes FastLanes very portable.

The performance benefits of FastLanes start by providing **much** faster decompression: our bit-unpacking followed by FOR and DELTA decompression improve over naive sequential bit-packed layouts by often an order of magnitude (or more). We showed that RAM-resident

queries can get even faster on FastLanes-compressed data, when compared with direct in-memory array scans.

Future Work. Our proposed kernels, such as FastLanes-RLE are not targeting full/eager decompression, but rather partial decompression into *compressed vector* representations. Such vector representations, that represent vectors of data in tight arrays that fit in a lane-width that is much smaller than the fully decompressed value, unlock opportunities for relational operators higher up in the pipeline to exploit compressed execution [49–51, 64, 77, 80].

Research could establish whether the data-parallelism that FastLanes creates makes it also suitable to efficiently scan and process data on widely-parallel hardware such as TPUs and GPUs [81].

In FastLanes we aim not only to improve the speed of LWC decoding, but also the compression ratio. We are researching the idea of *cascading* LWCs [82], where compression methods are stacked on top of each other, and combined with various exception handling schemes; with the ultimate goal of making general-purpose compression methods such as zstd, Snappy and (even) LZ4 less necessary in big data formats; as their decoding speeds are orders of magnitude slower than FastLanes, and holding back performance.

We leave an evaluation in a complete system on end-to-end benchmarks for future work. We intend to integrate FastLanes in a complete open source future-proof big data file format. Cascading compression implies that each logical column chunk gets stored in potentially multiple recursively compressed physical sub-column-chunks, and this involves making and evaluating many design decisions in row-group, data-chunk and meta-data organization.

3

ALP: Adaptive Lossless Floating-Point Compression

IEEE 754 doubles do not exactly represent most real values, introducing rounding errors in computations and [de]serialization to text. These rounding errors inhibit the use of existing lightweight compression schemes such as Delta and Frame Of Reference (FOR), but recently new schemes were proposed: Gorilla, Chimp128, PseudoDecimals (PDE), Elf and Patas. However, their compression ratios are not better than those of general-purpose compressors such as Zstd; while [de]compression is much slower than Delta and FOR.

We propose and evaluate ALP, that significantly improves these previous schemes in both speed and compression ratio (Figure 3.1). We created ALP after carefully studying the datasets used to evaluate the previous schemes. To obtain speed, ALP is designed to fit vectorized execution. *This turned out to be key for also improving the compression ratio, as we found in-vector commonalities to create compression opportunities. ALP is an adaptive scheme that uses a strongly enhanced version of PseudoDecimals [83] to losslessly encode doubles as integers if they originated as decimals, and otherwise uses vectorized compression of the doubles' front bits. Its high speeds stem from our implementation in scalar code that auto-vectorizes, using building blocks provided by our FastLanes library [84], and an efficient two-stage compression algorithm that first samples row-groups and then vectors.*

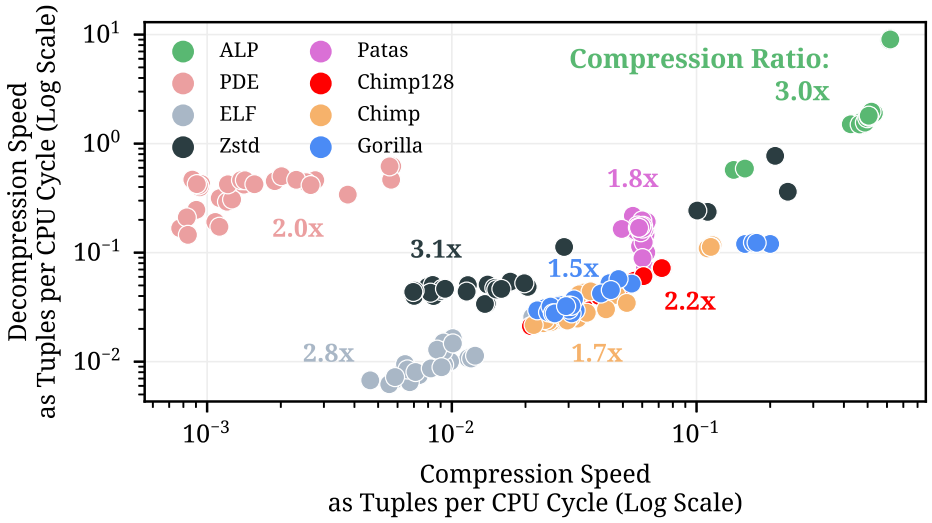


Figure 3.1: Compression performance for all schemes (on Intel Ice Lake). Each dot is one dataset. ALP is 1-2 orders of magnitude faster in [de]compression than all competing schemes, while providing an excellent compression ratio. The only one to achieve a compression ratio similar to ALP is Zstd, but it is slow and block-based (one cannot skip through compressed data). Elf is inferior to Zstd on all performance metrics. The evaluation framework is presented in Section 3.4.

3.1 Introduction

Data analytics pipelines manipulate floating-point numbers (64-bit *doubles*) more frequently than classical enterprise database workloads, which typically rely on fixed-point *decimals* (systems often store these as 64-bit integers). Floating-point data is also a natural fit in scientific and sensor data; and can have a temporal component, yielding *time series*. Analytical data systems and big data formats have adopted columnar compressed storage [1, 49, 50, 85–87], where the compression in storage is either provided by general-purpose or lightweight compression. Lightweight methods, also called "encodings", exploit knowledge of the type and domain of a column. Examples are Frame Of Reference (FOR), Delta-, Dictionary-, and Run Length Encoding (RLE) [88–90]. The first two are used on high-cardinality columns and encode values as the addition of a small integer with some fixed base value (FOR) or the previous value (Delta). These encodings also *bit-pack* the small integers into just the necessary bits. However, with IEEE 754 doubles [91], additions introduce rounding errors, making Delta and FOR unusable for raw floating-point data. General-purpose methods used in big data formats are gzip, Zstd, Snappy and LZ4 [92–94]. LZ4 and Snappy trade more compression ratio for speed, gzip the other way round, with Zstd in the middle. The drawback of general-purpose methods is that they tend to be slower than lightweight encodings in [de]compression; also, they force decompression of large blocks for reading anything, preventing a scan from *pushing down* filters that could *skip* compressed data.

Recently though, a flurry of new floating-point encodings were proposed: Gorilla [95],

Chimp and Chimp128 [96], PseudoDecimals (PDE) [83], Patas [97] and Elf [43]. A common idea in these is to use the XOR operator with a previous value in a stream of data; as combining two floating-point values at the bit-pattern level using XOR provides somewhat similar functionality to additions, without the problem of rounding errors. Chimp does an XOR with the immediate previous value, whereas Chimp128 XORs with one value that may be 128 places earlier in the stream – at the cost of storing a 7-bit offset to that value. After the XOR, most bits are 0, and the Chimp variants only store the bit sequence that is non-zero. Patas, introduced in DuckDB compression [97], is a version of Chimp128 that stores non-zero *byte*-sequences rather than bit-sequences. Whereas Patas trades compression ratio for faster decompression, Elf [43] does the opposite: it uses a mathematical formula to zero more XOR bits and improve the compression ratio, at the cost of lower [de]compression speed. PDE is very different as it does not rely on XOR: it observes that many values that get stored as floating-point were originally a decimal value and it endeavours to find that original decimal value, and compress that.

While these floating-point encodings avoid the need to always decompress largish blocks, as required by general-purpose compression, and thereby allow for predicate push-down in big data formats [98], their [de]compression speed (as well as compression ratio) is not much higher than that of general-purpose schemes [43]; in other words, these encodings are not quite lightweight.

We introduce ALP, a lightweight floating-point encoding that is *vectorized* [30]: it encodes and decodes arrays of 1024 values. It is implemented in dependency-free scalar code that C++ compilers can *auto-vectorize*, such that ALP benefits from the high SIMD performance of modern CPUs [99, 100]. In addition, ALP achieves much higher compression ratios than the other encodings, thanks to the fact that vectorized compression does not work value-at-a-time but can take advantage of commonalities among all values in one vector. Its vectorized design also allows ALP to be *adaptive* without introducing space overhead: information to base adaptive decisions on is stored once per vector rather than per value, and thus amortized. While per-value adaptivity (e.g., Chimp[128] has four decoding modes) needs control instructions (if-then-else) for every value, and can run into CPU branch mispredictions, ALP's per-*vector* adaptivity only needs control-instructions once per vector, but vector [de]compression itself has very few data- or control dependencies, leading to higher speeds.

Our main contributions are:

- a study of the datasets that were used to motivate and evaluate the previous floating-point encodings, leading to the new insights (e.g., many floating-point values actually were originally generated as a decimal).
- the design of ALP, an adaptive scheme that either encodes a vector of values as compressed decimals, or compresses only the front-part of the doubles, that holds the sign, exponent, and highest bits of the fraction part of the double.
- an efficient two-level sampling scheme (happening respectively per row-group, and per vector) to efficiently find the best method during compression.
- an open-source implementation of ALP in C++ that uses vectorized lightweight compression that can cascade (e.g, use Dictionary-compression, but then also

compress the dictionary and the code columns, with Delta, RLE, FOR – such as provided by [83, 84, 101]).

- an evaluation versus the other encodings on the datasets that were used when these were proposed, showing that ALP is faster *and* compresses better (as summarized in Figure 1).

3

3.2 Datasets Analysis

Compression methods achieve their best performance when they are capable of exploiting properties of the data. However, the same methods could fail to achieve any compression if the data lacks these exploitable properties. In this section we analyze a number of floating-point datasets, aiming to uncover properties relevant to compression performance. Furthermore, we are interested in analyzing these datasets from the point of view of *vectorized* query processing, since big data format readers and scan subsystems of database systems by now standardize on this methodology [87, 102]: they deliver vector-sized chunks of data, and use decompression kernels that decompress one vector (e.g., 1024 values) at-a-time.
We start by explaining in detail the IEEE 754 doubles representation in subsection 3.2.1. Then, we introduce the analyzed datasets in subsections 3.2.2 and 3.2.3. Next, in subsection 3.2.4 we analyze the data similarities at the vector level. In subsection 3.2.5 we revisit decimal-based encoding approaches and perform further analysis of these methods from a vectorized point of view. Finally, in subsection 3.2.6 we elaborate on the compression opportunities we found.

3.2.1 IEEE 754 Doubles Representation

IEEE 754 [91] represents 64-bit doubles in 3 segments of bits (Figure 3.2): 1 bit for sign (0 for positive, 1 for negative), 11 bits for an exponent e (represents an unsigned integer from 0 to 2047) and 52 bits for the fraction (represents a summation of inverse powers of two; also known as mantissa or significand) – which together represent a real number defined as: $(-1)^{sign} \times 2^{e-1023} \times (1 \sum_{i1}^{52} b_{52-i} 2^{-i})$. This definition allows for up to 17 significant decimal places of precision. However, it introduces errors in arithmetic (e.g. addition, multiplication) and limitations on the integer part of numbers which we will discuss later on in this section. The same standard also defines 32-bit floats (8 bits for exponent and 23 for mantissa).

Figure 3.2: IEEE 754 doubles bitwise representation.

3.2.2 Datasets

Table 3.1 presents an overview of the 30 datasets that we analyzed in detail in order to design ALP: 18 of these datasets were previously analyzed and evaluated to develop Elf [43] and Chimp [96], the other 12 were used to evaluate PDE [83]. We consider these 30 datasets to be relevant because they capture a variety of distributions, and because they played a role in the analysis, design and evaluation of competing floating-point encodings. Identifying new properties, we gained important clues guiding the design of ALP. Finally, by using these datasets we are able to perform a fair comparison between these methods and our new ALP compression.

3.2.3 Dataset Semantics

The first 13 datasets presented in Table 3.1 contain **time series** data. On these datasets, each double value v_{i1} is recorded further in time than value v_i. The next 17 datasets are more representative of doubles stored in classical database workloads; 12 of these non-time series datasets are part of the Public BI Benchmark [103] a collection of the biggest Tableau Public workbooks [104]. Note that all datasets are user-contributed data (non-synthetic).
The datasets have significant variety in their semantics. As presented in Table 3.1, 14 datasets contain doubles that represent monetary values (i.e., Exchange rates, public funds, product prices, stocks and crypto-currencies). 4 of them represent coordinates (i.e., latitude and longitude), 2 contain discrete counts stored as doubles and 1 contains computer storage capacities. Finally, the other 10 datasets contain a variety of scientific measures (i.e., temperature, pressure, concentration, speed, degrees and energy). Some datasets share a common prefix in their name followed by a number. This number represents the *index* of the analyzed *column* in a dataset.

3.2.4 Data Similarity

The underlying temporal property of time series data has been shown to result in *similar* values stored close-by [95, 96]. We can analyze similarity of doubles from two different points of view: (i) their bitwise representation (IEEE 754 [91]) and (ii) their human-readable representation.

Bitwise similarity. From a bitwise point of view, two double floating-point values are considered *similar* if their sign, exponent and fraction parts are similar. Table 3.2:C9 and C10 show the double exponent average and deviation per vector. We define a `vector` as 1024 consecutive values [30]. In most of the datasets, the exponent deviation is small, particularly in time series data. These small deviations are reflected by the number of leading 0-bits resulting from XORing the doubles with their previous value. When similar

[1] https://www.meteoblue.com/en/weather/archive/export/basel_switzerland
[2] https://github.com/influxdata/influxdb2-sample-data
[3] https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities
[4] https://zenodo.org/record/3886895
[5] https://github.com/cwida/public_bi_benchmark
[6] https://gz.blockchair.com/bitcoin/transactions/
[7] https://data.humdata.org/dataset/wfp-food-prices
[8] https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database
[9] https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks

Table 3.1: Floating-Point Datasets

	Name ↓	Semantics	Source	N° of Values
Time series	Air-Pressure[105]	Barometric Pressure (kPa)	NEON	137,721,453
	Basel-temp[1]	Temperature (C°)	meteoblue	123,480
	Basel-wind[1]	Wind Speed (Km/h)	meteoblue	123,480
	Bird-migration[2]	Coordinates (lat, lon)	InfluxDB	17,964
	Bitcoin-price[2]	Exchange Rate (BTC-USD)	InfluxDB	2,686
	City-Temp[3]	Temperature (F°)	Udayton	2,905,887
	Dew-Point-Temp[106]	Temperature (C°)	NEON	5,413,914
	IR-bio-temp[107]	Temperature (C°)	NEON	380,817,839
	PM10-dust[108]	Dust content in air (mg/m3)	NEON	221,568
	Stocks-DE[4]	Monetary (Stocks)	INFORE	43,565,658
	Stocks-UK[4]	Monetary (Stocks)	INFORE	59,305,326
	Stocks-USA[4]	Monetary (Stocks)	INFORE	282,076,179
	Wind-dir[109]	Angle Degree (0°-360°)	NEON	198,898,762
Non Time series	Arade/4[5]	Energy	PBI Bench.	9,888,775
	Blockchain-tr[6]	Monetary (BTC)	Blockchain	231,031
	CMS/1[5]	Monetary Avg. (USD)	PBI Bench.	18,575,752
	CMS/25[5]	Monetary Std. Dev. (USD)	PBI Bench.	18,575,752
	CMS/9[5]	Discrete Count	PBI Bench.	18,575,752
	Food-prices[7]	Monetary (USD)	WFP	2,050,638
	Gov/10[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/26[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/30[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/31[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/40[5]	Monetary (USD)	PBI Bench.	141,123,827
	Medicare/1[5]	Monetary Avg. (USD)	PBI Bench.	9,287,876
	Medicare/9[5]	Discrete Count	PBI Bench.	9,287,876
	NYC/29[5]	Coordinates (lon)	PBI Bench.	17,446,346
	POI-lat[8]	Coordinates (lat, in radians)	Kaggle	424,205
	POI-lon[8]	Coordinates (lon, in radians)	Kaggle	424,205
	SD-bench[9]	Storage Capacity (GB)	Kaggle	8,927

doubles are XORed, the result typically has a high number of leading- and trailing-zero bits [95, 110, 111]. However, in Table 3.2:C14 and C15, we see that the average number of leading and trailing zeros bits after XORing is comparable between time series and non-time series data. Hence, this *similarity* of values stored close-by is also present on non-time series data; which is also reflected by the fact that Chimp and Chimp128 do really well on this data [96]. Regardless of semantics, leading and trailing zero bits go down with lower percentages of duplicates (Table 3.2:C6 non-unique values) and higher decimal precision (Table 3.2:C2). For instance, in both datasets in which decimal precision reaches 20 digits (i.e., POI-lat and POI-lon), the leading and trailing 0-bit average of XORed values is the lowest.

Human-readable similarity. From a human perspective, two doubles are similar if their orders of magnitude (exponent) and their visible decimal precision are similar. On our time series datasets, the standard deviation of the magnitudes (Table 3.2:C8) is relatively small (e.g., Stocks-USA, Dew-Point-Temp, Air-Pressure). In contrast, on non-time series data, this measure is elevated for some datasets (e.g., Food-Prices, Gov/40, CMS/9), though never extremely high when compared to the average magnitude (Table 3.2:C7).
Decimal precision varies between datasets (Table 3.2:C2 and C3). For instance, datasets that contain geographic coordinates such as POI-lat and POI-lon can vary between 0 and 20 decimals of precision. On the other hand, datasets such as Medicare/9, SD-bench and City-Temp contain values with just 1 decimal of precision. Despite these differences inside a dataset, the deviation of this property is usually small from a vector perspective (Table 3.2:C5). In fact, for 25 out of 30 datasets, the decimal precision deviation inside vectors is smaller than 1. That means that most of the values inside a vector share the same decimal precision.
Decimal-based encoding approaches such as PDE exploit these human-readable similarities of doubles by trying to represent them as integers [83]. The more similar the decimal precision and the orders of magnitude of doubles inside a block of values, the better compression ratio can be achieved.

3.2.5 Representing Doubles as Integers

Representing double-precision floating-point values as integers is non-trivial. Take for instance the number $\boldsymbol{n}$ **8.0605**. At first glance, to encode $\boldsymbol{n}$ as an integer we could be tempted to move the decimal point $\boldsymbol{e}$ spaces to the right until there are no decimals left (i.e., 4 spaces). The latter can be achieved with the following procedure: P_{enc} $round(n \times 10^{e})$. Since one of the multiplication operands of P_{enc} is a double, we need to round the result to obtain an integer. Then, we could conclude that we have reduced our double-precision floating-point number into a 32-bit integer $\boldsymbol{d}$ 80605 (i.e., the result of P_{enc}) and another 32-bit integer representing the number of spaces $\boldsymbol{e}$ we moved the decimal point (i.e., a factor of 10). Hence, from the encoded integer $\boldsymbol{d}$ result of P_{enc}, and the number of spaces $\boldsymbol{e}$ we moved the decimal point, we should be able to recover the original double by performing the following procedure: P_{dec} $d \times 10^{-e}$.
Executing this in a programming language will *visually* yield on screen the original number 8.0605. However, the exact bitwise representation of the original double has been lost in the process. The correctness of the procedures fails to hold due to our number 8.0605 not being

a **real double** [112]. The real representation of the number 8.0605 as a double based on the IEEE 754 definition is: 8.0604999999999933209. To achieve lossless compression, this has to be the exact result of our procedure P_{dec}. However, in our example P_{dec} yields 8.0605000000000011084. This is a consequence of the error introduced in the multiplication by the inverse factor of 10 in P_{dec}. The latter turns out to be a double that does not have an exact decimal representation either. Hence, 10^{-4} is not 0.0001 but more something like 0.000100000000000000002082. This error is introduced in the multiplication, and reflected in the end result of the procedure P_{dec}. The P_{enc} procedure does not suffer this problem since 10^e has an exact double representation for $e \leq 21$.
Table 3.2:C11 depicts the percentage of doubles in each dataset that can be losslessly represented by an integer d and an exponent e using the P_{enc} and P_{dec} procedures. But, *always* using the *visible* precision of the doubles as the exponent e (e.g., for 0.**0001**, the visible precision is 4; for 1.**4297546**, the visible precision is 7). This results in only 82.5% of the values successfully encoded and decoded on average for all the datasets. However, in some datasets, the success probability gets as low as 61.7%. We found the success of the procedures P_{enc} and P_{dec} to encode and decode the exact original doubles to depend on two factors: (i) the *real precision* of the exponent e and (ii) the *visible precision* of the double n.

High exponents work for all values. Table 3.2:C12 shows the exponent e which leads to the highest success-rate of P_{enc} and P_{dec} on each dataset. It is evident that higher exponents e such as 14 and 16 are predominant, with an average of 95% successfully encoded values in all of the datasets; and up to a rate of 99.9% in datasets such as SD-bench, Stocks-UK, Medicare/9, Gov/31 and PM10-dust. The effectiveness of higher exponents stems from the fact that the more we *increase the exponent* e the closer we can get to obtaining the real double with the procedures. This is due to higher exponents e resulting in a more precise inverse factor of 10 on P_{dec}. For instance, 10^{-14} represented as a double is equal to $1.0000000000000007771E^{-15}$. As a consequence, the result of P_{dec} is more accurate. Furthermore, higher exponents are powerful because they are able to cover a wider range of decimal precision. Moreover, as shown in Table 3.2:C13, when optimizing to use a different exponent e per vector, we reach an average of 97.2% of successfully encoded values in all the datasets. Based on these results, *we question* whether a different exponent e for each value is needed – which is what PDE does.
However, by using higher exponents e the integers resulting from the procedure P_{enc} become big (i.e., 64-bits). These high exponents that lead to big integers are not used by PDE since they lead to a worse compression ratio than leaving the data uncompressed (because storing a 64-bit integer plus an exponent takes more space than a 64-bit double). Note that the doubles in datasets such as NYC/29, POI-lat and POI-lon are *only* representable as big integers.

The 52-bit limit for integers. Exponent e 14 is the most successful in most of the datasets to represent doubles as integers using P_{enc} and P_{dec}. This is due to the difference between the exact value and the real value of 10^{-14} being too small to have an effect in P_{dec} result. However, there are two datasets in which even higher exponents e are needed (i.e., POI-lat, POI-lon) because the visible precision of the double values inside those datasets on average exceeds 14 (Table 3.2:C4). As we explain subsequently, when the order of magnitude of a double n plus its visible decimal precision reaches 16, P_{enc} is prone to fail due to a limitation of the IEEE 754 doubles.

The multiplication inside P_{enc} yields a double due to having a double operand. Hence, before rounding, our resulting integer $\boldsymbol{d}$ is a double. However, there is a known limitation to the accuracy of the integer part of a double. Only the integers ranging from -2^{53} to 2^{53} can be exactly represented in the integer part of a double number. Going beyond this threshold is problematic. Between 2^{53} and 2^{54}, only *even* integer numbers can be represented as doubles. Similarly, between 2^{54} and 2^{55} only *multiples of 4* can exist. Furthermore, doubles stop having a decimal part after 2^{53}. Hence, if a double multiplication yields a double higher than 2^{53}, results will be automatically rounded to the nearest existing double number. The latter happens in P_{enc} when the order of magnitude of the double plus the visible decimal precision reaches 16. Hence, representing a number as an integer could be impossible in these cases using P_{enc} and P_{dec}. This is why POI-lat and POI-lon achieve a relatively low successful encoding rate of 76.4% and 70.5% respectively. Also, this is why we stated earlier that 10^e only has an exact double representation for $\boldsymbol{e} \leq 21$.

3.2.6 Unexploited Opportunities

All recently proposed competing floating-point encoding already exploit some of the properties discussed in the previous subsections. However, there is room for substantial improvement both in terms of compression ratio and [de]compression speed.

Vectorizing Decimal Encoding. In subsection 3.2.5 we demonstrated that it is possible to achieve near 100% success rate of our procedures P_{enc} and P_{dec} by using only one exponent $\boldsymbol{e}$ for every vector. The current state-of-the-art Decimal-based approach PDE [83] embeds the exponent $\boldsymbol{e}$ in every value. Hence, by exploiting this opportunity, compression ratio could be improved.

Cutting trailing 0s with an extra multiplication. In subsection 3.2.5 we demonstrated that high exponents $\boldsymbol{e}$ achieve the highest success rate on our procedures P_{enc} and P_{dec} to store doubles as integers. However, we also mentioned that using exponents such as 14 results in 64-bit integers being encoded. Despite this, we believe that using a unique exponent $\boldsymbol{e}$ per vector opens the opportunity to encode big integers without instantly falling *behind* in compression ratio against uncompressed values.
High exponents $\boldsymbol{e}$ in combination with low-precision decimals datasets (e.g., SD-bench, City-Temp, Stocks-UK) result in 64-bit integers that contain *tails* of repeated trailing 0-digits (e.g., $\boldsymbol{n} \approx 37.3$ and $\boldsymbol{e}$ 14, yields P_{enc} 3730000000000000; $\boldsymbol{n} \approx 100.8333$ and $\boldsymbol{e}$ 14, yields P_{enc} 10083330000000000). These tails of repeated 0-digits will have the same length in datasets with low magnitude variance and low decimal precision variance (e.g., SD-bench, City-Temp, PM10-Dust). Cutting these tails with an extra multiplication with an inverse factor of 10, namely $\boldsymbol{f}$, results in a smaller integer that can be used to recover the 64-bit integer with the inverse operation (i.e., a multiplication with a factor $\boldsymbol{f}$ of 10). Hence, we can redefine P_{enc} and P_{dec} as follows:

$$ALP_{enc} \quad round(n \times 10^e \times 10^{-f}) \tag{3.1}$$

$$ALP_{dec} \quad d \times 10^f \times 10^{-e} \tag{3.2}$$

Based on the analysis done in subsection 3.2.5 one might fear that this new multiplication with another inverse factor of 10 in ALP_{enc} could result in new rounding errors. However, the

error introduced by these inverse factors of 10 turns out to pose no problems. To illustrate, with $\boldsymbol{n} \approx 8.0605$, $\boldsymbol{e}$ 14 and $\boldsymbol{f}$ 10, ALP_{enc} and ALP_{dec} will execute as follows:

$$ALP_{enc} \;\; round(\; 8.06049999999999933209 \times 10^{14} \times 10^{-10})$$
$$ALP_{enc} \;\; round(806049999999999.875 \times 10^{-10})$$
$$ALP_{enc} \;\; round(80604.999999999985448) \;\; 80605 \rightarrow d$$
$$ALP_{dec} \;\; 80605 \times 10^{10} \times 10^{-14}$$
$$ALP_{dec} \;\; 8.06049999999999933209 \;\; n$$

In the third step of ALP_{enc}, the error introduced by 10^{-10} is negligible for the resulting integer $\boldsymbol{d}$. Using this reducing factor $\boldsymbol{f}$ in the procedures is a way of taking advantage of the high coverage and success rates of large exponents, without having to encode big integers $\boldsymbol{d}$. Note that this example is the same $\boldsymbol{n}$ we used at the beginning of subsection 3.2.5, which could not be encoded by simply using $\boldsymbol{e}$ 4. Also, note how a tail composed of 9-digits can also be reduced without any side-effect.

Limited Search Space. Until now, we have ignored the process of *finding* the exponent $\boldsymbol{e}$ for our decimal-based encoding procedures ALP_{enc} and ALP_{dec}. The current state-of-art on decimal-based encoding (i.e., PDE) performs a brute-force search for each value in a dataset in order to find the exponent $\boldsymbol{e}$. For our *ALP* procedures, an additional nested brute-force search needs to be performed in order to find the **best combination** of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$. We define the *best combination* as the one in which ALP_{enc} yields the smallest integer $\boldsymbol{d}$ with which ALP_{dec} succeeds in recovering the original double $\boldsymbol{n}$. This translates into a search space of 253 possible exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ combinations (given that $\boldsymbol{f} \leq \boldsymbol{e}$ and $0 \geq \boldsymbol{e} \leq 21$). However, we have already discussed that most values inside a vector can be encoded by using one single exponent. Furthermore, we have also mentioned that vectors exhibit a low variance in their decimal precision and in their magnitudes. Hence, our intuition was that the search space for the combination of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ can be greatly reduced and that it should be done on a per-vector basis. In order to confirm this, we computed the best combination for each vector in each dataset. For this experiment, the search was performed on all the possible search space of 253 combinations for every vector. Figure 3.3 shows that for most datasets a search space of 5 combinations is enough to obtain the best combination among all vectors in the dataset. For some datasets such as Basel-wind, Bird-migration, City-Temp, Wind-dir and IR-bio-temp, the entire search space is just one combination.

Front-Bits Similarity. When the magnitude plus decimal precision exceeds 16, it is often impossible to encode a double as an integer with our procedure ALP_{enc}. On such data, decimal-based encoding would have to deal with integers bit-packed to more than 52 bits (and similarly, Chimp variants would have to deal with trailing bit-strings of more than 52 bits). A basic observation is that such data is not very compressible in the first place (64-bit data takes at least 52 bits); but nevertheless, compression may still be worthwhile.
We believe that the approach of a decimal-based encoding is not appropriate for such compression-unfriendly data; and thus when encountering such data, our approach could

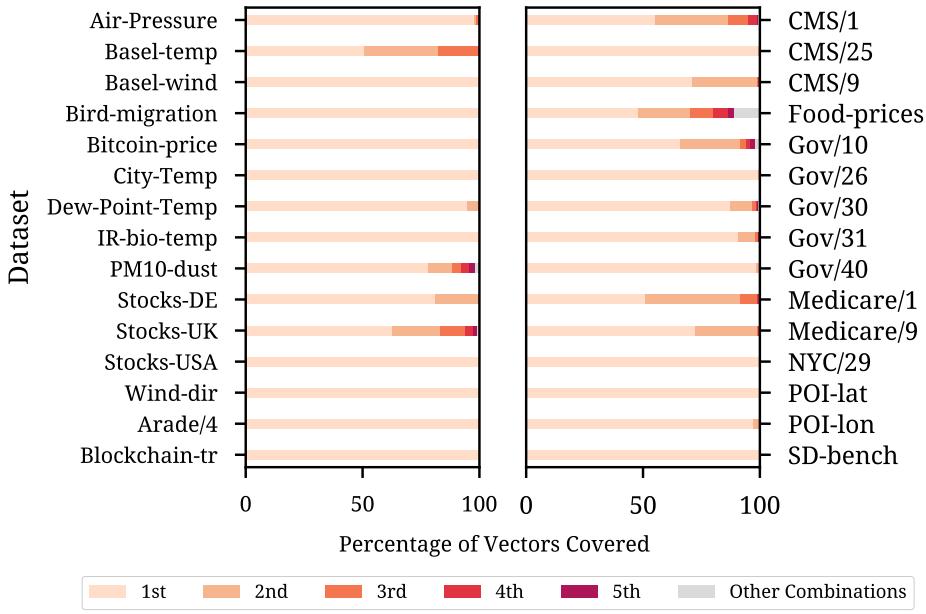


Figure 3.3: Analysis of the best combinations of exponent e and factor f for each vector of 1024 values. For most datasets, the best combination for any vector is found among a set of just 5 different combinations. For some datasets, a single combination is always the best one.

adaptively switch to a different encoding strategy, that exploits regularities in the front-bits in a vectorized manner. In Table 3.2:C10, even on these datasets (i.e., POI-lat, POI-lon) we see that the exponent of the bitwise representation of a double exhibits a low variance. Data with low variance can be compressed with lightweight integer encodings, such as RLE and Dictionary – all building blocks provided by our FastLanes compression library [84]. Furthermore, based on the analysis of leading 0-bits from XOR-ing with the previous value (Table 3.2:C14), on some of these datasets we should not limit this idea to just the exponent, because the highest bits of the mantissa often are regular (if the data stems from a particular value range).

Table 3.2: Detailed metrics computed on the Datasets

Name ↓	**Decimal Precision Max | Min | Avg. | Std. Dev.**				**Values per Vector Non-Unique % | Avg. | Std. Dev.**			**IEEE 754 Exponent per Vector Avg. | Std. Dev.**		**Success of P_{enc} and P_{enc} using one exponent e per: Value | Dataset | Vector**			**Previous Value XOR 0's Bits Front | Trail.**	
C1	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*	*C10*	*C11*	*C12*	*C13*	*C14*	*C15*
Air-Pressure	5	0	4.9	0.3	74.7%	93.4	0.1	1021.5	0.0	63.2%	14 (99.4%)	99.4%	44.5	32.9
Basel-temp	11	5	6.3	0.4	26.2%	11.4	4.6	1025.5	1.0	64.3%	14 (99.7%)	99.7%	14.0	2.6
Basel-wind	8	0	6.1	1.2	61.8%	7.1	4.1	1024.7	12.8	65.8%	14 (98.6%)	98.6%	14.2	3.1
Bird-migration	5	1	4.5	0.8	55.9%	26.6	6.0	1026.4	0.6	61.7%	14 (93.8%)	96.4%	26.4	7.8
Bitcoin-price	4	1	3.9	0.4	0.0%	19187.5	790.6	1037.0	0.0	84.2%	14 (99.9%)	99.9%	20.6	1.0
City-Temp	1	0	0.9	0.3	60.3%	56.0	21.3	1028.3	1.6	67.3%	14 (97.4%)	97.4%	15.8	11.0
Dew-Point-Temp	3	0	2.8	0.3	19.3%	14.4	1.4	1026.0	1.1	80.2%	14 (99.3%)	99.3%	16.8	1.5
IR-bio-temp	2	0	1.9	0.3	49.1%	12.7	4.2	1025.6	4.8	83.5%	14 (99.3%)	99.3%	22.0	7.8
PM10-dust	3	0	2.8	0.2	93.7%	1.5	0.8	1016.1	1.2	88.8%	14 (99.9%)	99.9%	40.5	38.3
Stocks-DE	3	0	2.4	0.5	89.2%	63.8	9.1	1027.8	0.3	84.2%	14 (98.9%)	99.1%	24.9	5.8
Stocks-UK	2	0	1.2	0.6	88.1%	1593.7	317.1	1032.2	0.4	84.5%	14 (99.9%)	100.0%	23.7	19.4
Stocks-USA	2	0	1.9	0.4	91.5%	146.1	11.7	1029.1	0.1	87.5%	14 (98.6%)	99.2%	32.6	16.8
Wind-dir	2	0	1.9	0.3	3.9%	192.4	81.1	1029.8	1.2	90.0%	14 (99.5%)	99.5%	13.8	2.6
TS AVG.	**3.9**	**0.5**	**3.2**	**0.5**	**54.9%**	**1646.7**	**96.3**	**1026.9**	**1.9**	**77.3%**	**94,8%**	**99.0%**	**23.8**	**11.6**
Arade/4	4	0	3.5	0.6	0.2%	738.4	389.8	1031.6	0.9	80.1%	14 (99.5%)	99.5%	13.1	1.1
Blockchain-tr	4	0	3.8	0.6	0.6%	638646.4	$1.3E^7$	1031.8	12.5	76.3%	14 (92.1%)	92.3%	9.8	1.7
CMS/1	10	0	4.0	2.8	54.7%	97.0	110.0	1028.0	1.3	83.2%	14 (98.5%)	98.6%	32.9	24.8
CMS/25	10	0	9.1	1.9	5.7%	12.6	19.2	984.1	179.1	68.0%	14 (98.7%)	98.7%	9.5	1.5
CMS/9	1	0	0.0	0.0	71.5%	235.7	908.5	1028.3	1.6	100.0%	14 (99.9%)	100.0%	11.8	47.3
Food-prices	4	0	1.1	1.1	52.5%	6415.8	14656.8	1030.4	1.8	92.4%	14 (99.2%)	99.2%	27.1	33.5
Gov/10	2	0	1.0	0.8	26.1%	240153.6	$1.6E^7$	873.5	298.8	90.5%	14 (89.9%)	95.9%	13.8	18.8
Gov/26	2	0	0.0	0.0	99.5%	442.3	8036.8	4.6	11.9	100.0%	14 (99.9%)	100.0%	63.7	63.8
Gov/30	2	0	0.1	0.3	89.7%	10998.7	102748.6	115.6	170.6	98.6%	14 (98.5%)	99.4%	56.6	57.1
Gov/31	2	0	0.1	0.1	96.0%	893.2	6288.2	69.9	57.4	99.1%	14 (99.8%)	99.9%	60.6	60.9
Gov/40	2	0	0.0	0.0	99.1%	791.4	6650.9	12.1	18.7	99.9%	14 (99.8%)	99.9%	63.4	63.5
Medicare/1	10	0	4.0	2.9	41.3%	97.0	146.2	1028.0	1.6	83.2%	14 (98.5%)	98.6%	25.2	16.6
Medicare/9	1	0	0.0	0.0	70.6%	235.7	1006.2	1028.3	1.7	100.0%	14 (99.9%)	100.0%	11.3	47.1
NYC/29	13	0	12.9	0.3	51.0%	-73.9	0.0	1029.0	0.0	93.7%	14 (99.9%)	100.0%	38.9	23.2
POI-lat	20	0	15.9	0.4	1.4%	0.6	0.4	1021.7	1.4	73.4%	16 (74.1%)	76.4%	10.6	1.0
POI-lon	20	0	15.7	0.5	0.8%	-0.1	1.2	1022.0	4.0	64.6%	16 (61.5%)	70.5%	5.1	1.0
SD-bench	1	0	0.9	0.2	92.4%	446.0	521.5	1030.3	1.2	65.8%	14 (99.9%)	100.0%	17.4	15.8
NON-TS AVG.	**6.4**	**0.0**	**4.2**	**0.7**	**50.2%**	**52948.9**	**1745162.6**	**786.4**	**45.0**	**86.4%**	**95.1%**	**95.8%**	**27.7**	**28.2**
ALL AVG.	**5.3**	**0.2**	**3.8**	**0.6**	**52.2%**	**30717.9**	**988967.2**	**890.6**	**26.3**	**82.5%**	**95.0%**	**97.2%**	**26.0**	**21.0**

3.3 ALP

ALP is an adaptive lossless encoding designed to compress double-precision floating-point data. ALP takes advantage of the opportunities discussed in subsection 3.2.6. Compression and decompression are built upon the ALP_{enc} and ALP_{dec} procedures described in section 3.2.6. Furthermore, ALP is able to *adapt* its encoding/decoding scheme if it encounters high precision doubles by taking advantage of the similarity in the front-bits uncovered in section 3.2.6. in both compression and decompression. In the following subsections, we describe the key design aspects of ALP and how it implements adaptivity.

3.3.1 Compression

ALP compression is built upon the ALP_{enc} procedure (Formula 3.1). ALP tries to encode all doubles $\boldsymbol{n}$ inside a vector $\boldsymbol{v}$ with the same exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$. Inside the encoding, ALP must verify that the procedures ALP_{enc} and ALP_{dec} yield the original double n. If the original double $\boldsymbol{n}$ cannot be recovered, we treat the double as an *exception*. Algorithm 3.1 shows the pseudo-code for ALP encoding.

Vectorized Compression. ALP introduces the use of one exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ for all doubles inside the same vector. Note that PDE needs to store one exponent *per value* – taking more space. Based on our empirical investigation, in order for this approach to be successful we need to be able to use high exponents $\boldsymbol{e}$. Hence, ALP does not limit the encoded integers to `int32` representations, but `int64`. Furthermore, ALP incorporates the new idea of the factor $\boldsymbol{f}$ for reducing the trailing 0-digits, explored in subsection 3.2.6. After multiplying with the factor, the resulting integer is small again and is then *bit-packed* compactly, using the same number of bits for all values inside the same vector. The exponent, factor and bit-width parameters do not use much space, as these parameters are stored only once per vector (1024 doubles). The fact that all three parameters are the same per-vector also means that the [de]compression work is regular and thus has no control-instructions inside the loops, making them suitable for auto-vectorization.
Fast Rounding. The `round` operation is not supported in SIMD instruction sets. However, ALP replaces the `round` function with a procedure (i.e. **`fast_double_round`**) that takes advantage of the limitation of doubles to store exact integers of up to 52 bits, discussed earlier. An algorithmic trick resulting from this limitation is that one can round a double by adding and subtracting the following number: $sweet_n \ 2^{51} \ 2^{52}$. In other words, we take the doubles to the range in which they are not allowed to have a decimal part (between 2^{52} and 2^{53}) and are "automatically" rounded. For instance, to round a double n, **`fast_double_round`** will go as follows: $n_{rounded} \ cast \ int64 \ (n \ sweet_n - sweet_n)$. This procedure is SIMD-friendly since it only consists of one addition and one subtraction; operations supported by SIMD. This rounding trick is also implemented in the Lua programming language. The use of **`fast_double_round`** can be seen in Algorithm 3.1: Line 10.

Handling Exceptions. Values which fail to be encoded as decimals become *exceptions*. Exceptions are stored uncompressed in a separate segment (i.e., **`exc_vec`** in Algorithm 3.1). However, since our approach is vectorized, we cannot simply skip the exceptions in the resulting vector of encoded values (i.e., **`encoded_vec`** in Algorithm 3.1). Hence, when exceptions occur we store an *auxiliary* value in the **`encoded_vec`** (i.e., `first_encoded` in

Algorithm 3.1: ALP Compression.

```
1  double i_F10 = {1.0, 0.1, 0.01, 0.001, ...};
2  double F10 = {1.0, 10.0, 100.0, 1000.0, ...};
3
4  // Adaptive search of exponent e and factor f in a vector
5  int e, f = ALP::ADAPTIVE_SAMPLING(input_vec, BEST_COMBINATIONS);
6
7  encoded_vec, exc_vec, exc_pos_vec = ALP::ENCODE([]() {
8    for (i = 0; i < VECTOR_SIZE; ++i){ // Encode the vector
9      double n = input_vec[i];
10     int64 d = fast_double_round(n * F10[e] * i_F10[f]); // ALP_enc
11     encoded_vec[i] = d;
12     decoded_vec[i] = d * F10[f] * i_F10[e]; // ALP_dec
13   }
14   int exc_count = 0;
15   for (i = 0; i < VECTOR_SIZE; ++i) { // Find Exceptions
16     bool neq = (decoded_vec[i] != input_vec[i]);
17     exc_pos_vec[exc_count] = i;
18     exc_count += neq; // predicated comparison
19   }
20   int64 first_encoded = FIND_FIRST_ENCODED(exc_pos_vec);
21   for (i = 0; i < exc_count; ++i){ // Fetch Exceptions
22     encoded_vec[exc_pos_vec[i]] = first_encoded;
23     exc_vec[exc_pos_vec[i]] = input_vec[i];
24   }
25 });
26 FFOR(encoded_vec);
```

Algorithm 3.1 Line 20). This auxiliary value is the first successfully encoded $\boldsymbol{d}$ which is obtained by the `FIND_FIRST_ENCODED` function in Algorithm 3.1: Line 20. Such value will not affect negatively the bit-width of the encoded vector. Note that by searching for this value after the encoding process we avoid an additional control statement in each iteration of the main encoding loop. Further, we also need to store in another storage segment the position in which each exception occurred within a vector (i.e., `exc_pos_vec` in Algorithm 3.1). For $\boldsymbol{v}$ 1024, each exception has an overhead of 80 bits: 64 bits for the uncompressed value and 16 bits to store the exception position. Lines 15 to 25 in Algorithm 3.1 show the exception handling process which is cleverly built to avoid control structures (i.e., `if-then-else`).

Fused Frame-Of-Reference (FFOR). By itself, ALP encoding does not compress the data. Rather, it enables the use of lightweight integer compression to further encode its output. Based on our study of data similarity in subsection 3.2.4, we decided to encode the yielded integers using a Fused variant of the Frame-Of-Reference encoding available in the FastLanes library called `FFOR`. FastLanes [84] proposes a new data layout to accelerate the encoding and decoding of lightweight [de]compression methods with scalar code that auto-vectorizes. `FFOR` fuses the implementation of bit-[un]packing with the `FOR` encoding and decoding process into a single kernel that performs both processes. The `FOR` encoding subtracts the minimum value of the integers in a vector; this will pick up on localized doubles (inside a tight range) and reduce bits needed in the subsequent bit-packing. Fusing saves a SIMD store and load instruction in between the subtraction and the bit-packing loop (improving the performance). We note that it would also be possible to also fuse `FFOR` and ALP; this is not done yet here, and could provide a performance boost, especially in decoding.
However, there is some more headroom as a modern compression library (e.g., [83, 84])

could try multiple different integers encodings and also *cascade* these. For instance, if the data is repetitive, one could use Dictionary coding, and compress the Dictionary with FFOR; or use RLE and then separately encode Run Lengths and Run Values. If the data is (somewhat) ordered, one could apply Delta encoding rather than FFOR to the Dictionary or the Run Values.

3.3.2 Adaptive Sampling

Our compression method does not perform a brute-force search for the exponent e and factor f to use in a vector. Instead, to find the best e and f for a vector, we designed a novel two-level sampling mechanism, inspired by the findings in subsection 3.2.6. Specifically, from Figure 3.3, we conclude that there is a *limited* set of best combinations of exponent e and factor f for the vectors in a dataset.

Our sampling mechanism goes as follows: on the first sampling level, ALP samples m equidistant values from n equidistant vectors of a row-group. We define a row-group as a set of w consecutive vectors of size v. The total number of values obtained from this first sampling is equal to $m \times n$. For each vector n_i we find the *best* combination of exponent e and factor f. This search is performed on the entire search space (i.e., 253 possible combinations). The *best* combination is the one which minimizes the sum of the exceptions size and the size of the bit-packed integers resulting from the encoded m values. This process yields n combinations (one for each vector). From these n combinations we only keep the k ones which appeared the most. If two combinations appeared the same amount of times, we prioritize combinations with higher exponents and higher factors. It could be possible that fewer combinations than k are yielded. If the same best combination is found in every vector, there would only be 1 combination. Hence, we define during runtime a k' which is smaller than or equal to k that represents the number of yielded combinations. Once we have found the k' best combinations, we proceed to the second level of sampling. The second level of sampling (Line 5 of Algorithm 3.1) samples s equidistant values from a vector. Then, it tries to find the combination of exponent e and factor f which performs the *best* on the s sampled values. However, this time, the search is performed only among the k' best combinations found from the first sampling level. To further optimize the search, we implemented a greedy strategy of early exit. If the performance of two consecutive combinations, namely k'_{i1} and k'_{i2}, is worse or equal to the performance of the combination k'_i, we stop the search and k'_i combination is selected to encode the entire vector. If k' is equal to 1, this second sampling level is omitted for all the vectors inside the row-group.

The first level of sampling is the most computationally demanding process of our compression scheme due to the large search space. However, it occurs only once per row-group. Hence, the time spent is amortized into $w \times v$ encoded values. The second sampling level happens once for each vector and it will only occur if k'_i 1. Hence, if the sampling parameters (i.e., m, n, w, k and s) are tuned optimally, the second sampling level will be skipped in datasets such as City-Temp or SD-bench, in which there exists only one best combination for all the vectors in the dataset (Figure 3.3).

3.3.3 Decompression

ALP decompression builds upon the ALP_{dec} procedure (Formula 3.2) to recover the original doubles from a vector of integers d yielded by the encoding process. In order to do so, ALP

Algorithm 3.2: ALP Decompression.

```
int e, f = ALP::READ_VECTOR_HEADER(input_vec);
int64_vec = UNFFOR(input_vec);
decoded_vec = ALP::DECODE([](int64_vec) {
  for (i = 0; i < VECTOR_SIZE; ++i){
    decoded_vec[i] = int64_vec[i] * F10[f] * i_F10[e] }}); //ALP_dec
ALP::PATCH(decoded_vec, exc_vec, exc_pos_vec);
```

Algorithm 3.3: ALP_{rd} Compression and Decompression.

```
// ENCODING //
p, DICT = ALP::RD::ADAPTIVE_SAMPLING(input_rowgroup);
left_vec, right_vec = ALP::RD::ENCODE([]() {
  for (i = 0; i < VECTOR_SIZE; ++i){
    double n = input_vec[i];
    left_vec[i], right_vec[i] = ALP::RD::CUT(p);}
});
BITPACK(right_vec);
SKEWDICT_BITPACK(left_vec, DICT);

// DECODING //
p, DICT = ALP::RD::READ_ROWGROUP_HEADER();
left_vec = BITUNPACK_DECODEDICT(encoded_left_vec, DICT);
right_vec = BITUNPACK(encoded_right_vec);
decoded_vec = ALP::RD::DECODE([]() {
  for (i = 0; i < VECTOR_SIZE; ++i){
    int16 left, int64 right = left_vec[i], right_vec[i];
    decoded_vec[i] = ALP::RD::GLUE(left, right, p);}
});
```

first reads from the vector header the unique exponent e and factor f used to encode the vector. Then, ALP needs to reverse the FFOR integer encoding to recover each value. Values encoded as exceptions are directly read from the exception segment alongside their position on the original vector in order to correctly reconstruct it (i.e., patching). The pseudo-code for ALP decoding is presented in Algorithm 3.2.

3.3.4 ALP_{rd}: Compression for Real Doubles

During the first level of sampling ALP will detect whether the doubles in a row-group are not compressible. In that case, ALP encoding would result in a high number of exceptions and integers bigger than 2^{48}. Therefore, for such data, ALP changes its strategy to a different encoding approach based on the analysis performed in subsection 3.2.6 which hinted to us that even on these doubles, their front-bits tend to exhibit low variance. We named this approach ALP_{rd}, which stands for *ALP for Real Doubles*. ALP takes this decision at the row-group level rather than the vector level, since we found no dataset in which the decimal precision deviates on more than 3 decimals; hence taking this decision at a vector level would neither be efficient nor effective. We believe that the data in 28 of the 30 datasets analyzed originate as decimals and are thus not "real" doubles; however, we think that this is representative of the majority of data people store in data systems as doubles. The encoding and decoding of ALP_{rd} are presented in Algorithm 3.3.

Encoding. The first level of sampling finds at a row-group level which is the smallest position $p \geq 48$ where the highest 64-p front-bits still have low variance. Afterwards, it uses this number p as the position to *cut* the bits of every double of that row-group in two parts

(Line 6 of Algorithm 3.3). The right part is compressed using p-bits bit-packing (BP). The position p is stored once per row-group (i.e., 8 bits of overhead per row-group, which can be safely ignored). At first glance, this method does not achieve any compression, however, the integers yielded from the left part are easily further compressible with integer lightweight encoding methods. For the version of ALP presented here, we compress them using a fixed method: skewed DICTIONARY+BP compression. A skewed dictionary is a DICTIONARY encoding which tolerates exceptions. Here, exceptions are values not in the dictionary, and these are stored as 16-bits values in an exception array, together with an array containing 16-bits exception positions. After sampling, we consider dictionaries of sizes 2^b with $b \leq 3$ (i.e., just 1, 2, 4, or 8 values), and fill these with the most frequent values in the sample and then choose the smallest dictionary size b 3 such that the exception percentage does not exceed 10% (or else use b=3). We bit-pack the dictionary codes in b bits; and store the dictionary as 16-bits values. Both BP and DICTIONARY encodings implementations are available in our FastLanes library[113].

Decoding. The b bits dictionary-codes are bit-unpacked using a fast vectorized bit-unpacking primitive (that does this for the entire vector of 1024 values in one go) and (64-p) bits right parts of the doubles as well. Dictionary decompression requires one memory load from the dictionary for every code; which is relatively expensive. In SIMD it can be implemented with a gather instruction, but this is not supported on all CPU architectures nor does this instruction tend to be fast; hence we do not use such an approach (explicitly). Because we use small dictionaries of size $\leq 2^3$ 8 and the front-bits are maximally 16-bits wide; we note that we could implement decoding by preloading the dictionary (maximally 8x16-bits values) in a 128-bits SIMD register and then use a shuffle instruction. However, the results presented in this chapter are based on purely scalar dictionary decompression code, leaving space for improvement. Finally, we *glue* both parts together by left-shifting p bits the dictionary-decoded front-bits, after applying exception patching [99, 114] and adding in the decompressed right part (using vectorized SHIFT and OR, fused together in a **GLUE** primitive seen in Line 18 of Algorithm 3.3). Notice again that all operations are performed in a tight loop over arrays (vectorized query processing [86]) and the work is regular in nature such that C++ compilers get to very efficient code. Only the exception patching has some data dependencies and random memory access, but it is performed on a minority of the data only – limiting its performance effects.

3.4 Evaluation

We experimentally evaluate ALP with respect to its compression ratio and [de]compression speed using all analyzed datasets in Table 3.1 against six competing approaches for lossless floating-point compression: Gorilla [95], Chimp / Chimp128 [96], Patas [97], Elf [43] and PDE [83]. Furthermore, we also compare against one general-purpose compression approach: Zstd [92]. To further test the robustness of ALP we tested its speed on different hardware architectures which are described in Table 3.3 and using Auto-vectorized, Scalar and SIMDized code. In subsection 3.4.3 we present end-to-end query speed benchmarks of ALP on Tectorwise [61] to test its performance in a real system. Finally, in subsection 3.4.4

Table 3.3: Hardware Platforms Used

Architecture	Scalar ISA	Best SIMD ISA	CPU Model	Frequency
Intel Ice Lake	x86_64	AVX512	8375C	3.5 GHz
AMD Zen3	x86_64	AVX2 (256-bits)	EPYC 7R13	3.6 GHz
Apple M1	ARM64	NEON (128-bits)	Apple M1	3.2 GHz
AWS Graviton2	ARM64	NEON (128-bits)	Neoverse-N1	2.5 GHz
AWS Graviton3	ARM64	NEON (128-bits) SVE (variable)	modified Neoverse-V1	2.6 GHz

we present a version of ALP for 32-bits floats and evaluate it on machine learning data.

Sampling Parameters. Based on Figure 3.3, we defined the maximum number of combinations k as 5. The number of vectors w inside a `row-group` is fixed to 100 to emulate the usual modern OLAP engines row-group sizes (e.g., DuckDB [87]). The size of every vector v is fixed to 1024 to comfortably fit in the CPU cache [30]. On the first sampling level, the number of vectors sampled per `row-group` m is set to 8, and the number of values sampled per vector n is set to 32. Finally, on the second sampling level, the number of values sampled per vector s, is set to 32. m, n and s were tuned during evaluation and showed to yield a good trade-off between compression ratio and speed.

Algorithms Implementations. ALP is implemented in C++ and is available in our GitHub repository[10]. ALP uses the FastLanes library [113] to perform the lightweight encoding and decoding on its output (i.e., `FFOR`, `DICTIONARY`, `BP`). Gorilla, Chimp, Chimp128 and Patas were implemented in C++. Gorilla was implemented by ourselves, and the other implementations were stripped from the DuckDB codebase [115] and adjusted to work as standalone algorithms. Note that Gorilla is part of a closed-source Facebook system. On the other hand, PDE and Elf[11] benchmarks were carried out using code from the original authors. Finally, we used Facebook's implementation of Zstd in C [92], configured at the default compression level (3).

3.4.1 Compression Ratio

Table 3.4 shows the compression ratios of all approaches measured in bits per value (uncompressed, each value is a 64-bit double). In this experiment the algorithms compressed *all* vectors in a dataset. The best-performing floating-point approach is marked in green. ALP evidently stands out from the other floating-point encoding schemes in compression ratio. ALP shows an average improvement of ≈31% compared to PDE. When compared to Gorilla, Patas, Chimp, and Chimp128, ALP is respectively ≈49%, ≈39%, ≈43% and ≈24% better. In time series datasets ALP achieves a ≈33% and ≈46% improvement over Chimp128 and PseudoDecimals. Similarly, on non-time series data, ALP performs better than both by a ≈19% and ≈21% on average. Elf is ALP's most fierce competitor in terms of compression ratio – excluding Zstd. On the other hand, Zstd is the only compression algorithm that slightly takes the upper hand in compression ratio with 20.6 bits per value on

[10] `https://github.com/cwida/ALP`
[11] `https://github.com/Spatio-Temporal-Lab/elf`

average. Even so, ALP is slightly better than Zstd on time series data. One has to take into account that Zstd has a much lower [de]compression speed and, being block-based, has the disadvantage that one cannot optimally skip through compressed data. For instance, in Zstd's 256KB block-based compression, a system has to decompress 32 8KB vectors, even if 31 of those 32 vectors are not needed.

When ALP shines. ALP outperforms Chimp128 and Elf on datasets with fixed or low decimal precision or with a low percentage of repeated values (e.g., Blockchain-tr, Arade-4, Dew-Point-Temp, Bitcoin-price). In other words, ALP gets its best gains when the doubles were generated from *decimals*. ALP performs better than Chimp128 in 27 out of 30 datasets, and better than PDE in the same amount. In fact, ALP is at most 2 bits worse than PseudoDecimals on CMS/9 and Medicare/9. Both these datasets contain mostly integers encoded as doubles (Table 3.1). PDE benefits from such data since 0 bits are stored after applying BP to the exponents output due to the exponents always being equal to 0. Nevertheless, on these types of datasets Decimal-based encoding approaches are much better than XORing approaches. When ALP encounters *real doubles*, ALP_{rd} comes into the equation. There are two datasets for which ALP failed to achieve any compression and ALP_{rd} encoding was used: POI-lat and POI-lon (marked with *). These datasets are characterized by almost 0% of repeated values and a maximum decimal precision of 20 (Table 3.2:C2). In both datasets, these compression ratios achieved by ALP_{rd} represent an improvement over all the other floating-point compression approaches.

When ALP struggles. ALP struggles to keep up with both Elf and Chimp128 on datasets in which the XORing process benefits from a high percentage of repeated values and the decimal-based encoding process is hindered by a high variability in value precision. Those datasets are: CMS/1, Medicare/1 and NYC/29. Despite ALP encoding also taking advantage of similar data, the profit of Chimp128 / Elf when it can find an exactly equal value is much higher than the profit that ALP can get. Nevertheless, on data with many duplicates, we question whether floating-point encodings were the best decision in the first place. For instance, due to the high percentage of repeated values we could plug-in a `DICTIONARY` encoding before applying a floating-point encoding (or RLE, if the repeats are consecutive). We in fact tried using `DICTIONARY` and then compressing the dictionary with ALP, allowing it to achieve 33.1, 35.7 and 24.7 bits per value for CMS/1, Medicare/1 and NYC/29 respectively. The compression ratios that ALP is able to achieve by cascading compression using another lightweight encoding (i.e., `DICTIONARY` or `RLE`) are shown in the penultimate column of Table 3.4. By doing so, ALP even beats Zstd in compression ratios while still retaining its advantages (higher speed, compatibility with predicate-pushdown).

3.4.2 [De]compression Speed

We measured speed as the amount of tuples (i.e., values) that an algorithm is capable of [de]-compressing in one CPU clock cycle. In order to do so we took a vector within each of our datasets (i.e., 1024 values) and executed the [de]compression algorithms. The measure *tuples per cycle* is then calculated as 1024 divided by the number of computing cycles the process took. We chose one vector as the size of the experiment since every float compressor we compare against is optimized to work over a small block of values at a time; except Zstd. As such, we increased the size of the experiment for Zstd to one rowgroup (i.e. roughly 1

Table 3.4: Compression ratio measured in Bits per Value. The smaller this metric, the more compression is achieved (uncompressed data is 64 bits per value). ALP achieves the best performance in average (excluding zstd). *: ALP_{rd} was used.

Dataset	Gor.	Ch.	Ch. 128	Patas	PDE	Elf	ALP	LWC+ ALP	Zstd
Air-Pressure	24.7	23.0	19.3	27.9	30.2	10.5	16.5	11.9^{dict}	8.7
Basel-Temp	61.6	54.1	31.2	36.5	39.3	32.9	29.8	13.8^{dict}	18.3
Basel-Wind	63.2	54.7	38.4	48.9	35.1	34.5	29.8	10.3^{dict}	14.6
Bird-Mig	48.7	41.9	26.3	35.9	35.2	19.9	20.1	19.8^{dict}	21.0
Btc-Price	51.5	48.2	45.1	57.1	44.1	31.9	26.4	26.4	49.9
City-Temp	59.7	46.2	23.0	24.2	31.5	15.1	10.7	10.0^{dict}	16.2
Dew-Temp	56.2	51.8	32.6	39.0	29.5	17.7	13.5	13.5	20.9
Bio-Temp	51.9	46.3	18.9	22.9	23.4	13.0	10.7	10.7	14.5
PM10-dust	27.7	24.4	13.7	19.9	12.9	7.1	8.2	8.2	6.9
Stocks-DE	46.9	42.9	13.6	20.8	25.1	12.3	11.0	11.0	9.4
Stocks-UK	35.6	31.3	16.8	21.5	26.1	11.0	12.7	12.7	10.7
Stocks-USA	37.7	35.0	12.2	19.2	26.1	8.8	7.9	7.9	7.8
Wind-dir	59.4	53.9	27.8	28.2	31.5	22.1	15.9	15.9	24.7
TS AVG.	**48.1**	**42.6**	**24.5**	**30.9**	**30.0**	**18.2**	**16.4**	**13.2**	**17.2**
Arade/4	58.1	55.6	49.0	59.1	33.7	30.8	24.9	24.9	33.8
Blockchain	65.5	58.3	53.2	62.6	39.1	39.2	36.2	36.2	38.3
CMS/1	37.8	34.8	28.2	36.8	40.7	25.4	35.7	33.1^{dict}	24.5
CMS/25	65.4	59.5	57.2	70.1	63.9	48.6	41.1	27.1^{rle}	56.5
CMS/9	17.1	18.7	25.7	26.0	9.7	15.8	11.7	11.3^{dict}	14.7
Food-prices	40.8	28.0	24.7	28.3	25.4	16.8	23.7	23.7	16.6
Gov/10	58.1	45.7	34.2	35.9	35.6	30.1	31.0	31.0	27.4
Gov/26	2.4	2.3	9.3	16.2	0.9	4.2	0.4	0.2^{rle}	0.2
Gov/30	10.3	8.9	12.9	19.3	8.2	8.0	7.5	6.2^{rle}	4.2
Gov/31	5.7	5.0	10.4	17.1	2.8	5.4	3.1	2.5^{rle}	1.5
Gov/40	2.7	2.6	9.4	16.4	1.2	4.3	0.8	0.5^{rle}	0.4
Medicare/1	45.9	42.7	32.3	39.9	42.8	29.9	39.4	35.7^{dict}	28.7
Medicare/9	17.9	19.1	26.0	26.3	10.2	16.0	12.3	11.3^{dict}	14.9
NYC/29	30.8	29.6	28.7	38.8	69.3	32.6	40.4	24.7^{dict}	20.5
POI-lat	66.0	57.7	57.5	71.7	69.3	62.5	55.5*	55.5*	48.1
POI-lon	66.1	63.4	63.1	75.9	69.2	68.7	56.4*	56.4*	53.1
SD-bench	51.1	45.7	19.2	23.0	30.6	18.4	16.2	12.0^{dict}	11.8
NON-TS	**37.7**	**34.0**	**31.8**	**39.0**	**32.5**	**26.9**	**25.7**	**23.1**	**23.3**
ALL AVG.	**42.2**	**37.7**	**28.7**	**35.5**	**31.4**	**23.1**	**21.7**	**18.8**	**20.6**

Table 3.5: Average compression and decompression speed as tuples processed per computing cycle of all datasets on the Ice Lake architecture.

Algorithm	Tuples per CPU Cycle (Higher is better)			
	Compression	ALP is faster by:	Decompression	ALP is faster by:
ALP	0.487	-	2.609	-
Chimp	0.042	12x	0.039	66x
Chimp128	0.040	12x	0.040	65x
Elf	0.010	47x	0.012	215x
Gorilla	0.052	9x	0.047	55x
PDE	0.002	251x	0.387	7x
Patas	0.060	8x	0.157	17x
Zstd	0.035	14x	0.101	26x

MB of data). In order to correctly characterize CPU cost, we repeated this process 300K times and averaged the result, to ensure all data is L1 resident. In this experiment, we prefer the metric *tuples per cycle* over *elapsed time* since it is a more effective comparison method across platforms. Furthermore, this metric makes Zstd speed measurements comparable regardless of the input data size. This experiment was performed on Ice Lake.
Figure 3.1 shows the result of this experiment. ALP clearly outperforms every other algorithm in both compression and decompression speed in every dataset; even being able to achieve sub-cycle performance in decompression. This speed measurement also includes the FFOR encoding and decoding in ALP. Table 3.5 shows the average amount of tuples per cycle processed in compression and decompression for every algorithm along all datasets. ALP is faster than all other approaches in both compression and decompression.
ALP is ≈7x faster than PDE; which is the second-best at decompression speed. However, PDE is also the slowest at compression (251x slower than ALP) due to the brute force and –per value– search for a viable exponent e to encode the doubles as integers. Furthermore, ALP is ≈8x faster than Patas, which is the second-best at compression speed. This was expected since Patas is a single-case byte-aligned variant of Chimp optimized for decoding speed. On the other hand, Elf speed under-performed against the other algorithms, with ALP being ≈47x times faster in encoding and ≈215x faster in decoding. This was also expected since Elf is a variant of Gorilla tailored to trade speed for more compression ratios. Hence, the fact that ALP achieved higher compression ratios than Elf is remarkable. ALP is x55 faster than Gorilla at decompression since the latter has complex if-then-else (i.e. branch mispredictions) and data dependencies that not only cause wait cycles, but also prevent SIMD. Zstd resides in a middle position in that it achieves better compression speed than PDE and Elf, and decompression speed only slower than Patas and PDE.

ALP on Different Architectures. In order to investigate the performance robustness of ALP, we evaluated it on all currently mainstream CPU architectures, as described in Table 3.3. CPU turbo-scaling features were disabled when available to allow for reliable tuples-per-cycle measurements. In our presentation here we just show results for decompression speed (due to space reasons) as this is the most performance-critical aspect

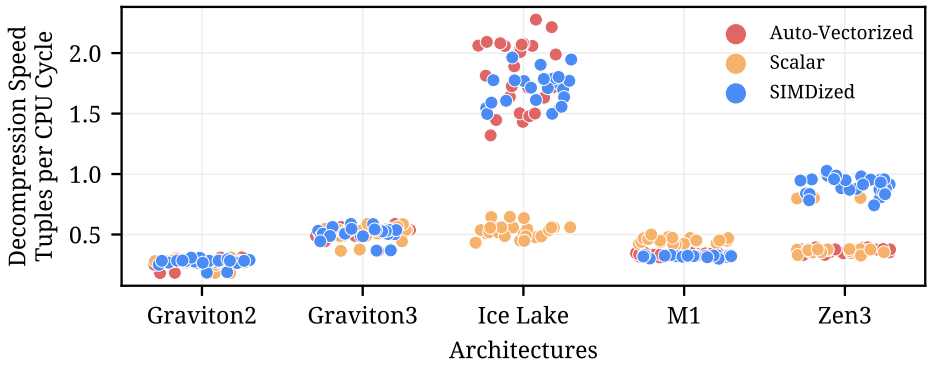


Figure 3.4: Decompression speed measured in tuples per cycle on different architectures. Each dot represents the decompression performance on a dataset in a different architecture.

for analytical database workloads. Furthermore, on each architecture we tested three different implementations of our decoding procedure: SIMDized, Auto-vectorized and Scalar. The SIMDized implementation uses explicit SIMD intrinsics. The Auto-vectorized implementation is the Scalar implementation automatically vectorized by the C++ compiler. Finally, the purely Scalar implementation is obtained when we explicitly disabled the auto-vectorization of the C++ compiler by using the following flags: `-O3 -fno-slp-vectorize -fno-vectorize`. Figure 3.4 shows the results of this experiment. We can see how Auto-vectorized and SIMDized on Ice Lake yield the best performance results. This is due to this platform having the widest SIMD register of all the platforms at 512-bits. We can also see that Gravitons have weak SIMD performance (compared to Scalar). Furthermore, in every platform Auto-vectorization matches or surpasses Scalar code. However, Zen3 auto-vectorized performance is hurt by the scalar code using the built-in rounding function due to the lack of a SIMD instruction to perform the cast from double to int64 in our fast rounding procedure.

Kernel Fusion. We performed speed comparisons of our decompression between `FFOR+ALP` as a fused kernel and as two separate kernels. The plot at the left of Figure 3.5 shows the result of this experiment. Fusing increases the decompression speed by a median ≈40% (but for some datasets 6x). However, the vectors from our datasets used for this experiment do not cover all the possible bit-widths that FFOR could use. The latter is a known factor that may affect the performance of vectorized execution [101]. Hence, for robustness purposes, we performed an additional comparison on synthetic integer vectors generated with a specific vector bit-width from 0 to 52. Bit-widths from 52 to 64 are omitted from this analysis since on these bit-widths ALP_{rd} is used. The right plot of Figure 3.5 shows the result of this experiment.

Sampling Overhead in Compression. ALP implements a two-level sampling mechanism to find the correct encoding method and parameters, described in section 3.3.2. The first level samples `row-groups` and the second level is done for every `vector`. We analyze the

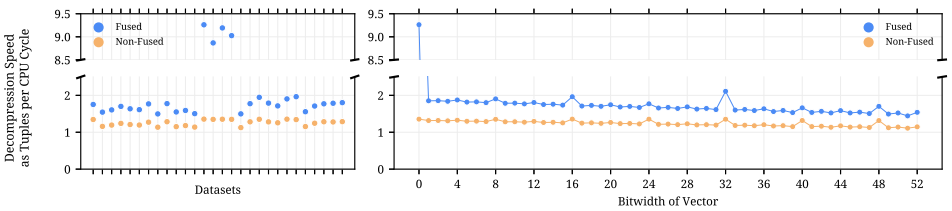


Figure 3.5: Speed comparison of ALP decoding with and without fusing ALP and FFOR into one single kernel (Ice Lake). Tests performed on our analyzed datasets (left) and on generated data with specific vector bit-width (right). ALP benefits from fusing consistently with a ≈40% decompression speed increase (and sometimes much more).

performance cost of the second sampling level, since it is on the performance-critical path of ALP compression.
When the first level sampling yields only one potential combination (e.g., Bird-Migration, Bitcoin-Price), there is 0 sampling overhead at a vector level for the entire `row-group` since ALP already knows which combination of exponent and factor to use for all the vectors. This occurs on ≈54% of the vectors in our datasets. However, when ALP has to perform the second-level sampling, there is a non-negligible overhead at compression. From our experiments, this overhead represents on average ≈6% of the total compression time. The latter is a trade-off for fast decompression; which in the context of analytical databases is a more often-used operation than compression. This overhead is bounded by k factor and exponent combinations, which was set to 5 in our evaluation. 22.9% and 20.0% of the vectors tried 2 and 3 combinations respectively in search of the best one. Only 2.9% and 0.3% of the vectors tried 4 and 5 combinations respectively on the vector sample.
Finally, we have also found that the best combination yielded from a brute-force search on the entire search space only improved compression ratio by less than 1% on average. Thus, demonstrating the efficiency and portability of our fixed sampling parameters across all datasets.

ALP$_{rd}$ speed. Doing a side-by-side comparison ALP$_{rd}$ is on average ≈3x slower in compression and ≈4x slower in decompression than the main ALP encoding. In fact, the two datasets in which ALP$_{rd}$ was used can be seen at the bottom of ALP green dots in Figure 3.1. Although ALP$_{rd}$ is still remarkably performant compared to the competitors, we deem this speed reduction necessary to achieve compression on these types of doubles, which present problems for every floating-point compression scheme. We believe there is room for improvement since *ALP$_{rd}$* encoding and decoding are not fused into one single kernel due to current implementation limitations. However, given that [de]compression in almost any encoding gets faster at high compression ratios, this result is not surprising: ALP$_{rd}$ is used when only low compression ratios can be achieved (maximum ≈1.2x).

3.4.3 End-to-End Query Performance

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [61]. The difference with our micro-benchmarks is that a complete dataset is decompressed by Tectorwise's scan operator

Table 3.6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core. ALP is even faster than uncompressed, and extends its lead w.r.t. the micro-benchmarks. The competitors are so CPU bound that they scale well in SCAN (=speed stays equal), while ALP and uncompressed drop speed when running multi-core, due to scarce RAM bandwidth. But when doing query work (SUM), speed is lower, and scaling is not an issue for ALP.

Algorithm	Tuples per CPU Cycle (Higher is Better) QUERY | THREADS						
	SCAN|1	SCAN|8	SCAN|16	SUM|1	SUM|8	SUM|16	COMP
ALP	1.337	1.074	0.882	0.233	0.230	0.234	0.147
Uncompressed	0.565 [x2 slower ↓]	0.408	0.350	0.197 [x1.2 ↓]	0.186	0.175	N/A
PDE	0.070 [x19 ↓]	0.071	0.071	0.058 [x4 ↓]	0.057	0.057	0.001 [x138 ↓]
Patas	0.067 [x20 ↓]	0.063	0.065	0.055 [x4 ↓]	0.055	0.055	0.039 [x4 ↓]
Gorilla	0.030 [x44 ↓]	0.030	0.030	0.028 [x8 ↓]	0.027	0.027	0.023 [x7 ↓]
Chimp	0.021 [x64 ↓]	0.021	0.021	0.019 [x12 ↓]	0.019	0.019	0.015 [x10 ↓]
Chimp128	0.028 [x47 ↓]	0.028	0.028	0.026 [x9 ↓]	0.026	0.026	0.019 [x8 ↓]
Zstd	0.044 [x31 ↓]	0.042	0.039	0.038 [x6 ↓]	0.037	0.035	0.014 [x11 ↓]

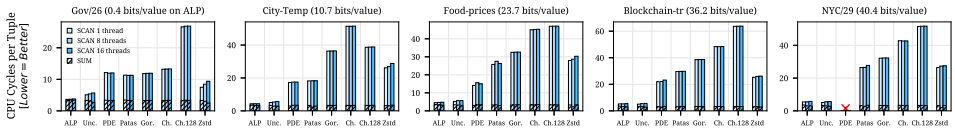


Figure 3.6: End-to-end SUM query execution speed for 5 datasets in Tectorwise (Ice Lake) measured in CPU cycles per Tuple. ALP is faster than all other schemes (even faster than *uncompressed*), while achieving perfect scaling (=speed stays the same) when using multi-core. Results show that SCAN is virtually free if data is compressed with ALP. PDE can't compress NYC/29.

(SCAN), rather than only a small part. Also, in the SUM experiment, the scan operator feeds data vector-at-a-time into an aggregation operator; using the vectorized query execution of Tectorwise. We scaled all datasets up to 1 billion doubles by concatenation (8GB uncompressed). We also test compression performance, which writes the compressed data. This also writes extra meta-data for the compressed blocks, at the least byte-offsets where they start, but for PDE and ALP also offsets where their exceptions start, as well as any other compression parameters (like bit-width for bit-packing).
For presentation purposes, we picked five datasets with diverse characteristics, such as magnitude, decimal precision, XORed 0's bits, and compressability. These datasets are: Gov/26, City-Temp, Food-Prices, Blockchain-tr and NYC/29. We benchmarked 3 queries: COMPRESSION (`COMP`), `SCAN` and `SUM` (Aggregation). For `SUM` and `SCAN` we also benchmarked the scaling of every algorithm when using multiple cores (up to 16). This experiment was again carried out on Intel Ice Lake in a machine with 16 cores (32 SMT) and 256GB of RAM with a bandwidth of 18.75 Gibps. The reported results are the average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

SUM and SCAN. Table 3.6 shows that in the single-threaded `SCAN | 1` experiment, the achieved 1.33 Tuples per CPU cycle is in line with the microbenchmarks shown in Figure 3.5 – though there is about a 25% drop in performance in the end-to-end situation compared to these. We attribute this to: (i) the extra effort in reading block meta-data (not present in the

micro-benchmarks), (ii) the interpretation cost of choosing and calling a decompression function based on the meta-data (always the same and thus free of CPU branch mispredictions in the micro-benchmarks) and (iii) the variable amount of exceptions present in the entire dataset.
Given these extra activities in end-to-end, and just a 25% drop, we deem our micro-benchmarks as representative of *core* decompression work achieved in end-to-end situations. What is further striking is that SCAN and SUM on ALP is faster than on uncompressed data, and the fact that ALP extends its performance lead over the competitors in the end-to-end benchmarks, compared to the micro-benchmarks. Note, however, that the micro-benchmark results were aggregated for all datasets (Table 3.5) so one should not directly compare with these tables.
Regarding multi-threading, the performance metrics in Table 3.5 and Figure 3.6 are *per-core*, hence equal performance would be perfect scaling. As all cores of the CPU get loaded, per-core ALP SCAN performance slightly drops – which also happens for uncompressed. This is caused by the query becoming RAM-bandwidth bound. However, in the SUM experiment, there is additional summing work (although not much) and therefore the query runs slower. As a result, ALP is able to scale perfectly while uncompressed is not.
Note that in Figure 3.6 the performance metric is reversed: lower is better. We present the summing work in the SUM query (=SUM-SCAN, because SUM also scans) as the lower part of the stacked bar: it is roughly 3 cycles per tuple. Figure 3.6 confirms our results across the board: ALP is much faster end-to-end than the other compressors, even faster than uncompressed, and scales well.

COMP. ALP again is the fastest when compressing (Table 3.6): it is x4 and x7 times faster than the second and third-best algorithms in the City-Temp dataset (i.e. Patas, Gorilla) while still maintaining distance from Zstd (x11 slower) and PDE (x138 slower). `COMP` end-to-end performance is lower than in our micro-benchmarks. We attribute this to: (i) the extra effort in storing meta-data, (ii) the variable amount of exceptions (which are rather costly at compression time) and (iii) the first sampling phase which was not present in the micro-benchmarks.

3.4.4 Single Precision and Machine Learning Data

We have also ported *ALP* to 32-bits. Those of our double datasets with decimal precision $\leq$10, can be properly represented as 32-bit floating-point numbers (all except POI's, Basel's, Medicare/1, and NYC/29); and 32-bit ALP works on them. This leads to the same compressed representation as in 64-bits (Table 3.4); but given that the uncompressed width is 32-bits, the compression ratio is halved (and becomes $\approx$1.77).
A currently relevant different kind of 32-bit floats are found in trained machine learning models (i.e., the weights). However, these were created out of many multiplications and additions, and hence tend to have high precision. Still, there will be commonalities in their sign and exponent parts (IEEE 754) that ALP_{rd} could take advantage of. Therefore, we also ported ALP_{rd} to 32-bits and benchmarked it on four different ML models, against those competing schemes that have a version for 32-bit floats (i.e. Gorilla, Chimp, Chimp128, Gorilla) as well as Zstd. The results of this experiment are in Table 3.7; with ALP_{rd} for 32-bit floats achieving the best compression ratios out of all the other algorithms (28.1

Table 3.7: Compression ratios (bits/value) that $ALP_{rd}32$ and its competitors achieved on machine learning models' weights (32-bits floats). $ALP_{rd}32$ achieved the best compression ratio.

Name	Model Type	N° of Params.	Gor.	Ch.	Ch. 128	Patas	ALP$_{rd}$	Zstd
Dino-Vitb16 [117]	Vision Transformer	86,389,248	34.1	33.4	33.4	45.8	28.3	29.7
GPT2 [118]	Text Generation	124,439,808	34.1	33.5	33.5	45.6	27.7	29.7
Grammarly-lg [119]	Text2Text	783,092,736	34.1	33.4	33.4	45.5	27.7	29.6
W2V Tweets	Word2Vec	3,000	34.1	33.3	33.3	45.5	28.8	29.8
		AVG.	34.1	33.4	33.4	45.6	28.1	29.7

bits/value; ≈12% of reduction). In fact, it is the only floating-point encoding to achieve compression. Alternatively, model weights are usually quantised (i.e. lossy reduction of precision) when deployed for inference[116]. However, if this is not desired or possible; ALP_{rd} thus can provide some useful lossless compression for ML.

3.5 Related Work

The techniques developed for floating-point compression can be categorized mainly into three groups: (i) Predictive schemes, (ii) XOR schemes and (iii) Integer encoding schemes.

Predictive Schemes were one of the first novel approaches designed to compress floating-point data [85, 120, 121]; even in the context of geometry data [122, 123]. In these approaches, a *function* is used to generate a *predicted value* based on patterns found within the data prior to the value to encode. The idea behind this approach is that the predicted value and the value to encode are similar enough such that an operation (usually ADD) between their exponent and mantissas represented as integers yield a compressible chain of bits. Ratanaworabhan et al. [110] demonstrated that such an operation could be a bitwise XOR. Based on that, Burtscher and Ratanaworabhan developed FPC [111], which achieved better compression ratios and speed compared to previous approaches.

XOR Schemes. Pelkomen et al. [95] re-evaluated the predictor function to obtain a similar value to the value to encode. Their key idea was that in certain contexts such as time series, using the immediate previous value works as well as using a predictor. This assumption motivated the development of Gorilla. Gorilla compresses floats by doing a bitwise XOR with the immediate previous value. Next, it encodes the resulting chain of bits as 0 in case of a perfect XOR (i.e. equal values), otherwise, it encodes the resulting number of leading zeros and significant bits. Gorilla is faster on [de]compression than prediction schemes since encoding and decoding are achieved using a simple XOR with the immediate previous value instead of tuning and running a prediction function.

Gorilla Variants. Chimp [96] refined Gorilla by exploiting properties of the bit-chains yielded by the XORing process in time series data. Chimp distinguishes four different encoding modes based on the number of leading and trailing zeros of the XOR result to optimize compression ratios. It was jointly developed with a variant called Chimp128 in which the algorithm looks into the previous 128 values in order to find the *most suitable* value to XOR at the expense of 7 additional bits to store the position of this value. This idea

of looking among previous values for the XOR was first introduced by Bruno et al. [124]. Chimp128 proved to be substantially better than FPC, Gorilla and other general-purpose compression schemes (e.g. Snappy, LZ4) in terms of compression ratio and speed [96]. In order to improve Chimp *decompression speed*, DuckDB Labs developed Patas [97]. The goal was to get a variant of Chimp128 faster at [de]compression, which it achieves by its design with a single encoding mode (fewer branch-mispredictions) and byte-aligned bit-manipulation (less CPU work). Patas encodes for every value a block of 2 bytes containing the 7 bits previous value index, the number of significant bytes and the number of trailing zeros. Patas trades compression ratio for a ≈75% speed improvement at decompression time compared to Chimp128. In the context of analytical databases decompression speed is important for obtaining fast query results. On the other hand, a recently proposed XOR scheme called Elf [43] trades [de]compression speed for compression ratio by erasing bits from the mantissa at encoding time to make the XOR result more compressible. Afterwards, it losslessly reconstructs the double at decoding. As seen by our results, Elf gains ≈19% in compression ratio over Chimp128 at the expense of ≈4x slower compression and decompression. In contrast to Patas and Elf, ALP improves Chimp128 in all aspects.
While Chimp128 seemed to be clearly superior to Gorilla, our results show that it can actually perform better than Chimp128 (and even Elf) in datasets with consecutive runs of zeros (e.g. Gov/26, Gov/40). On this type of data Gorilla (and also Chimp) do not need the extra 7-bits to make a reference to one of the past 128 values since the most optimal value to XOR is always the previous one. Hence, Chimp128 is not always the best XOR-based encoding. It does depend on the data characteristics.

Integer Encoding Schemes. Doubles can also be compressed by taking advantage of their visible decimal representation [125]. PseudoDecimals [83] (PDE) formally introduces a lossless approach to perform this encoding process. PDE tries to encode a double with a division between an integer and an inverse factor of 10 under the assumption that the double was generated from a `DECIMAL`. This is why we refer to this type of encoding as Decimal-based encoding. ALP presents a strongly enhanced version of this approach introducing the idea of using large exponents and mitigating the effects of those with an additional multiplication that gets rid of trailing zeros. ALP is designed for *vectorized execution*, and introduces an adaptive mechanism for high-precision decimals (i.e. ALP_{rd}). ALP prefers *multiplication* over *division* since division is an expensive operation in most ISAs [126]. PDE and ALP have the advantage that their output is further compressible using other lightweight encoding schemes such as `DICTIONARY`, `RLE`, `FOR` or `DELTA` [83, 84, 101].

3.6 Discussion

A striking feat of our study of datasets used for database compression of doubles is that out of the 30 datasets our community uses for evaluating double compression, only the two POI datasets would not better be represented as fixed-point decimals. In fact, most POI data comes from GPS, which has an accuracy of a few meters, and the Earth's diameter is ≈12.750.000 meters (i.e., 8 digits, which corresponds to 28 bits). Indeed, when the POI-lat and POI-lon values are converted back from radians by multiplying with $\pi/180$ we observe this precision in the data – but we think it would go too far to define a specific ALP mode

that deals with *pi*-multiplied data.
One may question why none of the datasets requires true double precision, nor is any all over the place in terms of magnitude – doubles allow numbers as close to zero as 10^{-308} and as large as 10^{308}. One interpretation could be that double is a catch-all type for two use cases: storing measures for which a-priori little is known about their domain (min/max), or where the magnitude is truly wide and/or variable. In the former use case, the actual data will tend to have min/max locality, leading to low variance in the high bits (equal or close exponent and highest mantissa bits). As the actual precision of actual values is limited by the measurement method, one either sees "pseudo-decimals" where the lower digits (in 10-base) are zero, or in the worst case, randomly filled in. The latter use case, high magnitude variance, seems to be rare, though weights and activations in machine learning could be the best example of this (not regarding large numbers, but numbers close to zero, i.e., highly variable negative exponents). Such data demonstrated to be hard to compress, for any scheme; and reducing their size is so crucial that it triggered the appearance of TensorFloat (Google) and Bfloat16[12] (Nvidia). These new thin floats, developed with Machine Learning hardware in mind, mostly cut down on mantissa and somewhat on exponent.
The use of doubles in scientific calculations is common; though researchers have criticized the rounding errors produced [112], and proposed alternatives like unum and posit[127]. There are strong arguments for compressing doubles stored in big data formats and database files: data gets smaller, reducing storage cost across the memory hierarchy, reducing also I/O time, network transfer time and usage. We think that with the increased convergence of data science and scientific computations there will be growing demand for doubles in databases, and their compressed storage.

3.7 Conclusions

We have presented and evaluated ALP: a strongly enhanced version of Decimal-based encoding with an adaptive fallback to front-bit compression if doubles have truly large precision. ALP beats the competition in all relevant dimensions. Its compression ratio is better than all recently proposed floating-point encodings, while being *much* faster in [de]compression speed. Its compression ratio is only equalled by heavy-weight general-purpose compression; but these methods have slow [de]compression speeds and are block-based: forcing database scans to fully compress a large block of data. In contrast, one can skip through ALP-compressed data at the vector-level; allowing for efficient predicate push-down. We think ALP will be a valuable encoding in *cascading* lightweight compression formats [83, 84], and recall that in our evaluation it already beat zstd (18.8 vs. 20.6) when cascading on Dictionary and RLE.
We would like to stress that the key idea behind ALP is to design for *vectorized execution*; it led us to analyze and uncover unexploited opportunities from a vector perspective in a variety of datasets. Vectorized execution reduces computational cost (reducing loop-, function call-, and load/store-overhead), brings out the best in compilers (vectorized code triggers loop-centered optimizations including auto-vectorization), but also amortizes storage (parameters such as exponent are stored once per-vector instead of per-value), allows for per-vector adaptivity without reducing performance due to branch-mispredictions (as

[12] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

happens in per-value adaptivity in e.g., the Chimp variants), and can take advantage of in-vector data commonalities.
As for future work, we think that the implementation of ALP on massively parallel hardware such as GPUs and TPUs could be fruitful.

4

Data-Parallelized Encodings on GPU

We show that compression can be a win-win for GPU data processing: it not only allows to store more data in GPU global memory, but can also accelerate *data processing. We show that the complete redesign of compressed columnar storage in FastLanes, with its fully data-parallel bit-packing and encodings, also benefits GPU hardware. We micro-benchmark the performance of FastLanes on two GPU architectures (Nvidia T4 and V100) and integrate FastLanes in the Crystal GPU query processing prototype. Our experiments show that FastLanes decompression significantly outperforms previous decompression methods in micro-benchmarks, and can make end-to-end SSB queries up to twice faster compared to uncompressed query processing – in contrast to previous work where GPU decompression caused execution to slow down. We further discovered that an access granularity of decoding vectors of 1024 values is too large for a single GPU warp due to register pressure. We mitigate this here using mini-vectors – a future work question is how to further reduce this granularity with minimal impact on efficiency.*

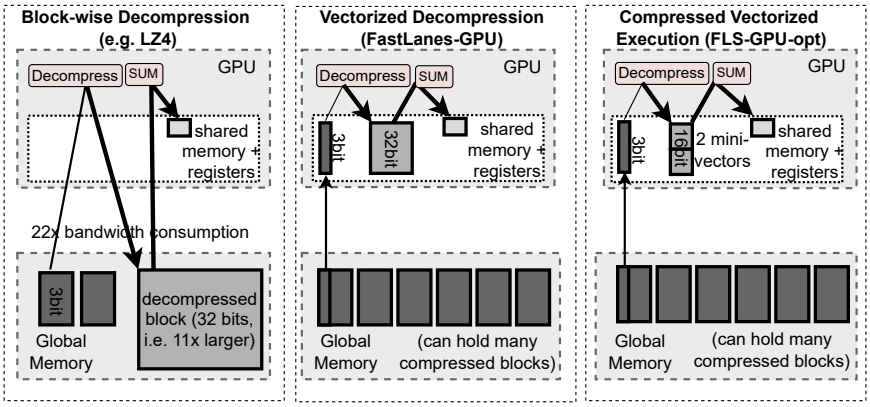


Figure 4.1: Three different ways of decompressing data on the GPU. Left shows that decompressing into global memory (as typically done in GPU decompression) can cause very high memory bandwidth consumption. The middle shows how vectorized decompression avoids spilling back to global memory, by directly processing the decoded data. The rightmost shows how the pressure on GPU registers and shared memory (cache) can be reduced, by (i) reducing the decoding batch size, and (ii) by decoding into thin (<32-bit) data types.

4.1 Introduction

The FastLanes project[1] is working towards a new analytical data format to better suit modern workloads and modern hardware. In the first chapter on FastLanes, that describes its novel and completely data-parallel columnar encodings, we showed its very high performance using data-parallel SIMD instructions on CPUs [128].

In this chapter, we evaluate and optimize FastLanes decompression on GPUs. In the past decades there has been a lot of research on database processing on GPUs and also numerous start-ups; but database workloads have not migrated to GPUs yet; instead, a flood of ML workloads have propelled GPUs to center stage in data centers. These workloads consume a steady stream of data in ML training and inferencing, producing a growing call for data formats that are compatible and performant on both CPUs and GPUs [129].

Data formats that are now ubiquitous in data lakes, such as Parquet and ORC, were originally designed for use on CPUs [129]. While these formats harbor multiple good ideas (schemas, statistics, columnar storage, compression, vectorized decoding) they have limitations that hurt GPUs; particularly the fact that their column encodings do not compress data enough, and therefore their data-pages are further compressed with general-purpose block-based schemes (gzip, zstd, lz4 or snappy), which are GPU-unfriendly.

Data compression is an attractive proposition for GPUs: they typically have smaller RAM ("global memory") than the host CPU machine, such that storing compressed data alleviates a capacity bottleneck. Further, data is moved into the GPU over the PCIe bus, so having to move less data thanks to compression helps to reduce that bottleneck. But this hinges on the capability of the GPU to efficiently decompress the data, ideally incrementally, when it is processed. As the left picture in Figure 1 shows, however, block-based compression operates

[1] github.com/cwida/FastLanes

on a coarse granularity that is too large to fit into the shared memory (the on-die GPU cache), whose size is typically just tens of KB, shared among 32 threads. Note that e.g. parquet-mr typically creates pages of 1MB, that get (de)compressed as one block, which exceed this size. This means that the uncompressed result of decompression must spill back to GPU global memory, significantly increasing the memory bandwidth usage inside the GPU, because the decompressed data is much larger than the compressed data. In the depicted example, when a compressed column that takes 3 bits per value, is decompressed into the standard 32-bit integer that GPUs manipulate; this will transfer in total 22x the compressed bandwidth (3+32bits for decompressing, plus 32bits upon use). There are three problems here: (i) decompression algorithms like lz4 are essentially sequential (have many control- and data-dependencies) and therefore run inefficiently on GPUs (ii) materializing buffers of uncompressed data in global memory as depicted, wastes scarce global memory capacity. (iii) transferring uncompressed data for processing (here SUM) into GPU kernels, can make these bandwidth-bound.

In FastLanes we pursue cascading (recursive) application of column encodings (FOR, DELTA, RLE, DICT) to remove the need for general-purpose compression like lz4 for getting good compression ratios [130]. FastLanes encodings are fully data-parallel, even eliminating the sequential dependencies that are normally present in DELTA decoding and RLE. This data-parallelism works great on CPU SIMD instructions but conceptually also fit the SIMT GPU model, where 32 threads that make up a warp and execute the same instructions in lockstep; without any data-dependencies between the threads. FastLanes stores data in 1024-value vectors, where 32 or more adjacent values can be decompressed completely independently of each other. We propose the integration of FastLanes decoding in GPU data processing as depicted in the middle of Figure 1 using *vectorized decompression*: decompression as the first step of data processing, where one vector of data is decompressed into registers or shared memory and is directly consumed from there, without spilling back to global memory.

In this chapter, we show that FLS-GPU decompressing a vector of 1024 values can easily be too coarse-grained for GPUs. If each of the 32 threads in a warp decodes 32 values, these should be stored in GPU registers or shared memory, in order not to spill into GPU global memory. However, depending on the GPU architecture, this can already be close to the average available registers per thread (see last row of Table 4.3). Worse, the consuming data processing task (like a database query, or ML inferencing) typically needs *multiple* columns, which increases memory pressure on registers and shared memory – which may also hold e.g. lookup-tables. This then causes GPU register spilling, leading to increased memory latency and/or a reduction of scheduled tasks, leading to GPU under-utilization. Therefore, we developed GPU-specific optimizations (FLS-GPU-opt) that reduce register and shared memory pressure. Key ideas are: (i) dividing a vector into mini-vectors and decompressing mini-vector at-a-time; (ii) thinking beyond the standard 32-bits GPU data type and simulating smaller data types; thus considering 16- and 8-bits data widths. We (iii) also increased the block-width from 32 to 128 or even 256 in order to reduce scheduling overhead.

Our main **contributions** in this chapter are:

- **Generating and Micro-benchmarking FLS-GPU code.** We use a code generator to generate the C++ FastLanes CPU encoding and decoding methods for all relevant

bit-widths statically at compile-time. This code generator was extended such that it can generate CUDA code. We show that FLS-GPU outperforms the current state-of-the-art GPU decompression algorithms GPU-FOR and GPU-DFOR by performing micro-benchmarks where we decompress into *global memory*, *shared memory* and GPU registers.

- **Integrating and Optimizing FastLanes in Crystal.** Further, FastLanes bit-unpacking is integrated into Crystal [131]. To increase performance we tested *compressed execution*, partitioning 1024-tuple vectors into *mini-vectors*, increasing the block size, and the sorting of columns to simulate RLE and obtain a better compression ratio. These optimizations enable to release pressure from registers and *shared memory*, and are ultimately combined with optimizations proposed by Crystal-opt [132].

Outline. First, we discuss Crystal, Crystal-opt and Tile-based decompression: the fastest state-of-the-art GPU academic database systems and decompression scheme respectively. We also summarize the FastLanes layout that uses *interleaved* bit-packing and the *transposed layout* to eliminate dependencies from the DELTA and RLE encodings. Then, we discuss how we adapted FastLanes to be compatible with GPUs and we propose optimizations to improve the performance of FastLanes on Crystal. Lastly, we discuss the obtained results and share our findings and ideas for future work to further optimize FastLanes for GPU-based data processing.

4.2 Background

In this section, we shortly explain the GPU memory hierarchy and basic principles of `CUDA`. In addition, the lightweight compression (LWC) algorithms used by FastLanes are briefly explained, along with an explanation of the intrinsics of FastLanes en/decoding.

4.2.1 GPU Programming

In this chapter we use CUDA for programming the GPU [133]. CUDA only works for NVIDIA hardware, so in an increasing heterogeneous hardware landscape, a more portable API such as Vulkan [134] could be considered. However, CUDA offers a more mature toolchain and higher-level programming model; hence we use it for this initial study of FastLanes on GPUs.

CUDA splits the written code in two parts: the *device code*, used to program the GPUs itself, and the *host code*, which is CPU code. The host code takes care of initialization and launching of the kernels in the program and allocating memory regions. The device code is written as a sequential program, thus for a single thread, but executed for multiple threads at once, a model known as Single-Instruction-Multiple-Threads (`SIMT`). CUDA *virtualizes* physical hardware. A *thread block* in CUDA is a virtualized streaming multiprocessor (SM). It is typically recommended to use at least 128 threads in a block, to limit scheduling overhead.

Each SM contains cores, a register file, a warp scheduler, data caches, instruction buffers and texture units. A SM can therefore be considered a whole machine on itself. Thread blocks are launched on a single SM and are independent of each other, meaning that they run to completion without preemption. A *thread* itself is a virtualized scalar processor which

contains its own registers. Threads are grouped into *warps*, which is the basic unit of execution for a single SM. A warp is a unit that consists of 32 threads that all run concurrently on a SM. All threads in a warp execute the same instruction when running a kernel. If the kernel contains branches (if-then-else), *thread divergence* can occur if some threads take the if- and other the else-branch. The threads must execute all instructions in lock-step, but the instructions off the chosen paths become no-ops. This also affects while-loops: all threads execute as long as the longest loop in the warp.
The memory hierarchy of a GPU differs significantly from the CPU memory hierarchy. In the memory hierarchy of the GPU, each thread contains its own 32-bit registers. Next to the amount of registers per thread, each thread contains its own *local memory* (lmem). Lmem is not really a memory – its bytes are stored in the GPU main memory (global memory). The name local memory refers to the memory where registers and other data from a thread is spilled, when e.g. a thread exceeds the register limit. The main differences between lmem and global memory however are (1) stores are always cached in L1 cache and (2) addressing is resolved by the compiler itself. If the L1 cache is full, a line gets evicted to L2 cache or DRAM. In this case, a store incurs multiple writes.
Each thread block contains its own *shared memory* which functions as a programmable L1 cache of usually a few tens of KB. Shared memory thus stores data which is accessible for *all* threads within a thread block and gets wiped when a new block is executed.
Unlike the L1 cache, the L2 cache is shared among all SMs. There is *global memory*, which is accessible to *any* thread at all times. Global memory is the main memory where data is stored when loaded from the CPU into the GPU. The bandwidth to the GPU is typically a few factors higher than main memory attached to a CPU (often using HBM technology). Data is transferred using wide cache-lines (typically 128 bytes), and best access is achieved if the threads in a warp load adjacent data (*coalesced* memory access).
Transferring data, and communication between host (CPU) and device (GPU) in general, happens via the PCI/e bus. However, such I/O should be minimized where possible as the PCI/e bandwidth is much lower than global memory bandwidth.

4.2.2 Lightweight Compression using FastLanes

This section explains state-of-the art approaches of different lightweight compression schemes on the GPU. In addition, it explains how FastLanes internally works to provide efficient en/de-codings for these lightweight schemes.

Common Lightweight Compression Methods. Analytical database systems make extensive use of compression to reduce memory footprint when storing and accessing data. However, general-purpose compression methods such as Snappy or LZ4 are computationally expensive for decompression at runtime. Therefore, lightweight compression methods or *encodings* such as Bit-packing, DICT, DELTA and Run-Length Encoding (RLE) are typically used as a first step. Exploiting the fact that the the actual domain of data stored close together is often much smaller than what their data type can represent, Bit-packing allows to represent larger data types in fewer bits. It is typically the lowest-level encoding applied to stored data. The other encodings work on top of bit-packing. DICT uses a dictionary, holding unique values, and represents the original data as (bit-packed) integer codes, which are positions in this dictionary. DELTA encoding exploits value locality, by

only storing the (bit-packed) difference between subsequent values. Note that during DELTA decoding subsequent values are dependent on its predecessor (a data dependency). RLE encoding, finally, is particularly effective on sorted data with lots of repeating values. It encodes the repeating values in so-called `run-lengths`. Note that decoding RLE requires a loop over the run-length, which on GPUs can lead to thread divergence .

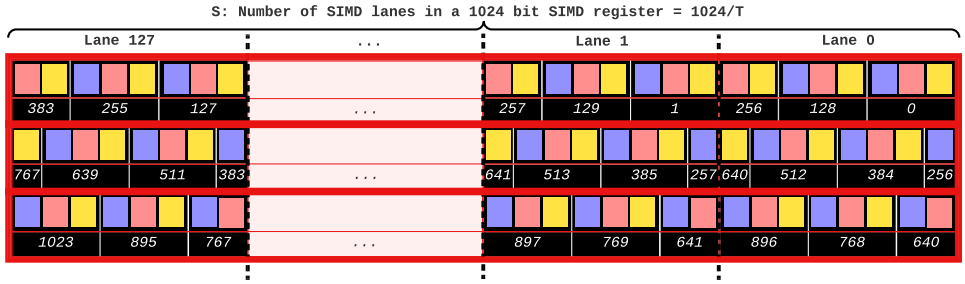


Figure 4.2: FastLanes bit-packs a vector by interleaving its values (vector positions in black below) over many lanes; in this example 128 lanes of 8-bits [128]. On GPUs this leads to all threads in a warp accessing directly adjacent lane data, which is the optimal memory access pattern.

FastLanes. FastLanes is an open-source library that provides lightweight encodings, in a fully *data-parallel manner*. Even though its CPU source code is scalar, this property allows compilers to *auto-vectorize* it into SIMD instructions, such as AVX512 on AMD and Intel, and NEON or SVE on ARM. SIMD instruction on CPUs can execute an operation on multiple data items, stored adjacently in 8-, 16, 32- or 64-bit *lanes*, in one instruction.
The first step in decoding is *un-packing* densely packed bits into 8-, 16-, 32- or 64-bits (byte-addressable) integers. Standard bit-packing schemes, however, pack bits of adjacent values tightly together right after each other. In principle, SIMD instructions can perfectly support bit (un)packing as they support the required operations (AND, OR, SHIFT). The problem here is that to produce the original value sequence with SIMD instructions, adjacent values will reside in the same SIMD lane. To avoid this, *interleaved* bit-packing distributes subsequent values round-robin over subsequent lanes. Thus, a data-parallel unpacking kernel can produce subsequent values from subsequent lanes, without needing (expensive) inter-lane data transfers.
After values have been bit-unpacked; they can be decoded using lightweight schemes like DICT, RLE and DELTA. The latter two schemes have proven challenging for SIMD instruction sets, because of the sequential dependencies in DELTA and the loops required to decode RLE (SIMD does not support loops). For example, if we encode a column of [1, 2, 3, 4, 5] into [1, 1, 1, 1, 1] using DELTA, we can only restore the original values if we know the predecessors of the value that we want to decode. To tackle this issue, FastLanes proposes the *Unified Transposed Layout* that removes the data dependency by using a special permuted order for the 1024 tuples in a vector (Figure 4.3 shows a simplified permuted order for 16 tuples). Thanks to this, every lane produces independent running sums to perform DELTA decoding. FastLanes further maps RLE coding to DELTA coding, to also make RLE fully data-parallel [128].

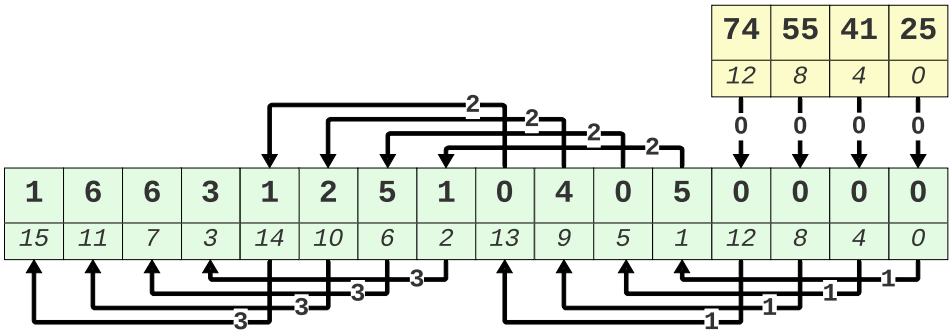


Figure 4.3: Transposed Layout of a 16-value vector for making DELTA decoding data-parallel[128]. Starting with a base stored in a header (yellow), DELTA has to sum up all differences, a sequential task. By storing multiple bases, and reordering the tuples in vector (small below numbers are positions) we can now execute this with four 4-way SIMD additions. The FastLanes layout (that stores 1024 values, not 16) provides enough parallelism to let all GPU threads in a warp do completely independent sums.

By removing all data dependencies, FastLanes achieves ultra-high performance [128]: up to 60 values per CPU cycle single-threaded. We think this is important, because in the vectorized decompression model, most of the computational time should go in the data processing (i.e. database query processing, or machine learning), and only a minority in decompressing, if we want to alleviate a memory bottleneck and make data processing faster. Ultra-fast decoding also facilitates our move towards *cascading* column encodings; that implies multiple decoding kernels are invoked per column – this otherwise could become expensive. As an example of cascading encoding, the thin integer codes arrays used in DICT, as well as its dictionary array can be further encoded, e.g. using RLE or DELTA. The work in BtrBlocks [130], which does not use data-parallel encodings, has established that cascading lightweight encodings can achieve compression ratio's comparable to Parquet with LZ4. In ongoing work on FastLanes, that adopts cascading encoding, we have confirmed this finding.

4.3 Related Work

There has been only a modest amount of research into compression schemes for GPU-based database systems. *Fang et al.* [135] implement nine compression schemes suitable for the GPU. These schemes are divided between main schemes and auxiliary schemes. These schemes include null-surpression (NS) of fixed and variable length, DICT, bitmap and RLE. Auxiliary schemes include FOR, DELTA, SEP and SCALE. *Shanbhag et al.* [136] propose GPU-FOR, GPU-DFOR, and GPU-RFOR where the latter two do respective DELTA and RLE. None of the previous GPU compression schemes [135–137] has changed the value order (i.e. interleaving or transposing) to make decompression data-parallel.

Crystal. [131] is the current state-of-the-art academic GPU-based query processing prototype [132]. It consists of hard-coded CUDA kernels implementing each of the SSB

benchmark queries [138]. To benefit from the parallelism provided by GPUs, Crystal adopts a *tile-based* execution model. Here, a *Tile* consisting of 128 tuples is the basic a unit of execution and is processed as a whole in a single thread block, which in its turn is partitioned by the GPU into warps. Processing at the granularity of tiles aligns with the 128-byte GPU cache lines and leads to reduced cache misses and *coalesced memory accesses*. Before performing any operations, a tile of 128 values is loaded directly into the available registers of each thread. Loading a tile enables coalesced memory access and loading into registers avoids an extra pass to shared and global memory. Crystal is not a production system, since all queries are hand-written. Other significant limitations are e.g. hard-coded parameters for hash tables and the lack of support for data types other than 32-bits integers.

Crystal-opt. The authors in [132] pointed out that Crystal is memory-bound, and there is room for improvement. Crystal (i)) loads unnecessary data from DRAM if filter predicates are selective and (ii) it bypasses the L1 cache which could be exploited for operating on intermediate results [132]. In Crystal-opt, query performance is improved roughly 2x, making it currently the fastest GPU-based query execution prototype for SSB queries. The first optimization is predicated loads (*PredLoad*, essentially predicate push-down) that avoid loading tuples from global memory that are already disqualified by a predicate. While such an if-then-else test introduces GPU thread divergence, doing so for avoiding unnecessary loads is a good trade-off. A somewhat less important optimization is to check whether an entire tile has no more qualifying tuples. This is only beneficial if *all* threads in a warp can be terminated early (a quite rare phenomenon). A second optimization to disable L1 cache bypass on certain queries (manually), for those queries that profit from this (a manually tuned decision).

Tile-Based Decompression. [136] introduced *tile-based decompression* to reduce memory bandwidth consumption in Crystal data scans. Three new compression schemes are introduced, based on the tile-based execution model that combine bit-packing and FOR: GPU-FOR, GPU-DFOR and GPU-RFOR. The latter two denote DELTA and RLE, respectively. Bit-unpacking in tile-based compression is a generic function that has bit-width as a parameter. For any tuple, it loads two values, computes two shifts and two AND-masks to apply to these values, and OR-s them together; which is a worst-case approach in terms of bit-unpacking. In contrast, a typical CPU-based vectorized bit-unpacking function would be templated by bit-width (rather than receive it as a dynamic parameter) and internally contain a series of hard-coded LOAD-SHIFT-AND-STORE instructions (with occasional OR work, only applied to those bit-sequences that cross a word boundary) to unpack all tuples in a vector – which is computationally much more efficient than computing the shift amounts and the masks for each tuple and always doing an OR. The authors argue that GPU decompression is bandwidth-bound and thus such computational expense is of no importance; an argument that in our view is only correct when compressing back into global memory (leftmost Figure 1, which we recommend not to do). The Tile-Based micro-benchmarks in [136] are decompressing into global memory – in this chapter we re-do those micro-benchmarks using the alternatives of decompressing into registers and into shared memory - where the query processor directly consumes it (middle of Figure 1: vectorized decompression). Finally, the end-to-end SSB results reported in [136] are 35% slower than running Crystal on uncompressed data; which is explained as something that

should be expected – however this chapter shows that SSB queries can get *faster* thanks to compression. This comes in addition to the compression benefit of being able to store 2-4x more data in GPU memory (and a reduction of the PCIe bottleneck when data is moved into the GPU).

4.4 FastLanes on GPU

In this section, we explain our first implementation of FastLanes on the GPU, which is available in our GitHub repository[2]. Moreover, we perform micro-benchmarks to test three different approaches (global-to-global, global-to-shared and global-to-registers) to determine which approach minimizes latency when decompressing data into memory.

4.4.1 Initial Implementation

FastLanes for CPUs leans into the *Single Instruction Multiple Data* (SIMD) capabilities of CPUs to decode multiple values in one pass. GPUs, however, are based on the SIMT model. To exploit the SIMT parallelism provided by GPUs, we assign a vector of 1024 values to a block. Since we assign one warp per block, every thread in a warp decodes 32 values and thus functions as a so-called *lane* in FastLanes. This means that at least 32 values are decoded in parallel in every step within a warp, when decoded into 32-bit integers. Thanks to use of interleaved bit-packing, this leads to coalesced memory access. Decoding into 1024 32-bits values still fits in GPU registers. We note that GPU systems so far focus on 32-bits data processing, since this is the native data type for GPUs. However, FastLanes has the capability of achieving data-parallelism using scalar instructions, i.e., it can use 32-bits instructions to decode 4x8bit and 2x16bit lanes per GPU thread. This capability can be used in GPUs to decode into thinner data-types than 32-bits, reducing GPU shared memory and register pressure; while at the same time performing 2x or 4x more operations per instruction.

4.4.2 Micro-benchmarks

We use two different GPUs for our benchmarks: the Tesla T4 and Volta V100, from which the exact specifications are shown in Table 4.3. We benchmark both FastLanes Bitpacking (FLS-BP) and FastLanes DELTA (FLS-DELTA) using C++20, nvcc and CUDA 12.3 for the implementation.

Measured execution time. In the micro-benchmarks we measure the execution from the moment the data is loaded into global Memory of the SM. Loading data from and to the CPU is thus not taken into account. Therefore, the PCI/e bandwidth is not a bottleneck in any of the experiments. The measured time is thus from the time the kernel is executed, until the execution is finished and the final results are written back to global memory. To measure the execution time per kernel we make use of the Nvidia Nsight Compute CLI (`ncu`).

Global-to-Global Memory. The global-to-global scenario of this experiment is depicted leftmost in Figure 4.1. A block or vector of compressed data is fetched from *global memory*. It then is decompressed by using bit-unpacking or GPU-FOR decompression, and the

[2] `https://github.com/cwida/FastLanesGPU`

Table 4.1: *global-to-global* decoding 3 to 32 bits (256M values)

GPU	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
T4	7.80 ms	5.89 ms	12.36 ms	6.05 ms
V100	1.60 ms	1.60 ms	1.77 ms	1.64 ms

Table 4.2: *global-to-shared* decoding of 3 to 32 bits (256M values)

GPU	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
T4	10.02 ms	2.42 ms	14.63 ms	4.16 ms
V100	1.22 ms	0.44 ms	1.90 ms	0.63 ms

decompressed data (1GB) is directly written back to *global memory*. We repeat this experiment for every bit-width between 1 and 32. We bypass the L1 cache and make no use of *shared memory*, but store the results directly in global memory. The results in Table 4.1 show that on the Tesla T4, FastLanes outperforms Tile-based for both bit-unpacking (GPU-FOR vs FLS-BP) and DELTA decoding (GPU-DFOR vs FLS-DELTA). On the V100, both FastLanes and Tile-based are memory bandwidth bound, hence they have an *exactly* similar execution time.

Global-to-Shared Memory and `SUM`. We now fetch compressed data from *global memory*, decompress the data to *shared memory*, and directly perform a SUM on the decompressed data (middle of Figure 4.1). This aggregation is necessary to prevent the CUDA compiler from optimizing away all computation. The result of this local aggregation in shared memory is then written again to *global memory*, such that the aggregation value persists between the execution of multiple blocks. This avoids two passes to global memory: first decompressing and writing the data to *global memory* and then, when decompressing is finished and all data is stored, fetching this data again from *global memory* to perform any operations on the decompressed data.
Performing a “simple" aggregation now includes (1) an explicit allocation of shared memory for every block and (2) an aggregation in each thread in the corresponding thread block and a write to global memory for each block that is executed. Explicitly using shared memory incurs overhead, especially for Tile-Based, since it otherwise only uses GPU registers. However, if one stores intermediate results in registers (which is thread-local memory), this intermediate result is not accessible by other threads. The aggregation in Tile-Based ultimately requires 128 threads to each aggregate 4 values, which incurs 128 writes per 512 values to store the temporary result in shared memory. Finally, the final aggregation is written back to global memory.
For FastLanes, global-to-shared works by decoding a vector of 1024 values using one warp, such that each of the 32 threads decodes and directly aggregates 32 values. This results into 32 writes for the intermediate result into shared memory and one write to global memory for each 1024 values. By avoiding to write the large uncompressed (32-bits) results back to global memory, all kernels are now compute-bound. This reveals that FastLanes indeed is computationally faster than Tile-based compression. A somewhat unexpected result on T4 is

Table 4.3: Specifications for Tesla T4 and Tesla V100 GPUs

Model	T4	V100
Memory	16GB	16GB
Memory Bandwidth	320.0 GB/s	900.0 GB/s
L1 size (per SM)	96 KB	128 KB
L2 size	6 MB	6 MB
SM count	40	80
Max. Warps/SM	32	64
Max. Blocks/SM	16	32
Max. Threads/SM	1024	2048
Max. 32-bit reg/SM	65536	65536
Avg. registers/thread	64	32
Max. registers/thread	255	255

Table 4.4: *global-to-shared* memory + `SUM` with varying bit-width for Tile-based (GPU-FOR) and FastLanes (FLS-BP)

	GPU-FOR		FLS-BP	
Bits	T4	V100	T4	V100
1	10.09 ms	1.21 ms	2.34 ms	0.435 ms
3	10.02 ms	1.22 ms	2.42 ms	0.434 ms
4	10.09 ms	1.22 ms	2.45 ms	0.436 ms
8	10.14 ms	1.23 ms	2.60 ms	0.437 ms

that the Tile-Based kernels are in fact slower than in the global-to-global benchmarks for GPU-FOR and similar behavior is observed for GPU-DFOR on the V100 (1.77 vs 1.90 ms), which is depicted in Table 4.2. This increase in compute can be attributed to the fact that including a SUM is computationally intensive, and since GPU-DFOR seems compute bound instead of memory bandwidth bound on both T4 and V100, this leads to deteriorating performance on both GPUs.
In Table 4.4 we confirm that FastLanes is consistently 3-5x faster than Tile-Based in global-to-shared decoding, for various bit-widths. Its kernels are compute-bound, as they only read compressed data, and the execution time increases by adding an aggregation. Increasing the data volume does not affect performance.

Global-to-Register. A third option is to *directly* store the output of the decompression in GPU registers, as opposed to *shared memory*. In CUDA, scalar variables are stored in registers by default by the compiler. Figure 4.4 shows the performance of FLS-BP for all bit-widths (1-32), using all three approaches. The red line models what we think is the computational cost of global-to-register. The solid black line (overall global-to-register time) follows this line for lower bit-widths, but starts to follow a linearly increasing line that we attribute to read cost from global memory. Note that the shared memory in V100 appears faster than on T4 as its line is completely identical on V100 with global-to-register. Global-to-global is at a constant distance from the read-cost, due to its additional cost for

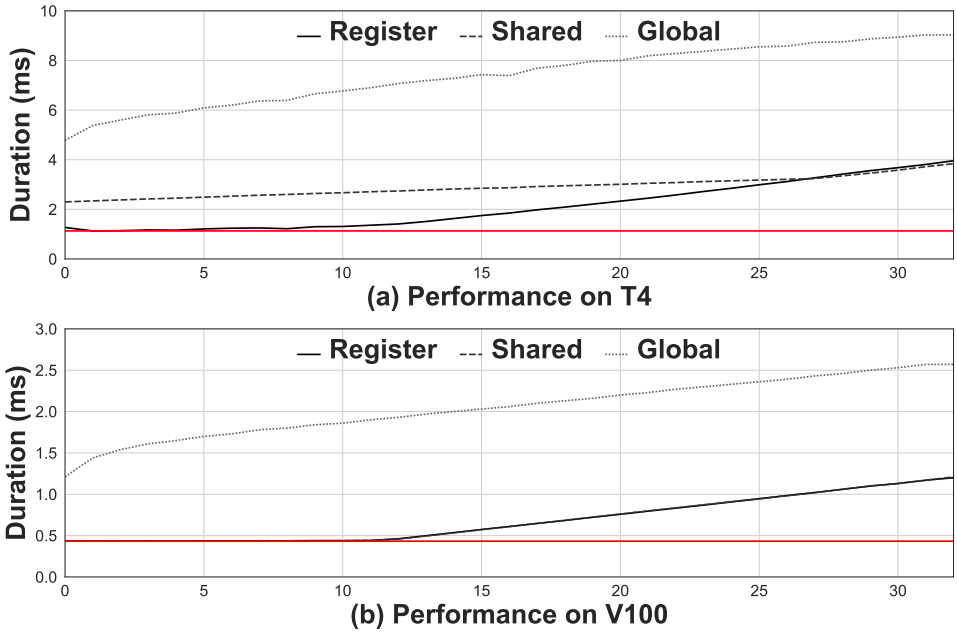


Figure 4.4: Micro-benchmarks for FastLanes unpacking into 32-bits values. X-axis is the bit-width of the compressed column (256M values). The red horizontal line is estimated computational cost of unpacking. The solid black line (global-to-shared) shows the impact of read-bandwidth; the dotted line the impact of write-bandwidth (global-to-global).

writing 1GB of data to global memory, which on V100 corresponds to the read-cost measured at 32-bits.

Measuring Compute. For the next micro-benchmarks, where we aim to measure the "raw" compute, we repeat the procedure described at the global-to-global memory benchmarks with one modification: we write nothing back to global memory. However, if we write nothing back to global memory the compiler optimizes everything away. Therefore, we trick the compiler by implementing an if statement containing a `STORE`, which has an almost 100% probability to evaluate to false. Now, we are able to measure the compute with increased precision. Note that the latency of fetching the compressed data from global memory is still included in the execution time.
The results in Table 4.5 indicate that FLS-BP is around 3-4x faster compared to GPU-FOR, and FLS-DELTA is around 4-5x faster then GPU-DFOR on a T4. On the V100, this difference is around 2x for FLS-BP and GPU-FOR, and 3x for FLS-DELTA and GPU-DFOR (Table 4.6). Also, on T4, FLS-BP is 2-3x faster then FLS-DELTA in terms of compute, and the same phenomenon occurs for GPU-FOR and GPU-DFOR. For V100, the difference for FLS-BP and FLS-DELTA is minimal, indicating that it is close to the minimum compute. GPU-FOR is around 2x as fast compared to GPU-DFOR on V100. Remarkably, in the global-to-global memory benchmarks the execution times of FLS-BP

Table 4.5: Measuring *compute* – thus no writes to global memory. Decoding of 3 to 32 bits (256M values) on Tesla T4. Compute is the "raw" compute of each method, including fetching compressed data from global memory. To-global are the values for T4 reported in Table 4.1, to highlight the difference with- and without writing back to global memory

	GPU-FOR	**FLS-BP**	**GPU-DFOR**	**FLS-DELTA**
compute	3.48 ms	0.98 ms	10.70 ms	2.49 ms
to-global	7.80 ms	5.89 ms	12.36 ms	6.05 ms

Table 4.6: Measuring *compute* – thus no writes to global memory. Decoding of 3 to 32 bits (256M values) on V100 GPU. Compute is the "raw" compute of each method, including fetching compressed data from global memory. To-global are the values for V100 reported in Table 4.1, to highlight the difference with- and without writing back to global memory

	GPU-FOR	**FLS-BP**	**GPU-DFOR**	**FLS-DELTA**
compute	0.86 ms	0.43 ms	1.58 ms	0.45 ms
to-global	1.60 ms	1.60 ms	1.77 ms	1.64 ms

and FLS-DELTA are very similar. An explanation for the smaller difference between global-to-global and solely compute for FLS-DELTA and GPU-DFOR is that the increased compute hides the write latency to global memory.

Investigating Occupancy. To explore whether we can further optimize FLS-BP global-to-shared and FLS-DELTA global-to-shared, we investigate the GPU utilization. We found that both FLS-BP and FLS-DELTA suffer from low occupancies compared to Tile-Based (Table 4.7). In addition, for both FLS-BP and FLS-DELTA the occupancy on V100 is structurally lower.
The lower occupancy on V100 can attributed to the fact that for both T4 and V100 GPUs there are 64k 32-bit registers available per SM, even though V100 provides double the amount of maximum active threads per SM (Table 4.3). As a consequence, software employed on the V100 suffers from high register pressure. Compiling with `ptxas=-v` shows that FLS-BP uses at most 64 registers per thread. This leads to at most $64K/64$ 1024 active threads; which equals the maximum amount of 1024 active threads per SM on T4 (Table 4.3). On V100, this is below the maximum amount of 2048; which already leads to a lower occupancy. This phenomenon is also visible in Table 4.7. However, the occupancy is below 50% in all cases. This means that there are other factors that attribute to the low occupancy. These are (i) there are too little blocks or warps launched per SM and (ii) the required amount of shared memory per block is too high.

Block Size. We benchmark different configurations for FLS-BP *global-to-shared*, to investigate whether increasing block size improves the occupancy reported in Table 4.7. However, Table 4.8 shows that although the execution time increases slightly for a block size of 64, which is probably due to the reduction of scheduling overhead, both the theoretical and achieved occupancy gradually decrease. The decrease of occupancy can be attributed to the fact that we configure around 65kb (as pointed out by `ncu`) of shared memory at our

Table 4.7: *Occupancy* reported by `ncu` of both FastLanes and Tile-Based for bit-unpacking of 3 bits and DELTA decoding. `T` indicates the maximum theoretical occupancy of this configuration, `A` indicates the achieved occupancy during execution.

Method	T4 T	T4 A	V100 T	V100 A
GPU-FOR	100%	94.16%	100%	93.99%
FLS-BP	46.88%	44.67%	34.38%	16.09%
GPU-DFOR	100%	95.94%	100%	96.15%
FLS-DELTA	21.88%	20.75%	17.19%	16.04%

Table 4.8: Micro-benchmarks for different configurations when decompressing 3 bits using FLS-BP. Both timing and occupancy are reported. Occupancy T indicates theoretical occupancy, Occupancy A indicates achieved occupancy. Experiments are done on Tesla T4.

Configuration	Exec. time	Occupancy T	Occupancy A
<32, 32>	2.42 ms	46.88%	44.67%
<64, 32>	2.24 ms	40.81%	43.75%
<128, 32>	2.33 ms	37.50%	34.63%
<256, 32>	3.49 ms	25.00%	24.63%

initial launch configuration for FLS-BP global-to-shared. This implies that it is not possible to achieve a higher occupancy in our current implementation since we are limited by the required amount of shared memory per block. It is therefore more favorable to use the global-to-register approach, where shared memory usage will not become a bottleneck for bit-unpacking.

4.5 FastLanes on Crystal

Our basic implementation of FastLanes on Crystal is referred to as FLS-GPU. In FLS-GPU we integrate FastLanes with Crystal using vectorized decompression (shown in the second scenario of Figure 4.1), where we decode the 1024 values using a block consisting of 32 threads, i.e. a single warp, such that each thread decodes 32 values. This approach is similar FastLanes on CPUs and is thus trivial as a basic implementation. `Crystal` however uses registers to store in-flight data. Relying on registers can be tricky, since the programmer does not have explicit control over the placement of data into registers. To guarantee that data resides in registers `Crystal` processes only 4 values per thread. Each thread in a warp can operate on at least 32 32-bit registers on a V100 GPU, which means that registers can be used for up to $\frac{32}{4} - 1$ 7 columns in `Crystal` without a performance penalty.
When integrating FastLanes in `Crystal`, we choose to follow a similar approach which resembles the global-to-register approach explained in section 4.4. This is more beneficial since (i) profiling indicates that the occupancy is inevitably low due to shared memory usage being a limiting factor for the global-to-shared version of FLS-BP and (ii) the micro-benchmarks in Figure 4.4 show that the global-to-register approach is the most performant for data that is bit-packed in small bit-widths.

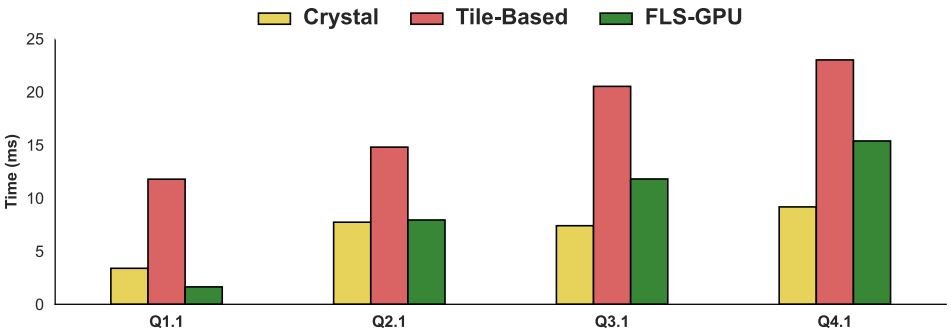


Figure 4.5: End-to-end SSB query execution times (SF10) on Tesla T4. Naive FLS-GPU significantly improves Q1.1 and generally performs better than Tile-Based decompression. Compared to Crystal however, there is still a performance penalty.

End-to-end benchmarks. We now benchmark the integration of FastLanes (de)compression in end-to-end queries, for which we use the Star Schema Benchmark (SSB) [138] queries implemented in Crystal [131]. SSB is a modified form of the `TPC-H` benchmark. Alongside integrating FastLanes in `Crystal`, we compare the benchmark results against Crystal-opt [132] and Tile-based decompression integrated in Crystal as reported in [136]. We only benchmark a single query from each different query family (Q1.*, Q2.*, Q3.* and Q4.* resp.), such that we can compare the performance among different types of queries within SSB. We chose to benchmark a scale-factor of 10 since this will fit as a whole in global memory.
Figure 4.5 shows that for all queries, FLS-GPU performs better compared to Tile-Based. However, except for Q1.1, FLS-GPU performs worse than baseline Crystal. This is because FLS-GPU incurs extra operations for decompression, and its naive approach of unpacking 32 values per thread with a maximum of 1024 values per block leads does not leverage the parallelism provided by GPUs.
Another unexpected result is the difference in behavior of FLS-GPU on the V100 and T4 as observed in figure 4.5 and figure 4.6. In fact, on T4, FLS-GPU performs significantly better than Tile-Based for all queries. On V100, however, Tile-Based outperforms FLS-GPU on Q3.1 and Q4.1. To find an explanation for this behavior we investigate whether register spilling or low occupancy cause this low performance. Specifically occupancy might be problematic, since Table 4.7 already indicated very low occupancy rates for FLS-BP and FLS-DELTA on a V100 GPU – which negatively affects performance.

Register Spilling. FLS-GPU unpacks 1024 values at a time, putting high pressure on registers. Therefore, a possible cause of slowdown for FLS-GPU is register spilling. If the spill is significant and the caches are full, this can cause a high slowdown (explained in section 4.2.1). To determine if spilling occurs at compile time, we compile the queries with the `'-ptxas-options=-v'` flag. We found that no register spilling occurs *at compile time* for both FLS-GPU and Tile-Based. However, for some queries, such as Q3.1 and Q4.1, which includes large hash tables and multiple columns, FLS-GPU assigns up to 226 registers

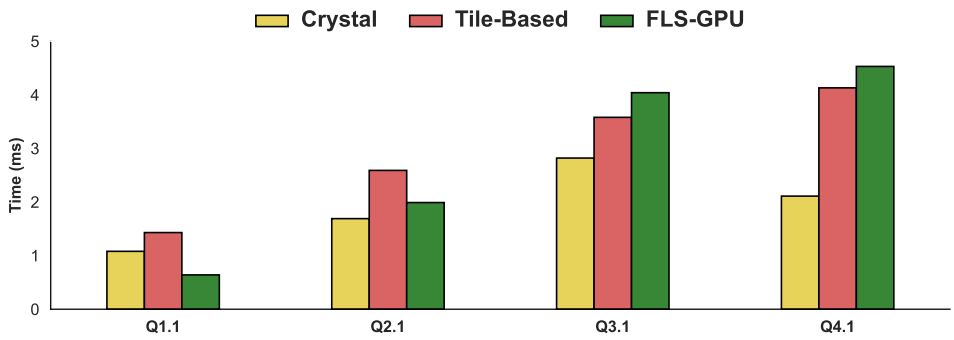


Figure 4.6: End-to-end SSB query execution times (SF10) on NVIDIA V100. Naive FLS-GPU still improves Q1.1 but generally performs worse compared to both Tile-Based decompression and Crystal.

Table 4.9: Occupancy for Q1.1, Q2.1, Q3.1 and Q4.1 using FLS-GPU. Both *theoretical* occupancy (`O-T`) and *achieved* occupancy (`O-A`) are reported for both T4 and V100 GPUs.

Query	O-T T4	O-A T4	O-T V100	O-A V100
Q1.1	50.00%	49.07%	37.50%	36.39%
Q2.1	50.00%	48.54%	25.00%	24.32%
Q3.1	25.00%	24.50%	12.50%	12.11%
Q4.1	25.00%	24.67%	12.50%	12.17%

per thread. This high amount of registers limits the number of threads (and thus warps) we can execute concurrently on a SM, slowing down performance. Due to the high amount of registers per thread, we obtain a low occupancy of 25% and even 12.50% for both queries on T4 and V100 respectively (Table 4.9).

Low Occupancy. Increasing the occupancy will, *in most cases*, lead to better performance. In its current form, FLS-GPU can only reach 50% of occupancy (Table 4.7 and Table 4.9), since it assigns only 32 threads, i.e. one warp, to a single block. Since the block limits on T4 and V100 are 16 and 32 with corresponding warp limits of 32 and 64 respectively, the highest theoretical occupancy we can achieve is 50% – assuming that register and shared memory usage are no limiting factors. On V100, where there are on average only 32 registers per thread available (Table 4.3), we only reach a very low theoretical occupancy of 12.50% for Q3.1 and Q4.1. This low occupancy explains the bad performance of FLS-GPU compared to Tile-Based on V100. To increase both theoretical and achieved occupancy, we consider increasing the thread block size from 32 to 128 or 256 such that we will achieve the maximum amount of blocks or warps that run concurrently on a SM.
However, only increasing the block size is not enough, as shown in Table 4.8. To achieve a better occupancy we therefore need to (i) reduce the amount of shared memory we use per block and/or (ii) decrease the amount of registers used per thread. Since FLS-GPU uses the global-to-register approach, we already minimized the amount of shared memory used for

bit-unpacking. If we want to boil down shared memory usage even further, this would require adapting the SSB queries in `Crystal`, which is out of scope. The second option is to reduce the register usage by unpacking less values per thread. For example, we could unpack 8 values per thread using 128 threads per block, which still leads to a single block processing 1024 values. This form of scheduling almost resembles the Tile-based processing model, which unpacks 4 values per thread for GPU-FOR and GPU-DFOR, using a total of 128 threads per tile (i.e. thread block). Another way to reduce the registers per thread is compiling with the `-maxregcount` flag or using `__launch_bounds()` parameter to limit the amount of registers per thread for all or specific kernels. Forcing a lower amount of registers per thread however leads to register spilling, which becomes quickly very expensive and reduces the overall performance. Therefore, we aim to process less values per thread to reduce register pressure in a more natural way.

4.5.1 FLS-GPU-opt

The results in figure 4.5 and figure 4.6 show that FLS-GPU (green) only improves performance over Crystal (yellow) in Q1.1. The reason why this happens is that the other queries involve more columns and joins which require in-memory hash-tables. As a consequence, the register pressure generated by FLS-GPU becomes too high in these queries. The register pressure in combination with scheduling too few blocks per SM leads to a low occupancy which in this case severely affects performance. Therefore, we started considering methods to reduce the register pressure and increasing the thread block size, moving to FLS-GPU-opt; depicted in the rightmost scenario of Figure 1. The optimized version of FastLanes on Crystal, FLS-GPU-opt, addresses shortcomings of FLS-GPU while leveraging GPU parallelism. This includes releasing pressure on registers by processing *mini-vectors* and using *compressed execution*. In addition, FLS-GPU-opt achieves a better compression ratio by using RLE for suitable SSB columns, such as `lo_orderdate`. Each of the optimizations is briefly explained below.

Processing *mini-vectors*. To release pressure of registers and shared memory we partition a vector of 1024 values into *mini-vectors* of 256 values. This means that each thread in a warp now processes 8 values at-a-time, thus using 8 32-bit registers per column, a 4x reduction of register pressure. Technically, this means that bit-unpacking logic is split over 4 FastLanes unpack methods; each delivering 256 values. For bit-widths that are not multiples of 4-bits this leads to some additional work if the unpacking does not start aligned on a 32-bits values, but the extra effort is low.

Compressed Execution. While processing queries in Crystal, all SSB columns are handled in-flight as 32-bit integer values. However, some columns of the SSB benchmark can be encoded in significant smaller data types. Using smaller data types is beneficial to reduce both the memory footprint and memory bandwidth. This however is not natural to GPUs, since each register in a GPU spans one *word*, and each word consists of 32 bits. It is thus convenient to decompress values into 32-bit integers to align with the word size. However, decompressing values into 32 bits is inefficient if significantly less bits are needed. For example, let's assume that we are able to represent values of a column in 8-bits. We then can *partially* decompress the bit-packed values into four 8-bit lanes in one 32-bit register, instead of decompressing a single 8-bit value into 32 bits. This also allows us to directly operate on

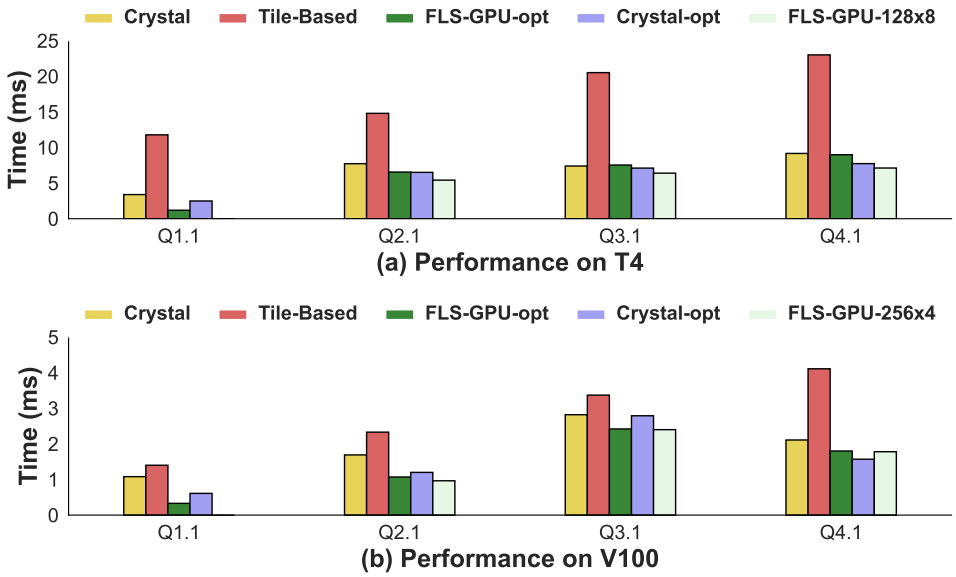


Figure 4.7: End-to-end SSB query execution times (SF10). Naive FLS-GPU significantly improves Q1.1; but it generates too much register pressure in the other queries, which involve more columns and are hash-probe rather than scan-bound. By reducing the decompression granularity with mini-vectors, using more threads per block and simulating RLE, FLS-GPU-opt can match its performance nevertheless.

these values at the same time, *within* a single thread. Thus, we use some SIMD parallelism with the GPU SIMT execution model. As a result, we are able to (i) fit more data into registers which avoids spilling to L1 cache (ii) enhance more data-parallelism by performing multiple operations at the same time and (iii) reduce bandwidth by a factor 4.

Predicate Pushdown. In real-world systems, columns that do not benefit from bit-packing will not be compressed. Therefore, we leave the column `extended_price` uncompressed, and we use the <PredLoad> predicate-pushdown optimization proposed by Crystal-opt [132] to reduce bandwidth. Crystal-opt [132] showed that Crystal loads unnecessary data from *global memory* and this affects performance.
Note that when using scans on compressed columns, such predicate-pushdown is impossible (or, it would require random access to compressed data). Therefore, *all* compressed data needs to be decompressed at least to *shared memory*, incurring global memory bandwidth. Only if an entire vector or tile would have zero selected tuples, this step could be skipped. This is similar to another optimization of Crystal-opt, which terminates a thread and eventually a warp early if no tuple is selected. However, this is mostly beneficial for highly selective queries, i.e. when a chunk of values does not satisfy any of the selection flags. None of the SSb queries Q1.1, Q2.1, Q3.1 and Q4.1 benefit from this.

Simulating RLE. In the current port of FastLanes to GPU, we do not support RLE yet. The SSB LINEORDER table is clustered on order, which means that it is quite

Table 4.10: Occupancy for Q2.1, Q3.1 and Q4.1 using FLS-GPU-128x8 for T4 and FLS-GPU-256x4 for V100. Both *theoretical* occupancy (`O-T`) and *achieved* occupancy (`O-A`) are reported. Q1.1 is not included, since this query already outperformed Crystal, Tile-Based and Crystal-opt significantly.

Query	O-T T4	O-A T4	O-T V100	O-A V100
Q2.1	100.00%	93.23%	100%	92.25%
Q3.1	87.50%	81.49%	75.00%	68.90%
Q4.1	87.50%	80.92%	75.00%	67.75%

RLE-compressible, as all columns that contain order information, rather than lineitem information, repeat on average four times. In the SSB queries tested here, this concerns the `lo_orderdate` and `lo_custkey columns`. Lacking RLE, the compression ratio FLS-GPU achieve is diminished to about 1.5x. In real-life datasets, such as public BI [130], FastLanes can achieve a compression ratio of 8X. To mitigate the bad compression ratio, partially caused by our lack of an RLE implementation, we decided in a separate experiment to sort LINEORDER on columns `lo_orderdate` and `lo_custkey`. This allows to store `_orderdate` in 8 bits instead of 16, and `l_custkey` in 8 bits instead of 20 (real RLE would reduce this even to 6 8 bits).

Larger Block Size. Table 4.9 indicates that the occupancy of FLS-GPU on both T4 and V100 is low for all queries, but particularly for Q3.1 and Q4.1. To improve occupancy by lowering register pressure, we now move to a 8-values-per-thread model, which we call *mini-vectors*, as described above. Instead of only launching thread blocks consisting of 32 threads, we now increase the size to 128 threads per block for T4 to still decode 1024 values per block (FLS-GPU-128x8). This allows to execute more warps concurrently while reducing register pressure. For V100 however, there are even less registers available per thread, leading to a higher register pressure. Therefore, we choose to use a 4-values-per-thread model, using 256 threads per block to decode 1024 values per block (FLS-GPU-256x4). For these configurations, we also remove the predicate pushdown optimization and instead compress all columns.

4.5.2 Discussion

SSB Q1.1 is a simple scan with filter and aggregation. The roof-line analysis in [132] already showed that this query is the most scan-bound – and thus stands to profit most from compressed storage. The fact that Tile-Based compression is not able to improve performance of this query is a missed opportunity, but explainable from the fact that its encoding format lacks data-parallelism needed by GPUs. The interleaving of values in a vector employed by FastLanes however allows Q1.1 to execute 2-3x faster, illustrated also in Table 4.11. For Q1.1 the predicate-pushdown on `extended_price` provided FLS-GPU-opt most of the additional gains over FLS-GPU. For the other queries, where FLS-GPU suffers from too high register pressure, the FLS-GPU-opt benefits most from using mini-vectors. Notably, we did not manage (yet) to make compression faster using the idea of compressed execution, i.e. using data types smaller than 32-bits. The reason for this lack of success is as of yet unclear, and there are still techniques we could try. We further think that more complex encoding schemes, like RLE and DICT, which we so far have not implemented,

Table 4.11: On the scan-bound Q1.1 that stands to profit most from compressed scans, FLS-GPU shows strong performance, which is significantly enhanced in FLS-GPU-opt.

Scheme	SF1-T4	SF10-T4	SF1-V100	SF10-V100
Crystal	0.35	3.39	0.115	1.080
Crystal-opt	0.26	2.49	0.070	0.608
FLS-GPU	0.21	1.92	0.087	0.642
FLS-GPU-opt	0.139	1.19	0.057	0.335

could benefit from GPUs.

For Q3.1 we managed to increase the performance further by sorting LINEORDER on `lo_orderdate` and `lo_custkey` to achieve a better compression ratio. The query performance from FLS-GPU-opt goes from 8.17 ms to 7.54 ms on T4, and from 2.78 to 1.33 on V100. Specifically for the V100 GPU the performance increase is a factor of 2, which is significant. For Q4.1, the improved compression ratio provided by sorting did not have a significant impact. We do intend to re-benchmark FLS-GPU when RLE support is ready and this sorting is no longer required.

Lastly, aiming to increase occupancy, we scheduled larger thread blocks. We found that for the T4 high occupancy is achieved for thread blocks of 128 threads, that process 8-values-per-thread (Table 4.10). For Q2.1 this improved the execution time from 6.55 to 5.42ms, for Q3.1 from 7.54 to 6.40ms and for Q4.1 from 8.99 to 7.12 ms. For V100, we still suffer from severe register pressure, and therefore were not able increase the occupancy with the 128x8 format. Instead, we tried 256x4 to process even less values per thread. However, register pressure remained problematic for the occupancy – only little performance improvement is observed. We note though, that Q3.1 and Q4.1 which involve more columns than the other two queries, also cause lower occupancy for Crystal itself.

4.6 Conclusions and Future work

In this chapter, we tested the data-parallel layout of FastLanes on GPUs. Both our micro-benchmarks as well as end-to-end SSB query results show encouraging results. The micro-benchmarks in section 4.4 showed that FLS-GPU outperforms Tile-Based decoding by a factor of 3-4x for bit-unpacking against GPU-FOR and FLS-DELTA decoding against GPU-DFOR. We also show in contrast to Tile-Based (which causes a 35% slowdown of end-to-end queries), that the overhead incurred by FastLanes decompression in Crystal is offset by reduced memory bandwidth; an important bottleneck for data processing on GPUs.

We also found drawbacks of the original 1024 tuples at-a-time decoding granularity of FastLanes: this forced it to use at least 32 registers per thread - which are not always available, or to store 1024 values on shared memory for each block. This proved to become a bottleneck on more complex queries with a multiple columns to process. We addressed this issue using the idea of *mini-vectors* and larger thread blocks, which perform FastLanes decompression in four steps of each 256 or 128 values to reduce pressure on GPU registers and shared memory, as well as the idea of decoding into thin data-types (8- and 16-bits). We however experienced that register pressure on the V100 remains a challenge, and were not able to significantly improve its execution time using a 256x4 configuration.

FastLanes on GPUs is still in an early stage of development, and its more complex encoding schemes (e.g. RLE) were not available yet in CUDA during these experiments. This causes SSB experiments to experience lower compression ratio's than are normally possible with FastLanes. The experiments with an artificially enhanced compression ratio (by sorting the LINEORDER table) already show that end-to-end query performance will further improve once the FastLanes GPU implementation reaches greater maturity.

4.6.1 Future work

Improving Mini-Vectors. In FastLanes, we bit-pack 1024 tuples using the interleaved layout, which has as advantage that all 32 threads do the same decoding work (no divergence) and have coalesced memory access. To support access using mini-vectors (we experimented with 8 resp. 4 values per thread), for bit-widths other than multiples of 4 resp. 8, memory access is not 32-bits aligned. In our experiments we used our original decoding methods and mitigated by rounding up bit-widths to the closest higher multiple of 4 resp. 8, hurting compression ratio and thus performance. This rounding up can be avoided by a proper implementation for all bit-widths at some additional computational cost during decoding.

Reducing Mini-Vectors. The observed strong effects of register pressure make us consider even smaller mini-vectors, and even the extreme approach of threads doing single-tuple access. The trade-off here is increased computational overhead in decoding calls, for invoking the decoding action appropriate for each mini-vector, as well as for interpreting cascaded encodings. The decoding interpretation cost would in the extreme case be incurred for each tuple. We note that the variability of data in-the-wild requires an interpreted approach for decoding, as parameters and encodings used will vary between different parts of a column.

Adding DELTA and RLE In this chapter, we mainly focused on bit-packing, as FastLanes on GPU is still in a very conceptual phase of development, and the full set of basic encodings had not yet been ported to CUDA (specifically DELTA/RLE and DICT). This is an opportunity to further improve our results, since quite a few columns from the SSB benchmark can be better compressed by RLE. We think that using RLE we can further speed up SSB queries as the overall compression ratio would increase.

Compressed Execution. We observed that the performance bottleneck of SSB queries Q2.*, Q3.*, and Q4.* are join probes. Therefore, memory latency is a significant cost, caused by the random and non-coalesced nature of hash-lookups; which may only be alleviated by caching (mainly in the L2 cache). While this problem appears to be unrelated to our main topic of accelerating compressed scans from a novel big data format, we do think that the idea of compressed execution (decompressing to thinner types) could allow to build smaller hash tables (which are faster hash tables thanks to improved caching locality). Further, a GPU data processing engine could potentially even squeeze in-flight data, by storing multiple thin (e.g., 8- or 16-bit) values in a single 32-bit value, to reduce register or shared memory pressure; thereby enabling higher kernel performance.

5

Rethinking Light-weight Encodings for GPUs

This chapter introduces G-ALP, which optimizes the GPU implementation of ALP, the state-of-the-art encoding scheme for floating-point data on CPUs, based on two core ideas. First, all parts of the decoding process must be fully data-parallelized, regardless of how insignificant they may be on CPUs. In the case of ALP, we fully data-parallelize exception patching, which is applied to only 1% of the data. While this step is negligible on CPUs, it becomes the main bottleneck on GPUs. Second, the decoding API must be as minimal as possible, delivering one value at a time to absolutely minimize shared memory (`sm`-memory) pressure, a highly scarce resource on GPUs, for users of G-ALP. We consider these two optimizations as guidelines for future GPU optimizations of lightweight encodings and a significant step toward extending the FastLanes file format, the next generation of file formats, to GPUs. Furthermore, we extensively optimized G-ALP through a series of microbenchmarks and evaluated its performance on an NVIDIA V100 GPU, demonstrating superior performance compared to NVIDIA nvCOMP in both decoding and filtering queries, especially under high local memory pressure, simulated by filtering over many columns.

5.1 Introduction

FastLanes is a project initiated at CWI, designed as a foundation for next-generation big data formats. With release v0.1 [139], FastLanes provides an open-source, dependency-free file format, implemented in C++. FastLanes is fully data-parallelized by utilizing the novel 1024-bit interleaved and Unified Transposed Layout [140], enabling fully data-parallel decoding even with scalar code on the CPU. It significantly outperforms the state-of-the-art, achieving an 80× speedup on the M1 processor compared to Parquet while also achieving a 40% better compression ratio.

Furthermore, we addressed the need for a data-parallel encoding for floating-point data by incorporating our novel decoding scheme, **ALP** [141]. ALP encodes floating-point data by mapping it to the integer domain while storing a small amount of metadata to convert floating-point values to integers. This conversion consists of two multiplications and one cast operation, which can be fully data-parallelized. The resulting integers are then further

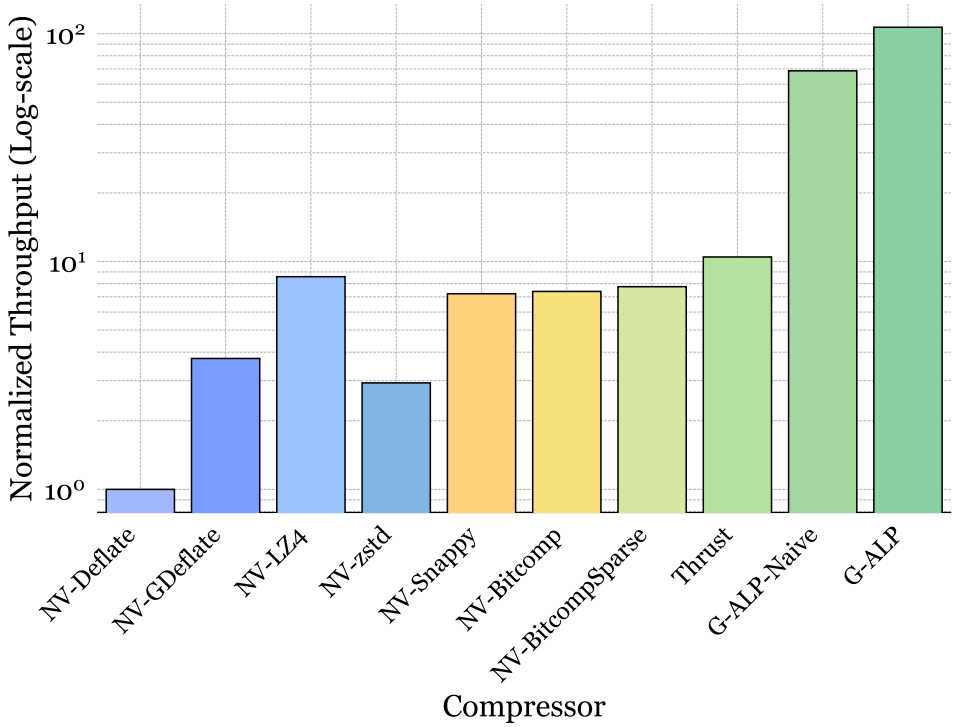


Figure 5.1: Throughput of various encoders on a filter query on floating-point columns from the real-world dataset `PUBLIC_BI`, measured in GB/s. Higher is better. The y-axis uses a logarithmic scale. All compressors implemented by NVIDIA are prefixed with `NV`. G-ALP achieves a throughput more than 100× higher than the baseline `NV-Deflate`. Additionally, the highly parallel compressed query execution of G-ALP enables a 14.8× higher throughput compared to NVIDIA's `Thrust` library, which does not use compression.

compressed using the FastLanes Frame of Reference (FFOR). We observed that approximately one percent of double-precision values cannot be mapped to integers. Therefore, we separate these values, referred to as exceptions, from the main data and encode them separately using a patching mechanism [34]. This mechanism reinserts exceptions during decoding with negligible overhead in terms of both compression ratio and decoding speed, as exceptions occur very rarely. Combining ALP with FFOR and Exception patching uses the *Expression Encoding* feature of Fastlanes, in which multiple light-weight compression methods can be cascaded.

Designing a new analytical file format like FastLanes, should take into account AI workloads, and therefore **GPU** based decoding and, to a lesser extent, encoding. Processing highly compressed data on the GPU is particularly attractive, as GPUs typically have smaller RAM ("global memory") than the host CPU, meaning that storing compressed data alleviates a capacity bottleneck. Furthermore, data is transferred to the GPU over the PCIe bus, so reducing the amount of data moved through compression also helps mitigate this bottleneck. The experiences in this chapter with adapting ALP to GPUs in G-ALP illustrate that GPUs can benefit strongly from small changes in compressed data-layouts. Therefore

we think that there are broader lessons to be learned from this chapter, and we intend to apply these in the next versions of FastLanes, also for other compression methods.
In our initial work on extending FastLanes to the GPU [142], we demonstrated that data-parallel encodings are key to utilizing the massive parallelism of the GPU. However, we observed that our API, optimized for the vectorized execution model—the most widely adopted execution model on the CPU—forces GPU kernels to materialize 32 values per thread per column, which can easily become a bottleneck, as the amount of nearby high throughput memory per thread (registers/shared memory/L1 cache) is significantly more limited on the GPU compared to the CPU. Therefore, we adjusted the FastLanes API to enable even more fine-grained decoding, allowing 16, 8, and 4 values per thread. We observed that this modification was crucial for achieving high occupancy on SSB benchmarks running on a state-of-the-art academic database, Crystal [31].
In this chapter, we introduce **G-ALP**, a GPU-friendly version of ALP with two optimizations. First, we propose a novel data-parallelized layout for storing exceptions, allowing the GPU to reinsert exceptions entirely in parallel. Second, we provide a flexible API for delivering decoded values, ranging from one value at a time per thread—offering the lowest possible number of materialized values to ensure minimal local memory usage for any library using FastLanes—to 32 values at a time per thread, providing a more versatile API with different granularities.

Contributions. Our main contributions are:

- A novel data layout for storing exceptions, fully data-parallel, enabling GPU threads to reinsert exceptions in parallel.
- The design and implementation of G-ALP, a GPU-friendly version of ALP with a data-parallel layout for exceptions and a novel API, enabling one-value-per-thread processing to minimize pressure on users of G-ALP.
- Open-sourcing the implementation of G-ALP[1].
- An extensive set of microbenchmarks used to fully optimize G-ALP.
- An evaluation against nvCOMP, the state-of-the-art compression framework developed by NVIDIA, demonstrating the superior performance of G-ALP in both decoding and aggregation queries.

Outline. We begin by explaining key aspects of GPUs in Section 5.2. Next, we present the design of G-ALP in Section 5.3, followed by our evaluation results in Section 5.4. We then discuss related work, with an emphasis on nvCOMP, in Section 5.5. Finally, we conclude our work in Section 5.6 and outline our vision in the future work section, Section 5.7.

5.2 GPU

In this section, first the general hardware structure of NVIDIA GPUs is described. The section continues with an explanation of how instructions are issued and executed. The

[1] https://github.com/cwida/FastLanesGpu-Damon2025

section concludes with how *instruction-level parallelism* (ILP) can be used to reduce the impact of instruction latencies on performance.

SIMT. An NVIDIA GPU consists of a number of *streaming multiprocessors* (SM). Each SM consists of *warps*. Warps consist of 32 *threads*. All threads in the same warp execute the same instruction; NVIDIA calls this the *single instruction, multiple threads* (SIMT) model. *CUDA*, NVIDIA's GPU programming language, enables developers to program the GPU as if they were programming a single, independent thread. However, because all threads within a warp execute the same instruction, it is more clear to think of instructions executed by the GPU as vector instructions [143].

Instruction pipelines. A SM contains a set of heterogeneous instruction pipelines. Each pipeline only executes certain classes of instructions, such as memory instructions or floating point arithmetic. Some classes of instructions can have multiple pipelines [143]. Warps themselves do not execute instruction, they issue these to the pipelines, which are shared by multiple warps. This sharing of pipelines is somewhat comparable to hyperthreading on CPUs. The SM's warps are distributed among multiple *instruction issuers*, each of which control access to a set of pipelines. Each clock cycle, the instruction issuer picks a warp to issue an instruction [144]. When the warp is not able to issue an instruction, the warp is considered *stalled*. A warp might not be able to issue an instruction due to *data hazards* or *structural hazards*, then the warp needs to wait for the result of a previously issued instruction to complete. When a warp stalls, the instruction issuer will pick another, non-stalled warp to issue an instruction [144–146].

Occupancy. SMs contain a fixed amount of resources that are shared among all warps. A kernel might be programmed in such a way that the SM needs to allocate a large amount of resources per warp. In that case the SM can disable some warps, lowering the *occupancy*, the ratio of *active* warps. If the number of active warps is relatively low, there is a smaller chance that the instruction issuer can find a non-stalled warp to issue an instruction and saturate the pipelines. This can slow down the execution of kernels.

Hiding latency. Reading memory from the GPU's RAM has high latency, in the order of hundreds of cycles [144]. GPUs can bypass this latency by switching warps, executing instructions from other warps while some of the warps are waiting for the results of their memory access, this is called *latency-hiding* [147]. In some situations, the latency of a memory access can be completely hidden, if there are enough other warps that are able to issue instructions.

Instruction-level parallelism. Another way of hiding latency is by enabling warps to issue instructions more often. By increasing ILP, a kernel's instructions can contain less data hazards and control hazards. Then, warps do not have to stall as often due to these hazards. Because warps do not stall as often, the instruction issuer is more likely to be able to pick an active, non-stalled warp, and saturate the instruction pipelines. ILP can be increased by replacing branches with branchless code, by using different algorithms, or by processing multiple values in parallel. ILP only helps when latency is a bottleneck, as when arithmetic throughput or memory bandwidth is the bottleneck, the instruction pipelines are already saturated [148].

5.3 G-ALP

G-ALP is a floating-point data compression scheme designed to optimize FastLanes ALP (referred to as CPU-ALP) for GPUs, built upon two core ideas:

All parts of decoding on the GPU must be fully data-parallelized, even negligible ones. For example, exception patching, which accounts for only one percent of the data, must also be data-parallelized because GPUs perform poorly on any sequential workload, even at such a small scale. In contrast, CPUs are designed to handle sequential workloads efficiently, which is precisely what CPU-ALP is designed for.

The decoding API should decode one value per thread. By decoding only a single value per thread for each kernel call, we minimize additional pressure on the local memory of libraries using the FastLanes reader to an absolute minimum.

Overview. The encoding process of G-ALP is similar to CPU-ALP, with one key difference: a data-parallel exception layout, which is explained later. In G-ALP, each set of 1024 floating-point values is considered a single encoding/decoding unit. During encoding, these 1024 floating-point values are mapped to integers, which are further compressed using `FFOR`, along with metadata specifying how to cast these integers back to doubles, which is later used during decoding. The decoding process follows FastLanes' 1024-bit ISA [140], where each lane corresponds to a separate thread. Conceptually, the decoding process on the GPU can be visualized as 32 threads, each decoding one value at a time for 32 iterations, resulting in a total of 32×32 1024 decoded values.

Data-Parallel Exception Layout. G-ALP stores exceptions in a fully data-parallel layout. This is implemented as an additional step at the end of ALP encoding, where exceptions are reordered into a data-parallel format.
The key idea behind this new layout is to provide each GPU thread, responsible for decoding a specific lane in the CPU-ALP data-parallel layout (e.g., thread 0 handling values at positions 0, 32, 960, 992), with direct access to all exceptions occurring in its lane in a single location. This eliminates the need for each thread to traverse the entire exception list to locate its exceptions, enabling fully parallel exception handling.
For each thread, after decoding a value, the next exception position is checked. If this position corresponds to the current value being decoded, the thread returns the exception; otherwise, it returns the decoded value.
For this data-parallel layout, we first store all exceptions for thread 0, corresponding to lane 0, which consists of positions 0, 32, 64, ..., 992 sequentially. Additionally, we store 16-bit metadata consisting of two parameters:

- **Offset** – indicating where the exceptions for lane 0 start.
- **Count** – specifying how many exceptions this lane has.

The offset and count are essential to provide each thread with direct access to its own exceptions without further computation. The offset can be as large as 1024, requiring 10 bits in the worst-case scenario when all values are exceptions. Each lane can have a maximum of 32 exceptions, requiring only 5 bits to store the count. We use 16 bits to efficiently pack these two parameters together.

This process is repeated 31 more times for threads 1 to 31. Figure 5.2 illustrates this layout by comparing the exception layouts of CPU-ALP and GPU-ALP, highlighting how this design enables more efficient decoding on GPUs.

Compression Overhead. G-ALP introduces a compression overhead compared to ALP, arising from the additional metadata required to make exception patching fully data-parallel. This overhead is fixed at 16×32 512 bits per vector (1024 values), which translates to 0.5 bits per value. Considering that ALP encodes double-precision data using 21 bits, the additional 0.5-bit overhead is negligible.

Decoding. Each thread is responsible for decoding values stored in its corresponding lane of the FastLanes layout, ranging from 0 to 31. The decoding process for a value at position X in the lane consists of the following steps:

1. Apply the frame-of-reference method to decode the integer at position X. This involves generating the appropriate bitmask to extract the relevant bits, followed by a right shift. If the bit-packed value spans two words, an `if` condition fetches the next word and combines the bits.
2. Cast the decoded value to floating-point using a cast instruction.
3. Check whether the current value is an exception. If not, deliver the decoded floating-point; otherwise, return the exception. To determine if a value is an exception, compare the next exception position in the list with the position of the current value in the lane. If they match, the value is an exception, and we move to the next exception.
4. Prefetch the next exception if the current value is an exception to ensure the data is available for the next iteration.

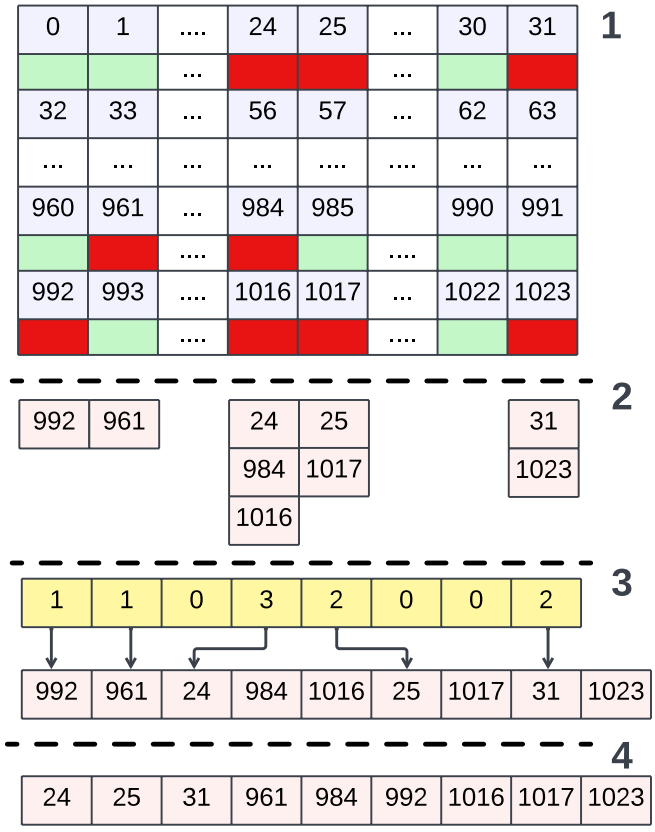


Figure 5.2: Example of Data-Parallel Layout for Patching. 1) A vector of 1024 values consists of two components for each value: the top box represents the position of the value within a vector, ranging from 0 to 1023, while the bottom box is color-coded (red and green) to indicate whether the value is an exception, with red denoting an exception. 2) The second part illustrates exceptions per lane, where each lane contains its own subset of exceptions. 3) The third part shows the actual storage format for exceptions. Exceptions are stored based on lane number, starting with lane 0, followed by lane 1, and so on. Yellow boxes represent metadata consisting of offsets (shown by arrows) that indicate the starting position of exceptions for each lane, as well as the number of exceptions denoted in the box. This structure allows each thread to efficiently access its corresponding exception list. 4) Finally, for comparison, we present the CPU layout of the same exception list, highlighting the structural differences between the two layouts.

5.4 Evaluation

We conducted two sets of experiments: the first set consists of microbenchmarks to evaluate the effects of possible design choices for optimizing ALP on the GPU, and the second set benchmarks G-ALP against other compression schemes used for GPUs across three important categories: scan throughput, compression ratio, and filter throughput.

Setup. All experiments were conducted on an AWS EC2 instance, `p3.2xlarge`, equipped with an NVIDIA V100 16GB GPU, featuring compute capability 7.0 and based on the Volta architecture, a data center-grade GPU. The code was compiled using NVCC 12.8 and `g++-12` as the host compiler. The version of nvCOMP used was 4.2.11. An older version of `g++` was chosen due to nvCOMP's lack of support for newer `g++` versions.

Data. To ensure a fair comparison, we selected double-precision columns from the Public BI dataset [41], as there was no single-precision data type available, and cast them to single-precision floats. We find `PUBLIC_BI` highly relevant, as it was also used in the original design of ALP. Additionally, floating-point data columns that would be better suited for compression with run-length encoding or ALP_{rd} were excluded from the evaluation.

Measurements. To measure throughput, we use two different approaches: Nsight Compute Command Line Profiler 2025.1.1.0 for microbenchmarks and CUDA events for end-to-end queries (Figure 5.1). CUDA events are the recommended method for measuring multi-kernel end-to-end execution time [149] and are also used by NVIDIA to benchmark nvCOMP [150].

Implementation. The experiments and implementation of G-ALP are open-sourced in our repository[2], which includes a FastLanes-compliant reader capable of decoding all FastLanes encodings on GPUs. The nvCOMP code is not open source; however, we used its header files and binaries, which are freely available from NVIDIA's website[3].

5.4.1 Micro Benchmarks

To measure the impact of our new one-value-at-a-time API, we benchmarked two versions of G-ALP: the optimized version versus the naive version, each tested with two different APIs—one-value-at-a-time and 32-values-at-a-time. The benchmark consists of a simple filter query while increasing the number of columns from 1 to 10, thereby increasing local memory pressure as more data needs to be materialized. The results are shown in Figure 5.4. As seen in the figure, our new API maintains throughput close to that of a single-column scan, even as the number of columns increases.
Furthermore, we examined achieved occupancy, as shown in Figure 5.5. When scanning more than six columns, occupancy starts to decline, indicating that the compiler can no longer reduce register usage and instead spills registers to higher-level caches. For a high number of columns, the kernel's performance depends on how efficiently the algorithm utilizes the remaining active warps, relying on instruction-level parallelism rather than occupancy to hide read latencies.

[2] `https://github.com/cwida/FastLanesGpu-Damon2025`
[3] `https://developer.nvidia.com/nvcomp`

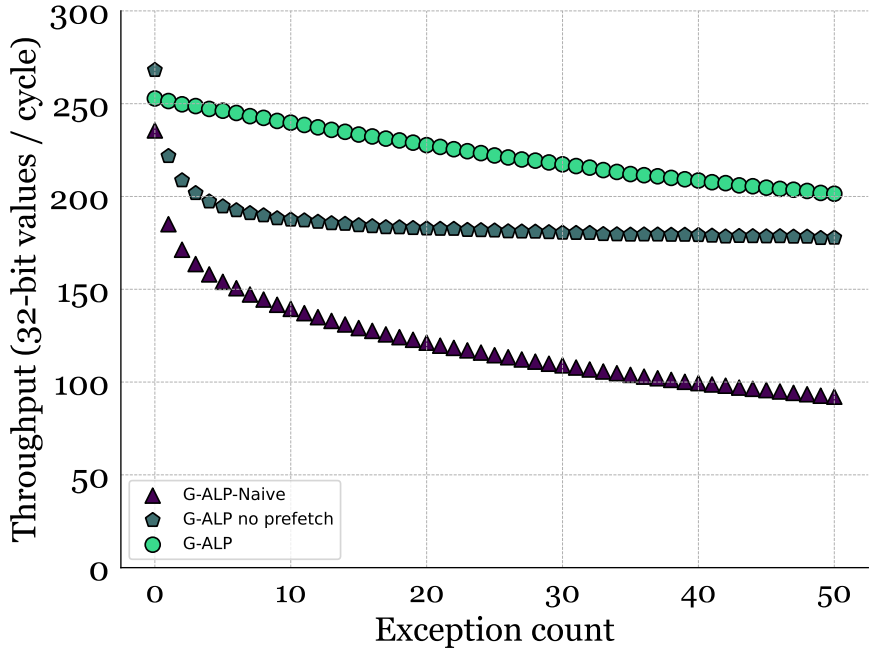


Figure 5.3: Throughput of different implementations of G-ALP, reported in values per cycle, for varying numbers of exceptions. Regardless of the exception count, G-ALP outperforms other implementations due to its data-parallel exception patching and efficient prefetching of exceptions.

5.4.2 End-to-End Benchmarks

To understand how different compressors perform in comparison to each other, we measure their throughput under two queries:

1. **Full decompression** – measuring the absolute decoding time, where data is fully written to global memory, simulating cases where G-ALP is forced to decode data completely.

2. **Filter** – measuring the end-to-end time between decompressing data, and evaluating a query on the decompressed data. For Thrust, GALP Naive, and GALP no separate decompression kernel is required, as thrust does not use compression, and GALP Naive and GALP can load compressed data directly. This simulates the performance of G-ALP in a tile-based execution model, where our API can be used to process data immediately after decoding instead of writing it to global memory. The filter evaluates whether a certain value occurs in a column. A variation of the traditional filter is performed, where not the row numbers are returned but simply a boolean answer on whether the value occurs in the column. We choose this variation as it requires no synchronization between threads, and also requires little write bandwidth. This in turn allows us to isolate the performance of how fast kernels can load data.

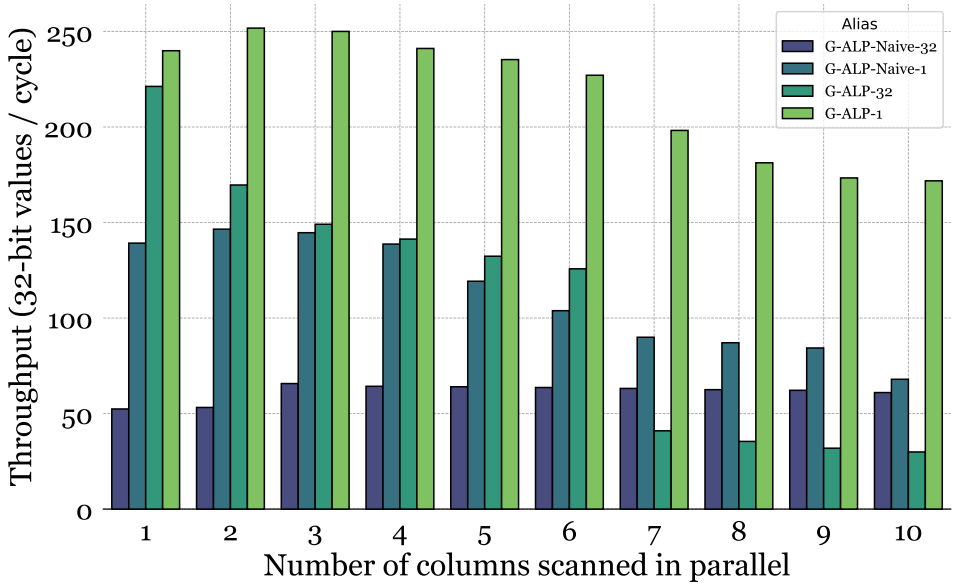


Figure 5.4: Throughput of G-ALP using different APIs with varying numbers of columns (1 to 10). G-ALP with a one-value-at-a-time approach is significantly faster than other implementations, as it requires fewer registers, allowing the kernel to achieve higher occupancy when the number of columns scanned in parallel increases.

We include naive G-ALP, where we simply map the FastLanes 1024 ISA to CUDA, to measure the extent to which our two optimizations: 1) data-parallel exception handling, and 2) one-value-at-a-time decoding API, accelerate the naive implementation of G-ALP. Additionally, we compare against the encodings supported by the nvCOMP framework as the current state-of-the-art in practice and use Thrust as a baseline to evaluate the performance of G-ALP against Thrust when there is no compression, highlighting the impact of compression on GPUs. The results are shown in Table 5.1. As can be seen, G-ALP outperforms all competitors by a significant margin.

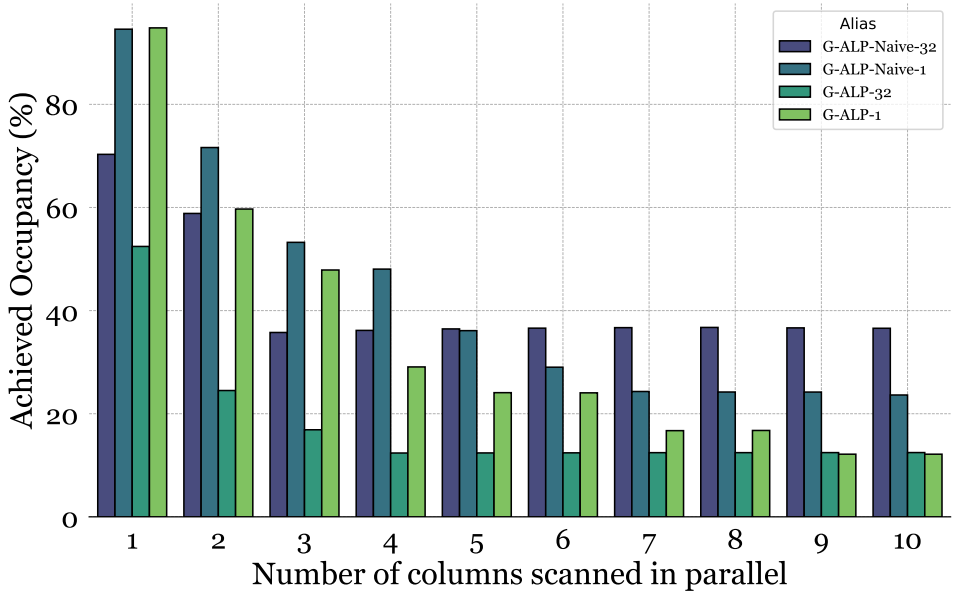


Figure 5.5: Achieved occupancy for the same benchmark as Figure 5.4. When scanning more than six columns, occupancy starts to decline, indicating that the compiler can no longer reduce register usage and instead spills registers to higher-level caches. For a high number of columns, the kernel's performance depends on how efficiently the algorithm utilizes the remaining active warps, relying on instruction-level parallelism rather than occupancy to hide read latencies.

Table 5.1: Throughput of different compressors for two queries: 1) Full decompression and 2) Filter. G-ALP outperforms NVIDIA compressors, achieving an average speedup of 16× for filter and 10× for full decompression. G-ALP is faster in filter than in full decompression due to its tile-based execution model, where data is processed immediately after decoding without being written to GPU memory. Interestingly, G-ALP is also 15× faster than Thrust, which has no compression and therefore needs to load more data from GPU memory, highlighting the win-win situation of fully data-parallel, one-value-at-a-time encodings on GPUs. Furthermore, G-ALP is on average 2× faster than naive G-ALP, demonstrating the potential for further optimizing lightweight encodings on GPUs.

Compressor	**Compression** *Ratio*	**Decompression** *Throughput (GB/s)*	**Filter** *Throughput (GB/s)*
G-ALP	5.68	399.92	1062.20
Naive G-ALP	6.31	332.05	608.98
NV-Bitcomp	4.29	52.22	49.05
NV-BitcompSparse	4.36	57.12	49.12
NV-Snappy	4.71	51.20	47.73
NV-GDeflate	3.89	27.19	24.26
NV-Deflate	3.94	6.85	6.86
NV-LZ4	4.86	60.93	56.61
NV-zstd	7.54	20.17	20.02
NV Average	4.80	39.38	40.67
Thrust [151]	1.00	n/a	71.74

5.5 Related Work

We consider the NVIDIA nvCOMP compression framework [152–155] to be the most relevant work related to G-ALP, as it is widely used in practice, with a total of 608,509 downloads as of the time of writing this chapter [156]. nvCOMP supports three data types: integers, strings, and floats (16-bit). Its encoding pool for floating-point data consists of several heavyweight compression schemes such as LZ4, Snappy, ZSTD, and Deflate, as well as GPU-optimized formats like Bitcomp (proprietary and closed-source) and GDeflate. GDeflate is a GPU-friendly variant of Deflate that introduces interleaved Huffman coding, where codes are permuted into 32 partitions, though its details are vaguely explained [157]. This enables intra-threadblock parallelism, allowing GPU threads within a block to decode different partitions simultaneously, similar to interleaved bit-packing in FastLanes, where data is distributed across 32 lanes.
For LZ4, nvCOMP enhances its GPU-friendliness by breaking datasets into blocks and compressing/decompressing each block using a thread block [153]. Within each thread block, only a single warp is used to ensure efficient coordination among threads via warp-level primitives.
Below, we compare nvCOMP to G-ALP:

Schema Selection. While providing a set of compression schemes, nvCOMP does not automatically select the best scheme, leaving this decision to the user [155]. In contrast, the FastLanes file format automatically chooses the most suitable compression scheme, with G-ALP being selected only if the data is decimal-like [141].

API. nvCOMP provides two different APIs: a Low-Level API and a High-Level API [155]. The Low-Level API allows users to define the chunk size, enabling data to be compressed in smaller chunks, where each compressed chunk can later be decompressed in parallel using different thread blocks, with one thread block assigned to each compressed chunk. This approach sacrifices compression ratio in favor of increased parallelism. The High-Level API abstracts the chunk size selection from the user by automatically determining the optimal chunk size. It then compresses the data as a whole, adding a header at the start of the compressed data that contains information about the chosen nvCOMP compression scheme. In contrast, G-ALP and FastLanes do not require additional decoding configurations to be set by the user, making them easier to use. We provide similar support to the High-Level API, allowing the entire dataset to be decoded and materialized in global memory. Additionally, we introduce an even more fine-grained Low-Level API capable of delivering decoded data that fits into registers, the fastest form of memory on the GPU.

5.6 Conclusion

General compression schemes have lost their popularity to type-specific encodings in modern file formats designed for CPUs due to their block-based nature—often larger than the cache for decompression—and their lack of data parallelism, preventing SIMDized decoding. By proposing G-ALP, the GPU-optimized version of ALP, which focuses on enforcing a data-parallel layout across all components, including less significant ones such as the patching layout, we demonstrate that type-specific encodings like ALP, despite being more complex than simpler schemes like FFOR, can also be a win-win solution for file

formats used on GPUs. They offer better compression and are faster at decoding thanks to data parallelism.
Additionally, we have shown that by fine-graining our API to its absolute minimal form—an API capable of delivering one value per thread—we enable data delivery to libraries with minimal pressure on local memory usage. This is highly beneficial for ML workloads and databases that tend to process data from many columns and subsequently face local memory pressure.

5.7 FUTURE WORK

FastLanes GPU Reader. G-ALP is a step forward toward developing a GPU reader for the FastLanes file format. Through optimizing G-ALP, we learned that even optimizing a single scheme requires significant effort, indicating that optimizing the entire file format for the GPU would be highly demanding. Therefore, we leave the remaining work, such as supporting additional data types (e.g., strings and nested data types) and integrating different schemes, for future work.

GPU Diversity. We conducted all experiments on a single machine, as detailed in Section 2.3. While it has similar capabilities to the most commonly used GPUs in the cloud, we plan to extend our benchmarking to different types of GPUs to gain a broader perspective on heterogeneous GPU architectures. This will enable us to design a more robust file format that performs efficiently across all GPUs without significant performance cliffs on any specific hardware.

New Benchmarks. To compare G-ALP with other compression schemes, we benchmarked only scan throughput when the data is fully materialized in GPU RAM and aggregation over a single column to evaluate the effect of the one-value-at-a-time API. While these benchmarks provide insight into the performance of G-ALP compared to other schemes, they are far from comprehensive. However, the lack of a standard benchmark or workload for GPUs—similar to TPC-H for CPUs—makes this particularly challenging. We envision the creation of a new set of queries, explicitly collected from real-world GPU use cases, to better design and understand GPU file formats.

Other Floating-Point Data Types. Throughout this chapter, we focused on 32-bit floating-point data. This choice was made to simplify the implementation, as the data-parallel layout of floating-point values in FastLanes perfectly aligns with 32 lanes, matching the GPU WARP model. In contrast, for double-precision (64-bit) data, lane-to-thread mapping must be handled differently. We defer the implementation of G-ALP for other floating-point types, such as 64-bit and 16-bit formats, to future work.

6

FastLanes File Format

This chapter introduces a new open-source big data file format, called FastLanes. It is designed for modern data-parallel execution (SIMD or GPU), and evolves the features of previous data formats such as Parquet, which are the foundation of data lakes, and which increasingly are used in AI pipelines. It does so by avoiding generic compression methods (e.g. Snappy) in favor of lightweight encodings, that are fully data-parallel. To enhance compression ratio, it cascades encodings using a flexible expression encoding *mechanism. This mechanism also enables multi-column compression (MCC), enhancing compression by exploiting correlations between columns, a long-time weakness of columnar storage. We contribute a 2-phase algorithm to find encodings expressions during compression.*

FastLanes also innovates in its API, providing flexible support for partial *decompression, facilitating engines to execute queries on compressed data. FastLanes is designed for fine-grained access, at the level of small batches rather than rowgroups; in order to limit the decompression memory footprint to fit CPU and GPU caches.*

We contribute an open-source implementation of FastLanes in portable (auto-vectorizing) C++. Our evaluation on a corpus of real-world data shows that FastLanes improves compression ratio over Parquet, while strongly accelerating decompression, making it a win-win over the state-of-the-art.

6.1 Introduction

The data formats Apache Parquet and ORC were designed in 2013, and quite similar designs are used in modern analytical systems that have an own storage format, such as DuckDB and Snowflake [51, 158]. Parquet is now the de-facto standard format for data lakes and "lake houses" [159]. However, we argue that changes in hardware and workloads during the past decade call for a re-design.

In the next decade, workloads of analytical data systems and data lakes will increasingly include AI pipelines that perform training or inference [160]. In terms of hardware that runs these workloads, CPUs have become quite diverse (not only x86, but also ARM and RISC-V) and are evolving mostly in novel instructions (SIMD), while AI pipelines increase the importance of GPU- or even TPU-based data processing. In order to efficiently process

	Parquet	BtrBlocks	FastLanes
Heavy-Weight Compression methods	yes	no	no
data-parallel: SIMD/GPU-friendly	no	no	yes
cascading Light-Weight Compression	no	yes	yes
Multi-Column Compression methods	no	no	yes
			compression methods
access granularity	1MB chunk	64K rowgroup	1K vector
can return compressed vectors	no	no	yes
			read access API

Figure 6.1: Feature comparison of big data file formats. BtrBlocks introduced cascading Light-Weight Compression to avoid the Heavy-Weight Compression (e.g. Zstd) used in e.g. Parquet, but its encodings are not data-parallel (SIMD/GPU-friendly). FastLanes is fully data-parallel, can do vector-at-a-time decompression (small footprint), introduces Multi-Column Compression & allows access to compressed vectors.

data on such hardware, algorithms need to harbor *data parallelism*, and specifically need to consist of massive regular computation patterns with absence of data- and control-dependencies. This imposes constraints on what algorithms a data format should employ, e.g. Snappy is the antithesis of a data-parallel algorithm. Current data formats [161, 162] were not designed with this in mind, and struggle to effectively use SIMD and GPUs for decompressing data.

Further, for efficient query processing *after* decompression, data needs to stay in SIMD-friendly representations during execution. Modern query engines such as DuckDB, Velox and Procella therefore added *compressed execution* capabilities, augmenting vectorized query execution with new compressed vector classes, such as constant-vectors, dictionary-vectors and FSST-vectors [49–51]. This trend calls for innovation in data format APIs, to directly deliver compressed vectors from a table scan on request of an engine that can handle this, by only partially decompressing data.

This chapter describes the **FastLanes** data format, marking its v0.1 release in open source. It is designed to efficiently support modern analytics+AI workloads. Its main contribution is a novel Expression Encoding mechanism, supported by an intricate segmented block layout, enabling flexible cascaded encodings and multi-column compression. This allows it to achieve excellent compression ratios while using only simple and ultra-fast data-parallel encodings.

Outline. In Section 6.1.1, we describe our core ideas and in Section 6.1.2 outline the FastLanes design. Section 6.2 explains Expression Encoding. Our novel segmented layout is detailed in Section 6.3. We evaluate vs. Parquet, BtrBlocks and DuckDB in Section 6.4, showing that FastLanes achieves state-of-the-art compression ratios at higher decompression speed. Additional design decisions and related work are covered in resp. Section 6.6 and Section 6.5. We conclude in Section 6.7 and outline future work in Section 7.2.

6.1.1 Design Ideas

From Heavy- to Light-Weight Compression. A columnar layout typically reduces data entropy over row storage, as it concentrates data belonging to the same distribution, making

it more compressible. Heavy-Weight Compression **(HWC)** schemes, also referred to as {general-purpose, block-based, type-agnostic} compression schemes, such as `Snappy` [163] and `Zstd` [92], are used by default in Parquet to compress column chunks. While such compression libraries provide good compression ratios, they are typically CPU-intensive, making decompression considerably slower than accessing uncompressed data [35, 77]. In contrast, Light-Weight Compression **(LWC)** schemes such as `FFOR [140]`, `Delta [140]`, `DICT [140]`, `ALP [141]`, `FastLanes-RLE [140]`, and `FSST [164]` are specifically designed for certain data types and encode data by capturing simple compression patterns. Unlike HWC schemes, it is possible to fully data-parallelize LWC decompression, which makes LWC profit from wide SIMD CPU capabilities, accelerating them up to 64x, which can make accessing compressed data even *faster* than uncompressed data [140]. Data-parallelism also helps GPU decoding performance, as it provides independent work and interleaved memory access for all threads in a GPU warp [165]. However, when considering compression ratio, rather than speed, a micro-benchmark on the `Public BI` dataset shows that adding HWC schemes in ORC, on top of LWCs, improves the compression ratio 3x [166] (`ORC` vs. `ORC+Snappy`). This means that using HWCs is necessary in current big data formats.

Cascading LWC schemes. To achieve the same compression ratio as HWCs while maintaining the speed of LWCs, Cascaded Compression, also known as {recursive, composable} compression [48, 167, 168], has been implemented in BtrBlocks [44]. It combines multiple LWCs to capture a wider range of data patterns. To illustrate how this approach can improve the compression ratio, consider the array:


```
{'Cascading','Cascading','Cascading','Cascading','Cascading',
 'Cascading','Compression','Compression'}
```

This array exhibits two patterns: repeated values and low entropy, which are well-suited for {`RLE`, `DICT`} encodings. However, applying only `DICT` or `RLE` captures just one of these patterns. By first applying `DICT`, the array is transformed into codes `{0,0,0,0,0,0,1,1}` and dictionary `{'Cascading', 'Compression'}`. Then, applying `RLE` turns the codes into `{{0,6},{1,2}}`, achieving better compression.

Multi-Column Compression (MCC) is a new category of compression schemes that takes multiple columns into account, with the key idea that correlation between two columns can be used to infer one column from another, thereby achieving a higher compression ratio [32, 169–171]. Note that compressed columnar formats store columns independently of each other, missing out on this opportunity – leading to the phenomenon that some tables with strongly correlated columns can be more compact in the row-oriented RC format than in Parquet. A simple example is when two columns are completely identical. Our MCC also includes schemes that split one column into multiple sub-columns, which can be encoded individually. For example, string values composed of names and numbers like "Compression101" could be separated into two columns: one for strings and one for integers. This enables further compression e.g., by applying integer-based specific encoding (such as `DELTA` or `FOR`) on the suffix [172, 173].

Vectorized decoding carries over the efficient properties of vectorized execution [174] when applied to decoding compressed data. When a vectorized table scan decompresses a vector, the (compact) compressed data in RAM is decompressed into an uncompressed vector,

which is a small array of e.g., 1024 values that fits into the CPU's L1 cache and is immediately processed by the query pipeline, typically without spilling to RAM. As such, decompression occurs only when the data arrives in the CPU for query processing, keeping it small while in transport, reducing memory, network, and disk bandwidth consumption [35]. Reading while decompressing FastLanes data was found faster than reading uncompressed data (memcpy) [48], because of the reduced bandwidth needs plus ultra-fast auto-vectorized decoding kernels (e.g., decoding 60 values in 1 CPU cycle). The BtrBlocks format not only relies on older encodings that are not data-parallel, but also performs decompression on the full rowgroup level, imposing a large memory footprint. However, allowing fine-grained read access is to avoid overwhelming L1 CPU caches, and even more pressingly, GPU cache and register space [165].

Compressed Execution LWCs (encodings) are more than just a technique to compress data; they capture patterns that can also be used later to optimize query execution on this data. The simplest example is constant encoding, which can tell the query engine that all operations on this column could to be done once instead of on all the values in the column. Modern systems like Procella [175], Velox [50], and DuckDB [51] support *compressed vectors*, where data is both randomly accessible yet still might be encoded in e.g., `DICT`, `FOR` or `FSST`. For example, the `l_tax` column in the TPC-H benchmark is a `decimal(18,2)`, which many systems would implement as a `int64` because it can represent numbers of up to 18 digits, where internally the decimals are multiplied by 100 (due to decimal scale 2). Now suppose, the actual data just contains values between `0.01` and `0.08` (i.e., integers 1-8). Applying LWC, would typically compress such a column using `FOR` and bit-packing (`BP`) in 3 bits per value (`FOR` base 1; and differences 0-7). Whereas legacy systems would decompress this column in their scan into its SQL type `decimal(18,2)` (i.e., `int64`), a system like DuckDB can decompress it into a `int8` (byte). This allows to use 8x thinner SIMD lanes for decompression, accelerating decoding 8x; and also creates better chances for exploiting SIMD in the query e.g. for comparisons or subsequent arithmetic operations, and further, reduces memory pressure e.g. when the column is materialized in a join hash table.

6.1.2 The FastLanes File Format

FastLanes is a project initiated at CWI, designed as a foundation for next-generation big data formats. In the first chapter on FastLanes [140], we focused on significantly improving data decoding performance over the state-of-the-art by introducing a 1024-bit interleaved and Unified Transposed Layout, enabling data-parallel decoding even with scalar code. In the second chapter, we demonstrated that data-parallelized layouts are essential to fully exploit GPU parallelism [165]. Additionally, we designed and implemented ALP [141], a new vectorized and data-parallel encoding for floating-point data. In this chapter, we introduce *Expression Encoding* and design and implement the *FastLanes file format*, with expression encoding at its core.

In our Expression Encoding, relationships within a cascade of encodings for a single column, or between different columns, are represented as a chain of operators. When reading, data in the expression chain is decoded-bottom-up, but not necessarily fully until the end of the chain. This allows modern query engines to choose to partially decode data, yielding *compressed vectors* that query engines can exploit for compressed execution.

For the FastLanes file format we designed and implemented a novel *Segmented Page Layout*,

that allows to store data encoded with any arbitrary nested encoding expressions, while providing an efficient vectorized API to decode the data. This API allows for the fetching and decoding of arbitrary vectors in case of random access or sequences of vectors in cases of full vectorized scans. The segmented layout stores similar parts of encoded data, encoded by an expression, in a single location, along with additional metadata for the encoded data; essentially pointers to this encoded data at the granularity of a vector. Having all encoded data of the same type in one place makes it ideal for recursive compression due to shared types and data semantics. This also enables fine-grained access to the encoded data, at the vector granularity. As a consequence, decoders can perform advanced predicate pushdown, e.g., the base of `FOR` encoding, representing minimum values, can be evaluated first to skip individual vectors in range queries.
Our main contributions are:

- An open-source, high-quality implementation of FastLanes (v0.1) in C++ with absolutely zero code dependencies.
- The design of Expression Encoding, a novel compression model providing a unified approach to cascaded encoding, MCC, compressed execution and vectorized decoding.
- A novel segmented layout to store any arbitrary expression-encoded data, enabling the query engine to interpret underlying encoded data and apply further optimizations.
- The design of a Two-phase Expression Detection algorithm that identifies the optimal expression among a wide pool of possible encoding expressions.
- An evaluation against other file formats on the Public BI dataset, demonstrating that FastLanes achieves faster decompression and better compression ratios.

6.2 Expression Encoding

We first explain the operators that serve as the building blocks of Expression Encoding. We then describe how to serialize an expression and its operators within a file format and how to interpret a serialized expression during decoding at execution time. Finally, we outline the process for identifying an optimal expression within a potentially infinite domain space, as operators can be combined in any order.

6.2.1 Expression Operators

Expression Encoding is similar to white-box compression models [172] or cascaded encodings [44] in that it combines different primitives to achieve better compression. However, operators in FastLanes Expression Encoding are neither simple functions with single tasks, as in white-box compression models, nor entire LWCs, as in cascaded encoding. An operator in FastLanes is a data structure that stores data in a compressed format and transforms it to the next format during decoding. These transformations come from breaking down LWCs into parts that are both reusable and efficient.
For example, the `DICT` operator in FastLanes maintains a pointer to a dictionary and a vector of associated codes, replacing the codes with actual values only if needed. Furthermore, to support vectorized execution and leverage data-parallel layouts, such as a Unified

Table 6.1: FastLanes operators, and the encoded data held by each. An operator is a vector of 1024 values in an executable encoded layout. Encoding operators can be exploited for compressed execution. Take FFOR as an example, which keeps the base separated from the bit-packed data. In the case of simple query predicates such as addition, decoding can be delayed, and the value can be added only to the base. Multiple of these operators can be combined in a chain, forming Encoding Expressions. For instance, FFOR can be combined with DICT to build dictionary encoding with bit-packed codes.

ID	Operator	Encoded Layout
0	FFOR	Bitpacked-data, Base, Bit-width
1	PATCH	Data, Exceptions, Exception Positions
2	DELTA	Deltas, Bases
3	ALP	Data
4	ALP_RD	Left side data, Right side data
6	DICT	Dictionary, Codes
7	Transpose	Transposed Data
8	Cast	Data
9	FRLE	RLE-values, Length
10	CROSS RLE	RLE-values, Indexes
11	FSST	Symbol Table, Compressed Strings
12	FSST12	Symbol Table, Compressed Strings
13	CONSTANT	Single Value
14	EQUALITY	Data or Pointer to data
15	EXTERNAL DICT	Dictionary, Pointer to another column

Transposed Layout or 1024-interleaved layout, each operator holds only 1024 values at a time (a vector). All operations on data are performed in a tight loop over these 1024 values, with consistent work patterns that enable compilers to auto-vectorize [140].

The operators used in FastLanes are summarized in Table 6.1 and are explained as follows:

FFOR. The FFOR operator stores data in a FOR vector, consisting of a base and a vector of bit-packed data – it is Fused with bit-packing. This fusion eliminates a SIMD store and load instruction between the addition resp. subtraction and bit-[unp]packing loop, improving performance. Unlike BtrBlocks and DuckDB, we use only FFOR and avoid a separate bit-packing operator, since the performance of FFOR decoding is almost identical to bit-unpacking.

PATCH. The PATCH operator, inspired by Patched Encoding [35], addresses the vulnerability of encodings such as FFOR and ALP to outliers, by keeping outliers separate from the main vector and reintegrating them during full decompression. We do not fuse patching with encoding operators like FFOR, as having a separate PATCH operator allows us to apply the patching mechanism to enhance any other LWC or operator. For example, if a vector is 95 percent constant, we can still use constant encoding while storing exceptions separately. In FastLanes, we implement only one variation of the possible options for patching, namely {LinkedList (LL_PATCH) [35], SelectionVector (SL_PATCH) [48], Bitmap (BM_PATCH) [48]}: SelectionVector patching, as SelectionVector is the only patching technique capable of being data-parallelized on a GPU [176]. SelectionVector patching uses

a separate array to store the positions of exceptions.

DELTA. The DELTA operator stores delta values in the Unified Transposed Layout[140] that breaks data dependencies among values, accelerating the decoding of delta encoding with scalar code that auto-vectorizes. Unlike BtrBlocks, which completely avoids delta encoding, we argue that delta encoding is crucial for future file formats, particularly for encoding (mostly) sorted data and, more importantly, for encoding offset arrays that are always sorted and are necessary to represent any variable-size data (strings).

ALP. The ALP operator, used specifically for the DOUBLE and FLOAT data type, keeps data in an ALP-encoded format and utilizes our own ALP [141], which significantly improves previous DOUBLE schemes in both speed and compression ratio. ALP is designed for *vectorized execution* and uses an enhanced version of PseudoDecimals [44] to encode doubles as integers if they originated as decimals. Its high speed is due to our implementation in scalar code that auto-vectorizes, using building blocks provided by our FastLanes library [140], and an efficient two-stage compression algorithm that first samples rowgroups and then vectors.

ALP_RD. is used to compress high precision values, by separating the front bits of a float/double from the rest. These front bits are then compressed using primitives designed for the INTEGER data type and, during decoding, are reassembled with the rest of the double using the Glue operator.

Glue. The Glue operator combines two sources of bit-packed data, used to merge the front bits and tail bits in ALP_RD encoding or in one-to-many mappings from MCC schemes.

DICT. The DICT operator stores data in a dictionary-encoded format, consisting of a reference to a dictionary and a vector of codes. We support compressed dictionaries, using either Cast and FSST; because dictionaries must allow random-access (note that e.g., ALP and FFOR store data bit-packed, which does not allow random-access). We chose this approach because otherwise dictionary decoding would become rather block-based: access to dictionary-encoded data would then require to fully decode the dictionary first.
We also support a special **Shuffle Dictionary**, used only for fixed-size data types. It contains the eight most repeated values and uses the SIMD shuffle instruction for decoding, as the dictionary can be loaded into a single register. This dictionary is now only used to encode front bits in ALP_RD.

EXTERNAL-DICT. This operator enables us to use "external codes", i.e. the codes from a different column, with a different dictionary. This is useful to support column correlations with a one-to-one mapping, where the codes of the two columns are the same, but their dictionaries are different.[1]
Transpose. The Transpose operator is applied only during encoding, so the decoded data remains in the Unified Transposed Layout (UTL) after decoding. FastLanes provides a shareable *selection vector*: an array of 1024 integers containing the permutation of the UTL, which vectorized query engines can put in front of vectors decoded by FastLanes to recover the original ingested tuple order. The FastLanes decoder can also be requested to restore this

[1]We also experimented with the opposite idea: sharing a dictionary between columns with different codes – however in our tests this did not improve compression ratio significantly.

order, performing a gather operation on this selection vector.

Cast. The Cast operator keeps values of a column in a different type from what is specified in the schema, to a type which simplifies encoding and query execution. We employ Cast in three scenarios: STRING to INTEGER, allowing query engines to benefit from the SIMD-friendly, fixed-size properties of integers; DOUBLE to INTEGER, enabling the use of the richer INTEGER encoding pool and allowing query engines to operate on integers instead of floating-point data types; and INTEGER to a narrower INTEGER type (e.g., 64-bit to 8-bit). The Cast is a useful end-point for compressed execution.

RLE. The RLE operator stores data in the FastLanes-RLE [140] compressed format, which consists of two vectors: one for repeated values and another for indexes pointing to these repeated values. For full decoding, the RLE values are placed in their correct positions using the indexes. Note that FastLanes-RLE maps RLE to dictionary encoding and applies delta encoding to the indexes. This enables the use of a Unified Transposed Layout to break data dependencies among values, accelerating the decoding of RLE encoding with scalar code that auto-vectorizes.

FSST. The FSST operator compresses a vector of STRING data using FSST [164], a lightweight compression scheme with decompression and compression speeds comparable to, or better than, the best speed-optimized compression methods, such as LZ4. FSST uses a static symbol table (stored in the rowgroup header) that enables random access to individual compressed strings, allowing for query processing directly on compressed data.

FSST12 is an alternative version of FSST that uses 12-bit instead of 8-bit codes [177], allowing it to encode up to 4,096 symbols (each up to 8 bytes long). The larger dictionary allows FSST12 to obtain better compression ratios than FSST on distributions with more entropy; but comes at the cost of a large CPU cache footprint. For instance, JSON and XML benefit more from FSST12.

Cross RLE. The motivation behind this operator is that our data-parallel RLE is very fast but introduces a 128-byte overhead per vector. For a rowgroup of size 64×1024 with very few RLE values, this overhead becomes significant (8KB). To address this issue, we introduce the Cross RLE operator, which applies classical run-length encoding across an entire rowgroup. The main challenge for Cross RLE is efficiently supporting vectorized decoding on the Unified Transposed Layout. To overcome this, we implement the decoding in two steps: first, we traverse the RLE lengths to identify the initial value belonging to the vector currently being decoded, and then we proceed with standard RLE decoding. An additional mapping is performed to correctly decompress into unified transposed layout, but this step is only needed at the boundaries of RLE stretches, adding low overhead.

6.2.2 FastLanes Expression Notation

To store and represent expressions we use a modified form of *Reverse Polish Notation (RPN)*, that separates operators and operands into two distinct RPN-style (postfix order) sub-expressions:

1. **Operators**: Stored as integers, with each operator assigned a unique ID.
2. **Operands**: Stored as integers representing either a column or a segment within a data page in *FastLanes* (Segments are discussed in greater detail in Section 2.2).

Whereas standard RPN requires (string) parsing to tokenize operators and operands, our approach directly stores operators and operands as integers. This design aims to minimize the overhead of interpretation at runtime, while also using little space for expressions. Each operator is uniquely identified based on its type. For example, the `FFOR` operator has distinct IDs for each data type it supports, further reducing the need to interpret the operator's data type. Thus, `FFOR_UINT8` is the version of `FFOR` that operates on 8-bit data, different from `FFOR_UINT16`, for 16-bit data.
Decoding interpretation consists of initializing a chain of physical expressions, by reading the operator and operand arrays from a column descriptor inside the rowgroup file-footer. These physical expressions are initialized by (i) looking up function pointers from operand IDs, and (ii) binding parameters by looking up values and offsets in the column descriptor, which contains encoding parameters such as e.g. the bit-width for bit[un]packing in `FFOR`, as well as segment descriptions that point to raw bytes in the rowgroup. Execution of [en/de]codingthen calls these functions in the physical expressions one after the other.

6.2.3 Expression Detection

Expression Encoding enables a file format to encode data using any combination of operators. This flexibility introduces a challenge in identifying suitable expressions that achieve both fast decompression and high compression ratios for a given table, as the search space is infinite. We address this challenge with a two-phase approach for expression selection: a *rule-based* phase for detecting relationships between columns and determining the appropriate type for each column, followed by a *sample-based* encoding phase that selects an expression from a predetermined pool of expressions.

Rule-Based Operator Selector. The process of operator selection or expression creation begins by applying rules in the order they are defined. The FastLanes v0.1 rule set consists of the following:
1. Constant: We first identify constant columns where all values are identical. These columns are represented by an expression with only one operator, `CONSTANT`. The constant value is not stored directly; instead, the Min-Max information in the rowgroup footer is used to retain this value. This decision allows the reader to use this column in a query without fetching any data.
2. Equality: We check for equality columns where the values in two columns are (almost) completely identical row-by-row. In this case, the second column is encoded with an expression consisting of only one operator, `EQUAL`, and as operand a `{column-id}`.
3. String as Numerical: It is not uncommon for database users to select a string type as a fallback data type for a column that contains mostly numerical data [103]. Converting these

strings to numerical types, such as integers or doubles, improves compression efficiency, as numerical data compresses better than strings (as shown in Section 2.3), and simplifies processing since numerical data is fixed-size, making it suited for SIMD instructions. This rule first detects such columns and then selects the appropriate numerical type. A `CAST` operator is added to the expression to retrieve the original type if necessary.
4. Double as Integer: Similar to "String as Numerical," this rule aims to select a more efficient data type when possible, particularly for double columns that consistently have a zero after the decimal point. A `CAST` operator is added to the expression to retrieve the original type if necessary.
5. Narrower Types for Integer: Similar to "String as Numerical" and "Double as Integer," this rule aims to select a more efficient data type when possible, particularly a narrower integer data type. The narrowest integer type is determined based on the number of bits required to represent the maximum value. A `CAST` operator is added to the expression to retrieve the original type if necessary.
6. One-to-One Map: In a one-to-one correlation, a specific value "X" in one column is always associated with a single specific value "Y" in another column. In this case, the first column proceeds to the second phase to determine the best expression, with the caveat that only expressions with a dictionary as the root are considered. For the other column, we store only a reference to the dictionary column, and the expression selection stops here by choosing `EXTERNAL_DICTIONARY`.

Sampling-Based Encoding. After the Rule-Based Operator Selector, three categories of columns – constant, equal, and one-to-one map – are removed from the pipeline for choosing the optimal expression. For the remaining columns, we use a sampling-and-try approach to select the best encoding expression from a limited predefined **expression pool**, shown in Table 6.2. This pool consists of expressions we derived from trying a large set of combinations of encodings on the Public BI datasets; where we kept those encoding expressions that ended up being the best for some column.
We call our sampling method *three-way*: FastLanes simply takes the first, last and middle vector (each of 1024 values) in the rowgroup and tries compressing this limited data with all encoding expressions in our pool for that datatype. The key intuition behind this sampling strategy is that many tables exhibit locality, while gradually changing the data distribution throughout the rowgroup.
Figure 6.2 shows a benchmark of compression ratio achieved on the Public BI dataset using two sampling strategies and various sample sizes; where 100% compression ratio was achieved by choosing the best encoding expression after measuring the compression ratio on *all* vectors of the rowgroup (i.e., no sampling). The sequential strategy uses a front-biased strategy, whereas the three-way strategy recursively applies a binary-search approach: after sampling the first and last vector, it incrementally probes the middle of the largest unexplored space. We see that this strategy, after just three vectors (hence: three-way) achieves more than 99% accuracy in terms of compression ratio, comparing quite favorably to front-biased sampling.

Expression (where $X \in \{08, 16, 32\}$)	Count	Popularity (%)
string columns		
CROSS_RLE_STR	210	9.17%
FSST_DICT_STR_FFOR_SLPATCH_UX	191	8.34%
FSST_DICT_STR_FFOR_UX	166	7.25%
CONSTANT_STR	81	3.54%
EXTERNAL_FSST_DICT_STR_UX	41	1.79%
FSST_DELTA_SLPATCH	33	1.44%
FSST_DELTA	21	0.92%
FSST12_DICT_STR_FFOR_SLPATCH_UX	8	0.35%
FSST12_DELTA_SLPATCH	7	0.31%
RLE_STR_SLPATCH_UX	2	0.09%
FSST12_DELTA	2	0.09%
RLE_STR_UX	1	0.04%
numeric columns		
CONSTANT_INTEGER	334	14.59%
FFOR_SLPATCH_INTEGER	227	9.92%
DICT_INTEGER_FFOR_SLPATCH_UX	210	9.17%
DICT_INTEGER_FFOR_UX	190	8.30%
CROSS_RLE_INTEGER	121	5.29%
FFOR_INTEGER	47	2.05%
EXTERNAL_DICT_INTEGER_UX	24	1.05%
RLE_INTEGER_UX	15	0.66%
RLE_INTEGER_SLPATCH_UX	3	0.13%
floating-point columns		
ALP_DBL	87	3.80%
DICT_DBL_FFOR_SLPATCH_UX	38	1.66%
RLE_DBL_UX	30	1.31%
DICT_DBL_FFOR_UX	28	1.22%
CONSTANT_DBL	18	0.79%
ALP_RD_DBL	17	0.74%
EXTERNAL_DICT_DBL_UX	12	0.52%
CROSS_RLE_DBL	2	0.09%
RLE_DBL_SLPATCH_UX	1	0.04%
correlated columns		
EQUALITY	56	2.45%

Table 6.2: The FastLanes v0.1 Expression Pool: Sorted by Category & Popularity in being chosen in encoding the Public BI benchmark. For string columns, run-length encoding with long stretches dominates (CROSS_RLE_STR), and second most effective are FSST-compressed dictionary encoding with exceptions (FSST_DICT_STR_FFOR_SLPATCH_UX). This pool was chosen by exhaustive testing of a wide spectrum of expressions encodings on Public BI, and retaining the winners.

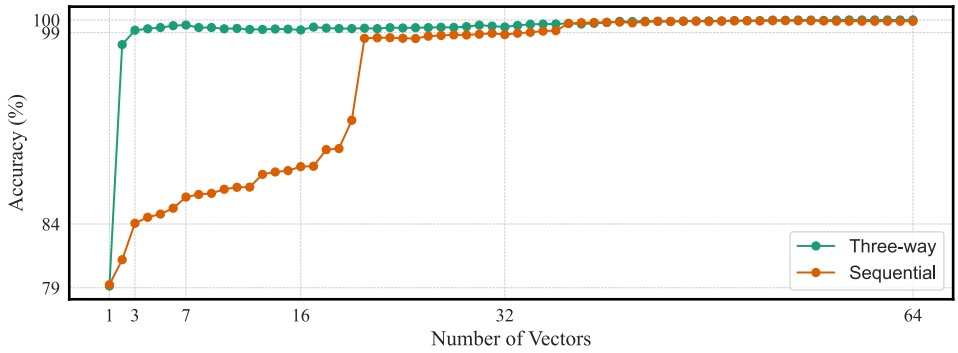


Figure 6.2: Compression ratio accuracies of Sequential and Three-Way sampling methods. Three-way sampling, vectors at positions 0, 32, and 64 achieve more than 99% accuracy.

6.3 FastLanes File Format

In this section, we first explain the file format from a high-level perspective, beginning with the rowgroup, then moving to the column chunk, and finally delving down to the *segment*, the fundamental building block of the FastLanes file format. We also demonstrate how segments are used to store the encoded data of an expression. Additionally, we provide a detailed example of how a table is stored in the FastLanes format.

File Format Overview. The FastLanes file format consists of two main components: the footer and the data. The footer, in v0.1 still stored in JSON format (though this will change in a later version), contains all the necessary information to access, decode, and decrypt the data, along with statistics such as Min-Max values. The data itself is stored in a binary format after being expression-encoded. We propose storing the footer metadata separately from the data – e.g., in different files or objects within an S3 bucket in cloud storage – query engines can process metadata first, possibly from a cache or catalog that consolidates metadata for many rowgroups, enabling optimizations like projection pushdown and zone map filtering that avoid accessing data files unnecessarily.

Rowgroup. FastLanes first divides a table horizontally into smaller mini-tables called *rowgroups*. Each rowgroup stores records using the PAX layout [178], which keeps the attribute values of each record together in the same file, while the attributes themselves are stored in a DSM (columnar) layout [179, 180]. All decisions in FastLanes, such as expression detection, are made on a per-rowgroup basis, allowing for more fine-grained tuning and adaptability to data, rather than applying a single expression to an entire column. Additionally, we store statistics for each rowgroup, such as Min and Max values, enabling the ability to skip entire rowgroups for range queries. The size of a rowgroup in FastLanes is a fixed number of records, similar to ORC, and is always a multiple of 1024. This design ensures compatibility with data-parallel encodings [140], as these encodings require 1024-value batches to fully leverage SIMD registers or all threads within a GPU warp [165]. In contrast, Parquet uses fixed physical sizes for rowgroups, resulting in a variable number of

```
{   "Rowgroup size in terms of number of vectors": 2,
    "Rowgroup binary size": 26,
    "Rowgroup offset within binary file": 0,
    "Rowgroup ID": "0",
    "Column Descriptors": ["Explained below"]         }
```

Figure 6.3: Row-group Descriptor in JSON format for the first row-group in Table 6.7, stored separately from the row-group binary data. This row-group contains two vectors and is located at an offset of 0 bytes in the binary FastLanes file format, with a size of 26 bytes. To retrieve this row-group, 26 bytes starting from offset 0 need to be loaded.

```
{   "type": "STRING",
    "Column offset": 0,
    "Column binary size": 23,
    "Expression": "DICT_FFOR_UINT8"],
    "Column Index": 0,
    "Segments Descriptors": ["Explained below"],      }
```

Figure 6.4: Column Descriptor for the first column in Table 6.7, stored within the row-group descriptor depicted above. The Expression for decoding and encoding code ("DICT_FFOR_UINT8") gets mapped to an array of operators and an array of operands, a form a Reverse Polish Notation.

```
{   "Entrypoint offset": 16,
    "Entrypoint binary size": 2,
    "Data offset": 18,
    "Data binary size": 2                             }
```

Figure 6.5: Segment Descriptor for the segment that stores bases for FFOR. The entry point array is of size 2, as each entry point is a 8-bit unsigned integer. There are two entry points needed, since the row-group size is 2 vectors. The size of the data is 2 bytes: 1 byte for each 8-bits base (1 byte).

records.

Rowgroup Descriptor. Each rowgroup in the FastLanes file format is associated with a descriptor in the footer. This descriptor includes the size of the rowgroup (in terms of the number of vectors), along with the size and offset specifying where the rowgroup starts in the binary data and the number of subsequent bytes it occupies. The descriptor enables the query engine to fetch only the relevant bytes from the binary file, skipping unnecessary rowgroups through zonemap filtering. Additionally, it contains an array of column descriptors, which are explained later. An example of a rowgroup descriptor is shown in Figure 6.3.

Column Chunk. Within each rowgroup, there is exactly one column chunk per column, storing the data of that column in a columnar format after being expression-encoded. By keeping all data of a column in a contiguous location, this setup enables query engines to perform projection pushdown, allowing them to load only the columns relevant to the query instead of loading all columns.

Column Chunk Descriptor. For every ColumnChunk in a FastLanes rowgroup, there is a corresponding descriptor in the footer. This descriptor includes the type defined by the schema, segment descriptors (explained later), the size and offset indicating where the column starts in the binary data, and the number of subsequent bytes belonging to the

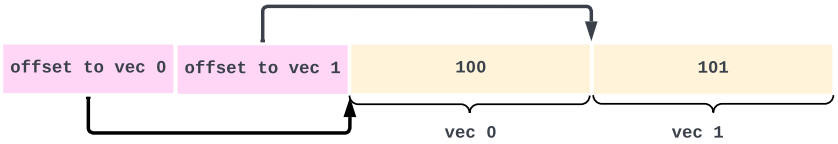


Figure 6.6: Example segment storing the bases of the FFOR expression of the ID column with a row-group size of 2 vectors. The segment holds bases for 2 vectors {100, 101}, with an entry point array at the start that points to the base of each vector to enable vectorized decoding. Storing bases in one location allows further compression of these bases, e.g., with FFOR. Bases can also be exploited in queries (e.g., ID>101) to skip vectors to provide per-vector min-max stats.

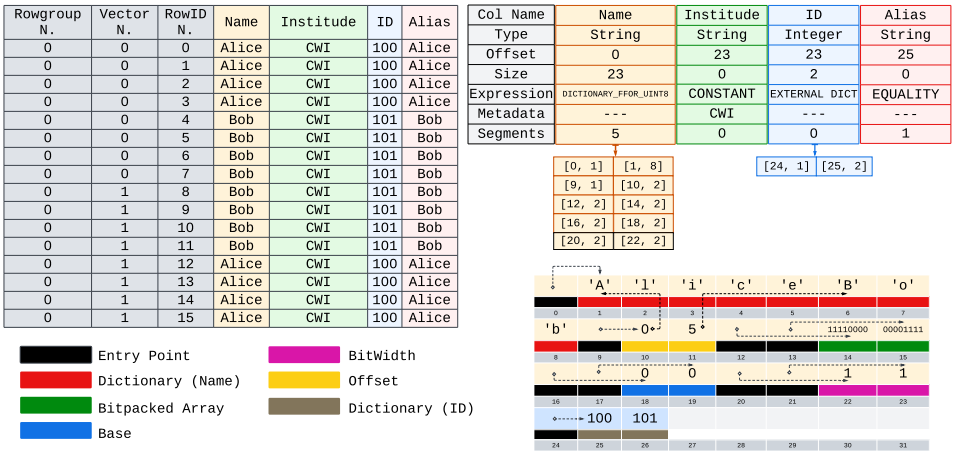


Figure 6.7: FastLanes file layout for a table with columns Name, Institute, and ID. In gray we also draw inexistent columns rowgroup, vector and rowId, indicating which row belongs to which rowgroup and vector. We reduced the rowgroup size to 2 vectors of size 8 (v0.1 defaults are 64x1024). The footer is shown in top right, with required fields Col Name; Type; Offset, representing the binary offset of this column in the rowgroup; Size, allowing to compute the end of the column chunk; Expression, specifying the encoding-expression of this column; Metadata and Segments, an array of two byte ranges [offset, size]. The first range specifies where the segment "entry point" array (with one entry point per vector) starts within this column chunk and its size, while the second range specifies where the segment data starts and its size. We assign a color to each column, which is also applied to the raw bytes in the FastLanes binary data, depicted in the bottom right. Each box represents one byte, tagged below with a color that matches the legend, and a byte position in gray. FastLanes used Dictionary_FFOR_UINT8 encoding for the Name column, as the values come from a small domain. Constant encoding is used for the Institute column, as all values are equal. One-to-One Mapping is selected for the ID column, since it completely correlated with the Name column. Finally, the Alias column is compressed using Equality, as it is a copy of Name. For the first column in the binary format, there are a total of five segments: (1) one for bytes of dictionary values ("Alice" and "Bob"), (2) one for offsets of strings stored in the first segment ("0,5"), (3) one for the bit-packed array, which in this case requires two bytes as the 16 values can be represented in 1 byte, (4) one for the base values (0 and 0), and (5) one for bit widths (1 and 1). For Institute there is no segment, as the constant ("CWI") is stored in the metadata. For ID, we store only the dictionary with one segment consisting of values [100, 101].

column. These details enable the query engine to access relevant columns while skipping unnecessary ones, an optimization known as projection pushdown. The descriptor also includes the column index and statistics, such as the maximum value. An example of a column descriptor is shown in Figure 6.4.

Segment. A segment stores encoded data of the same nature—data that has the same role in

encoding and shares the same type—resulting from encoding a column chunk (multiple vectors) through the expression encoder. It enables fine-grained access to this encoded data (at the granularity of a vector at a time, 1024 values) to support vectorized decoding. Each segment achieves this by storing an additional *entry points* array alongside the data itself, which keeps track of the offset to the start of the data for each vector. Note that the segment stores data from the assigned source consecutively after this entry points array.

Segment Descriptor. For each segment in the file format, there will be a descriptor in the file footer containing two key pieces of information: the entry point offset and size, and the data offset and size. These fields determine the exact location and extent of each segment within the binary data file. An example of segment descriptor is shown in the middle of Figure 6.5

6.4 Evaluation

Hardware. We conducted all experiments on an EC2 instance `i4i_4xlarge`, with Intel Xeon (Ice Lake) CPU, 16 vCPUs and 128 GiB RAM. FastLanes is portable accross multiple operating systems and compilers, and we have previously evaluated its encodings also on Apple and Graviton ARM hardware [140]; however, BtrBlocks depends on x86 intrinsics, which is why we chose this platform. **Data Formats.** FastLanes v0.1 is released under an

MIT license in our GitHub repository[2]. All experiments are released separately in a dedicated repository[3]. For Parquet, there are several open-source implementations; we use the implementation in DuckDB v1.2 [181], as it employs the latest Parquet encodings [182] and is widely used for writing Parquet files. We compare two variants of Parquet: Parquet+Snappy, widely used in practice, and Parquet+Zstd, which offers the best compression ratio. For BtrBlocks, we use the original implementation provided by the authors of BtrBlocks [183], run with its default settings at cascading level 2.
We also evaluate DuckDB's native format. Note that DuckDB does not provide any API to directly determine the storage size occupied by a table, making it challenging to accurately measure DuckDB's compression performance. We replicate each sample dataset until the number of samples reaches at least 10 and is a multiple of 1024×120, as DuckDB begins compressing data only when a rowgroup size reaches 1024×120. This setup ensures that the resulting files are sufficiently large, minimizing inaccuracies caused by DuckDB's storage being allocated in fixed increments of 256 KB, thus allowing a fair evaluation of its compression performance.

Data. We chose data from the `PUBLIC_BI` [103, 104] benchmark as a basis to design and compare FastLanes against other file formats, and also used it to identify the expressions encodings (see Section 6.2.3). The `PUBLIC_BI` dataset is particularly relevant because it captures a wide variety of data distributions, is derived from real-world datasets, and has previously been used in the analysis, design, and evaluation of other encoding schemes such as `ALP, FSST, White-Box Compression, C3, Chimp, Chimp128`.
We used 36 datasets from `PUBLIC_BI`, summarized in Table 6.3, consisting of 2,289 columns. For consistency, we only considered the first table from each dataset to ensure

[2] `https://github.com/cwida/FastLanes`
[3] `https://github.com/cwida/fastlanes-vldb2025`

equal-sized samples, as datasets vary in the number of tables. An exception was made for datasets with very few records, such as `TrainsUK1`, where we used Table 2 instead of Table 1.
Additionally, for datasets with similar schemas, such as `Redfin1`, `Redfin2`, `Redfin3`, and `Redfin4`, only the first dataset was included to avoid the effect of redundant tables on the results. For all experiments in this section, we selected 65,536 records (64 vectors) from each table to ensure fair benchmarking, as BtrBlocks also uses this number as the rowgroup size. Exceptions were made for the datasets `CommonGovernment`, `Generico`, and `USCensus`, where only 32,768 rows were used, as including 65,536 rows would result in file sizes exceeding 100MB.

Compression ratio. Table 6.3 summarizes the compression ratios for all evaluated file formats. All ratios are reported relative to FastLanes, where positive values indicate the percentage by which FastLanes compresses data more effectively, and negative values indicate the percentage by which another format compresses better than FastLanes. As shown, FastLanes achieves the highest compression ratio among all evaluated file formats. Parquet+Zstd is the closest competitor, compressing only 2% less efficiently than FastLanes. In contrast, the commonly used default, Parquet+Snappy, results in 40% more data compared to FastLanes.

Decoding and Encoding Speed. Next, we evaluate the decoding and encoding speed of all file formats. The results are shown in Table 6.4. FastLanes is faster on all datasets and, on average, is 43 times faster than Parquet+Snappy, 44 times faster than Parquet+ZSTD, 7 times faster than BtrBlocks, and 29 times faster than DuckDB.
Regarding encoding speed, we note that the FastLanes v0.1 code path is extremely basic and no effort whatsoever has been made to make it fast. With all optimization still on the table, we note it is already faster than BtrBlocks.

Random access. Next, we evaluate the random access time of all file formats. To do so, we execute the following query: `"SELECT * FROM read_parquet() LIMIT 1 OFFSET 0"` in DuckDB to retrieve only the first row from a Parquet file. For BtrBlocks, we fully decompress the entire block. While this could be optimized further by decoding only the first element of the last recursion of BtrBlocks, this is not possible as it would require a complete reimplementation of BtrBlocks with a new API that supports this.
The results are shown in Table 6.5. FastLanes takes 0.14 milliseconds to retrieve the first value from all datasets, making it 315 times faster than Parquet+Snappy, 416 times faster than Parquet+ZSTD, 800 times faster than BtrBlocks, and 5 times faster than DuckDB. DuckDB is the closest to FastLanes in performance. This benchmark demonstrates that block-based compression methods are extremely inefficient for random access, as they require decompressing the entire block to access a single value. In contrast, the vectorized decoding model used in both FastLanes and DuckDB provides a good balance between compression ratio and small enough block sizes to enable efficient tuple retrieval.

SIMD. To evaluate in howfar fully data-parallelized encodings improve with SIMD, we benchmark the total decoding time of FastLanes on Intel Ice Lake using three different compilation flags: `-O3`, `-O3 -mavx2`, and `-O3 -mavx512dq`.
The results are shown in Table 6.6. As observed, AVX512 improves decoding time by nearly

Table 6.3: Compression ratios of file formats relative to FastLanes. Positive percentages indicate better compression performance by FastLanes, while negative ones show superior performance by other formats. FastLanes achieves the highest overall compression ratio, closely followed by Parquet+Zstd (2% less efficient). The default Parquet+Snappy format results in 40% larger compressed data.

Table	CSV	FastLanes	Parquet		BtrBlocks	DuckDB
Name		V0.1	Snappy	Zstd	[183]	v1.2
Arade	5.44	0.90	+92.00%	+82.00%	+17.00%	+22.00%
Bimbo	2.72	0.31	+108.00%	+42.00%	-2.00%	+82.00%
CMSpr	14.02	2.86	+43.00%	+10.00%	+11.00%	+63.00%
CityM	22.98	8.53	+14.00%	-21.00%	+16.00%	+39.00%
Commo	25.20	2.22	-8.00%	-39.00%	+3.00%	+103.00%
Corpo	11.62	2.40	+7.00%	-24.00%	+88.00%	+57.00%
Eixo	52.56	4.50	+18.00%	-21.00%	+15.00%	+64.00%
Euro2	12.97	5.10	-9.00%	-39.00%	+21.00%	+10.00%
Food	2.01	0.48	+15.00%	-10.00%	-18.00%	-20.00%
Gener	18.34	0.86	+50.00%	-24.00%	-8.00%	+83.00%
HashT	82.52	15.85	+51.00%	-4.00%	+55.00%	+103.00%
Hatre	23.19	8.63	+13.00%	-23.00%	+21.00%	+41.00%
IGloc	5.26	1.58	+24.00%	+6.00%	-5.00%	+21.00%
MLB	11.41	2.11	+73.00%	+4.00%	+3.00%	+84.00%
MedPa	15.30	3.56	+33.00%	+7.00%	+14.00%	+67.00%
Medic	13.03	3.28	+29.00%	+9.00%	+7.00%	+27.00%
Motos	36.94	1.76	+57.00%	-24.00%	-9.00%	+150.00%
Mulhe	54.61	4.56	+18.00%	-21.00%	+14.00%	+68.00%
NYC	39.54	3.70	-11.00%	-31.00%	+26.00%	+66.00%
PanCr	15.02	3.25	+42.00%	+6.00%	+7.00%	+59.00%
Physi	15.30	3.53	+34.00%	+8.00%	+16.00%	+64.00%
Provi	15.30	3.52	+37.00%	+13.00%	+13.00%	+64.00%
RealE	17.57	3.61	+63.00%	+10.00%	+23.00%	+39.00%
Redfi	29.31	8.17	+13.00%	+1.00%	+13.00%	+55.00%
Renta	65.94	12.49	+22.00%	+8.00%	+18.00%	+46.00%
Roman	21.07	8.16	+11.00%	-28.00%	+19.00%	+50.00%
Salar	48.65	8.45	+4.00%	-8.00%	+20.00%	+72.00%
Table	17.42	0.49	-23.00%	-47.00%	-11.00%	+209.00%
Taxpa	15.28	3.49	+38.00%	+13.00%	+19.00%	+68.00%
Telco	47.41	17.20	+121.00%	+54.00%	+3.00%	+59.00%
Train	19.93	2.62	+95.00%	+38.00%	+5.00%	+134.00%
Train	19.87	2.96	+67.00%	-0.00%	+15.00%	+85.00%
USCen	54.63	3.44	+72.00%	+13.00%	+6.00%	+147.00%
Uberl	54.52	4.59	+17.00%	-22.00%	+12.00%	+65.00%
Wins	56.19	11.80	+87.00%	+30.00%	+13.00%	+76.00%
YaleL	16.96	1.10	+14.00%	-14.00%	+25.00%	+122.00%
Total	980.03mg	172.05(mg)	+41.00%	+2.00%	+18.00%	+66.00%

Table 6.4: Decoding/encoding throughput of file formats, based on the number of rowgroups decoded/encoded per second.

File Format	Total Decoding Time (ms)	Decoding (rowgroup/s)	Total Encoding Time (ms)	Encoding (rowgroup/s)
FastLanes	16.32	61.27	81341.63	0.012
Parquet+Snappy	712.55	1.40	5867.07	0.17
Parquet+Zstd	731.45	1.37	6927.52	0.14
BtrBlocks	115.43	8.66	111091.39	0.009
DuckDB	483.45	2.07	20347.82	0.05

Table 6.5: Random access time comparison across different file formats. The result is presented in terms of the millisecond taken by FastLanes and how many times FastLanes is faster than others. FastLanes achieves the fastest access time. The results highlight the inefficiency of block-based compression for random access and the advantage of vectorized decoding in balancing compression ratio and retrieval speed.

Table	FastLanes	Parquet		BtrBlocks	DuckDB
Name	V0.1	Snappy	Zstd	[183]	v1.2
Total	0.14053	315.62x	413.66x	813.57x	5.96x

40%. To further emphasize the necessity of next-generation file formats for data-parallel encodings, we repeat the same benchmark for Parquet+Zstd, Parquet+Snappy, and BtrBlocks. The observed performance gain is negligible. While BtrBlocks uses explicit SIMD instructions, it employs non-fully data-parallel layouts, which limit its ability to benefit from AVX512, as clearly shown in Table 6.6. This benchmark clearly demonstrates the importance of fully data-parallel encodings.

Expression Pool. To measure the effect of each expression in our pool, we conducted an experiment where we measured the impact of each expression encoding included, compared to when it was removed from the pool. The results indicate how much an expression practically improves the compression ratio performance and decompression time of FastLanes[4]
The results are shown in Table 6.7. `Dictionary` encoding has the most significant impact on the compression ratio, improving it by 40%, followed by `DELTA` decoding, which improves it by 6%. Based on these results, we address the following questions, which could serve as guidelines for future file formats, including FastLanes.

Exception Handling. Despite its proven benefits [35, 184], almost all new file formats, such as BtrBlocks, DuckDB's native file format, and Nimble from Meta, avoid supporting any exception handling mechanism. The `PATCH` operator, which handles exceptions, improves the compression ratio by 2.5% in the presence of all schemes, demonstrating its significance. We consider patching a first-class citizen in FastLanes, though we notice that it has considerably slowed down decompression.
FSST. FSST12 and FSST work similarly, with the key difference that FSST12 uses 12-bit

[4]This micro-benchmark was conducted on an Apple MacBook Pro M4.

Table 6.6: Total decoding time of FastLanes, Parquet, and BtrBlocks on Intel Ice Lake using different SIMD compilation flags. AVX512 improves decoding time by nearly 40% on FastLanes, demonstrating the efficiency of data-parallelized encodings. In contrast, the performance gains for Parquet+Zstd, Parquet+Snappy, and BtrBlocks are negligible. BtrBlocks only works on machines with AVX2/AVX512 instruction sets.

ISA	FastLanes		Parquet+Snappy		Parquet+Zstd		BtrBlocks	
	Time(ms)	Speedup	Time(ms)	Speedup	Time(ms)	Speedup	Time(ms)	Speedup
SSE	23.16	-	2110.27	-	2152.29	-	✖	✖
AVX2	18.96	22.10%	2057.16	2.58%	2167.52	-0.70%	113.09	-
AVX512	16.32	41.87%	2070.90	1.90%	2197.58	-2.06%	115.43	-2.07%

Table 6.7: Improvement brought by adding each scheme to the pool compared to having it removed. Usefulness is a mix of impact on compression ratio, decompression speed (and code complexity - but this is harder to quantify).

Expression	Compression Ratio	Decompression Speed
Dictionary	+42.36%	+44.19%
DELTA	+5.92%	-1.91%
Equality	+4.70%	+2.66%
ALP	+4.36%	-7.28%
FSST	+3.86%	+3.84%
Patch	+2.51%	-7.11%
FFOR	+1.30%	+4.48%
One-to-One Map	+1.17%	+9.47%
FSST12	+0.84%	+2.62%
Cast	+0.78%	+10.16%
RLE	+0.69%	+6.65%
CROSS RLE	+0.65%	+16.27%
ALP RD	+0.57%	-2.30%
Frequency	+0.09%	+2.06%
Constant	+0.00%	+2.92%

symbols, allowing it to capture 16 times more symbols at the cost of 4-bit longer codes. This raises the question: do future file formats need FSST, FSST12, or both?
By looking at the table, FSST improves the compression ratio by almost 4%, which is significant, while FSST12 contributes only 1%, which is still meaningful. Despite not having as much impact as FSST, a detailed analysis shows that FSST12 performs very well on long string columns, making our file format more future-proof for handling long strings. Therefore, we support both FSST12 and FSST in FastLanes.

Frequency. BtrBlocks argues for using frequency encoding, which considers the most commonly used value as a default and stores only values that do not match the most frequent one. In a sense, it is similar to constant encoding with exceptions.
Our analysis, summarized in Table 6.7, shows that this scheme brings only a 0.05% improvement, which is very insignificant compared to the complexity it adds to the file

format. Therefore, in FastLanes, we do not support the Frequency encoding.

MCC. MCC schemes, including Equality with (4.7%, 2.6%), One-to-One Mapping with (1.2%, 9.47%), and Cast with (0.78%, 10.16%), bring an overall improvement of approximately (8%, 20%) to the compression ratio and decoding speed of FastLanes, which is very significant. Therefore, we support MCC schemes as a first-class citizen of FastLanes and continue to explore further improvements to our MCC schemes.

6.5 Related Work

Today, there are multiple open columnar [185] file formats, including Parquet [1], RCFile [15], ORC [16], BtrBlocks [44], DuckDB [51, 186], Albis [187], Carbon [188], DataBlocks [189], Artus [175], Capacitor [190], LanceDB [191], Bullion [160], and Nimble [160]. These formats have been analyzed and surveyed in [48, 161, 162, 192]. In this section, we review BtrBlocks as the current state-of-the-art file format, the first (and only) to implement cascaded encoding. We also examine studies within the database context, focusing specifically on `Cascaded Encoding` and `Multi-Column Compression (MCC)`.

6.5.1 BtrBlocks

BtrBlocks implements cascaded compression through recursion, where an entire column chunk is compressed recursively using multiple lightweight compression schemes (LWCs). Here we highlight some key reasons FastLanes provides advantages over BtrBlocks.

Block-Based Compression: Despite using LWCs that could potentially support vectorized decoding, BtrBlocks' cascading compression reverts to a coarse-grained approach. This is due to the recursive nature of the implementation, which requires an entire rowgroup (64*1024 values) to be fully [de]compressed multiple times for each LWC used in a combination. One could argue that the size 64*1024 could be reduced to one vector of 1024, and that repeating the process of recursive decompression for each vector could support vectorized decoding. However, this approach is not feasible because, for example, the crucial dictionary encoding code-path would then get executed separately for each vector, resulting in a separate dictionary being stored for each vector. This could lead to dictionaries with potentially repeated values across vectors and significantly worse compression ratios.

No Compressed Execution Support: BtrBlocks always completely decompresses values. While one could extend its cascaded decoding to support compressed execution, this would only provide a limited form by skipping decoding of the final recursion level. This limitation arises from the recursive nature of its implementation, where lower levels of encoding remain opaque and can only be accessed after full decoding.

Not Fully Data-Parallelized: BtrBlocks relies on 128-bit interleaved bit-packing provided by the FastPFOR library. At best, a 128-bit interleaved layout can only use a SIMD register of width 128 bits or only 4 threads of a 32-threaded warp – and there is no BtrBlock GPU decoder yet.

Missing Schemes: The BtrBlocks scheme pool lacks three critical schemes: `ALP`, `Patching`, and `Delta encoding`. `ALP` is a state-of-the-art encoding for floating-point data and an essential LWC scheme to achieve better compression ratios than `Zstd`. `Delta`

encoding is crucial in niche domains (e.g., timeseries data) but is also a useful component for encoding offsets needed for string storage.

Hardcoded: Btrblocks implements cascaded compression with hardcoded configurations. For example, in all cascades involving dictionary encoding, uint32 is always used for codes. In contrast, the FastLanes implementation of cascaded encoding allows a expressions of any combination of operators; and even though we limit the space of possibilities with an expression pool, it offers multiple variants for dictionary codes. In FastLanes, dictionary codes are typically further compressed with FFOR. Note that in the FFOR of FastLanes the bit-width is a parameter that gets stored in a column segment, hence it can vary from vector to vector.
Dependencies: the BtrBlocks implementation relies on several external dependencies, which complicates its use in practical systems. We argue that a file format should be implemented with zero external dependencies, following the DuckDB and SQLite implementation model, making it usable as an embeddable library for any query execution engine.

6.5.2 Encoding/Compression.

Encoding/compression is frequently studied in database systems, with a focus on improving decoding speed [35, 71, 75, 140, 184, 193–203], optimizing encoding/compression selection [204, 205], enhancing compression ratios [173, 206–211], integrating compression with query execution [173, 212–219], evaluating predicates on encoded data [69, 70, 220, 221], and GPU encoding/compression [168]. HWC schemes, such as Zstd [92], Snappy [163], and LZ4 [93], are the default in most open file formats. Several LWC schemes have also been developed to encode specific data types, such as DOUBLE [43, 44, 95–97, 110, 122, 124, 141], INTEGER [46, 48, 75, 140, 184], and STRING [73, 164, 222]. Grammar-based compression schemes like Sequitur [223], Re-Pair [224], and GLZA [225] have been proposed to compress data by building a context-free grammar for it. However, these grammar-based schemes are generally unsuitable for file formats due to their slower decompression speeds [164].

Cascaded Encoding. Fang *et al.* [168] propose cascaded encoding, which combines LWC schemes to improve compression ratios. Damme *et al.* classify LWC schemes into logical and physical compression categories and study how well they can be combined [226, 227]. However, their work is limited to integer columns and combinations of at most two LWC schemes. Afroozeh *et al.* [48, 167] propose a Composable Compression Model that decomposes LWC schemes into several efficient functions that can later be used to construct more complex encodings, though this work focuses on decoding speed rather than compression ratios. BtrBlocks [44] implements cascaded encoding recursively, while Nvidia's nvCOMP [228] applies cascaded encoding recursively for GPUs, though it is limited to a single variation of [DICT, RLE, BITPACK].

Multi-Column Compression. White-box Compression [172] proposes a conceptual model that represents logical columns in tabular data as an openly defined function over some physically stored columns, allowing the query optimizer to enable optimizations such as improved filter predicate pushdown during query execution. PIDS [173] identifies common patterns in string attributes using an unsupervised approach and uses the

discovered patterns to split each attribute into sub-attributes. These sub-attributes can then be encoded individually, which enables future engines to push down many query operators to sub-attributes, thereby minimizing I/O and potentially costly comparisons, resulting in faster execution of query operators. C3 [32] proposes six MCC schemes—Equality, 1To1Dict, 1toNDict, Numerical, DFOR, SharedDic – to address a key limitation of column stores relative to row stores, namely that they compress attributes of each record in isolation. Corra [170], similar to C3, looks for column correlation for compression and proposes the same compression schemes under different names with the same compression ratio.Virtual [171] implements Corra in Python on top of Parquet. Expression encoding extends and integrates the concepts of White-box Compression, PIDS, and C3 by proposing a unified framework that allows future file formats to fully leverage MCC schemes.

6.6 Discussion

In this section, we discuss two layout strategies – Unified Transposed Layout within a vector and Segmented Page Layout within a page – as these are fundamental decisions for future file formats.

Unified Transposed Layout. We use the Unified Transposed Layout (UTL) [140] as an option rather than as the default. Although this layout enables complete data-parallelism for FastLanes-RLE and Delta schemes, its effect to permute the order of the tuples in a vector may sometimes not be desirable – though the original order can always be restored, this comes at an overhead. However, we argue that the substantial compression ratios achieved by FastLanes-RLE and DELTA make these schemes essential, making the UTL a valuable option for efficient data-parallelized decoding [140] in contexts where high compression ratios are a priority.
We address a common point of confusion regarding the applicability of the Unified Transposed Layout to variable-sized data, such as String or List. We find it can be used without problem, as variable-sized data are always accompanied by an offset array – a *fixed-size* vector of 1024 values to which we apply the UTL.

Vectorized Page Layout. An alternative to the segmented page layout for storing the result of an expression in a file is the vectorized page layout, where all encoded data of an expression for a vector are stored sequentially in one place. We have chosen the segmented page layout over the vectorized page layout for three main reasons: Supporting structs in a vectorized page layout can lead to filling the cache with unnecessary data during reading, particularly when only subfields of a struct are required. The segmented page layout allows for additional query optimizations by enabling query engines to access relevant data in a single location, such as the bases in FFOR, which effectively serve as vector-based zone maps [229], specifically the minimums of each vector. Collecting data with similar properties in one segment allows for further compression in a single pass. Although we currently avoid compressing these segments, having this option remains beneficial for compression-sensitive workloads where achieving a high compression ratio is a priority.

6.7 CONCLUSION

Popular big data file formats only partially benefit from the full compression potential of Light-Weight Compression (LWC) schemes [44, 48], missing opportunities for compressed execution, cascaded compression and multi-column compression. The latter two issues affect compression ratio and make the use of Heavy-Weight Compression (HWC) methods necessary, even though these are SIMD and GPU unfriendly. This is why FastLanes introduces `Expression Encoding`, paired with a intricate segmented page design, that enables fine-grained and efficient decoding of cascading LWC schemes.

With this chapter, we release a high-quality open-source C++ implementation of FastLanes v0.1. Designing a data format requires a lot of effort and getting many details right. We think this release is a major contribution.

Our evaluation of FastLanes versus Parquet, BtrBlocks and the DuckDB format shows that HWCs can now be avoided without sacrificing any compression ratio, and very significantly improving decoding speed; while offering efficient fine-grained data access as well as novel opportunities for compressed execution.

7

Conclusion & Future Work

7.1 Contributions

Let us briefly recap the research questions posed in Section 1.1.

- **Research Question 1:** *How can SIMD instructions be leveraged to accelerate the decompression of LWCs?*
- **Research Question 2:** *Can data-parallel encodings be implemented in the most maintainable way—namely, scalar code with auto-vectorization—by relying on compilers to generate SIMD instructions, while still achieving maximum performance?*
- **Research Question 3:** *Is it possible to design a data-parallel encoding for floating-point numbers that uses SIMD instructions to decode many values in parallel while achieving a compression ratio comparable to heavyweight compressors (HWCs)?*
- **Research Question 4:** *Do data-parallelized encodings, originally tailored for CPUs, remain efficient on GPUs? What is their impact when integrated into query execution engines on GPUs, and can they be further optimized?*
- **Research Question 5:** *How should LWCs be implemented on GPUs? What should their API look like?*
- **Research Question 6:** *Can data-parallel lightweight encodings be used to achieve better compression ratios than heavyweight compressors (HWCs) while maintaining the key advantages of lightweight encodings, such as support for compressed execution, fast decoding, and vectorized processing?*
- **Research Question 7:** ***What could a file format built on the ideas from this thesis look like?***

The contributions addressing these questions are summarized below.

The Novel 1024-bit Interleaved Layout. We designed and implemented a novel 1024-bit interleaved layout for bit-packing. The core idea is to reorganize the layout of bit-packed data such that, during bit-unpacking, each value is already placed in its correct SIMD lane. This is achieved by distributing the bit-packed values across lanes rather than storing them sequentially. The layout enables efficient bit-unpacking by leveraging the SIMD capabilities of any architecture and instruction set (ISA), while relying solely on scalar code that can be fully auto-vectorized by modern compilers. We show that this approach achieves performance on par with explicit SIMD implementations and is overall faster than any existing bit-packed layout.

The Novel Unified Transposed Layout. We designed and implemented the Novel Unified Transposed Layout to address the data-dependency challenge of delta encoding. Our layout applies a two-step strategy. First, it transposes the values to ensure that data with mutual dependencies are placed within the same SIMD lane, rather than being spread across lanes. This eliminates inter-lane dependencies and allows each SIMD lane to operate independently. Second, it unifies the transposed layout across data types of varying widths (from 8-bit to 64-bit). Since wider types, such as 64-bit values, require fewer SIMD lanes to fill a register (e.g., 8×64-bit vs. 64×8-bit for AVX-512), we designed a single consistent transposition pattern that works efficiently across all types. We show that this approach achieves performance on par with explicit SIMD implementations and significantly outperforms all existing delta encoding layouts.

The Novel FastLanes RLE. We designed FastLanes RLE, a novel variant of Run-Length Encoding that resolves the inherent data dependency challenges of traditional RLE. Our approach conceptually maps RLE to a form of dictionary encoding, where duplicate values are retained in the dictionary, and the input is encoded using index codes that reference this dictionary. A key property of this design is that the indices increase only when a new run begins, making them highly compressible using delta encoding—since the maximum delta is always 1. Our implementation demonstrates that FastLanes RLE significantly outperforms existing SIMD-optimized RLE solutions and achieves performance on par with explicit SIMD implementations.

The FastLanes Frame-of-Reference (FFOR). We designed and implemented the FFOR primitive, which fully optimizes the classic Frame-of-Reference (FOR) encoding. While traditional FOR implementations typically require two separate function calls—one for bit-unpacking and another for adding the base—FastLanes FFOR fuses these steps into a single function. It adds the base directly to the unpacked value within the generated code. This implementation allows the addition to occur while the data remains in registers, eliminating one load and one store instruction—both of which are common bottlenecks in high-performance, fully data-parallel implementations such as those in FastLanes. Our evaluation shows that FFOR significantly outperforms existing SIMD-optimized FOR implementations and matches the performance of explicit SIMD code.
These four contributions—1) the Novel 1024-bit Interleaved Layout, 2) the Novel Unified Transposed Layout, 3) the Novel FastLanes RLE, and 4) the FastLanes Frame-of-Reference (FFOR)—emerged from our efforts to answer **Research Questions 1 and 2**.

We conclude that the answer to **Research Question 1** is: lightweight compression schemes (LWCs) can be significantly optimized using SIMD instructions. This is made possible by carefully redesigning the decoding logic with hardware-awareness in mind, specifically by: (1) eliminating branches to ensure consistent control flow, (2) structuring the data layout to exploit repetitive patterns, (3) using sequential access patterns to ensure data is already in cache and not bottlenecked by cache-to-RAM latency, (4) providing enough data to fully saturate the available parallelism, and (5) eliminating dependencies across SIMD lanes. The achieved acceleration is independent of the ISA and works on both x86 architectures—with SSE, AVX, and AVX-512—and ARM architectures—with NEON and SVE SIMD instructions.
We conclude that the answer to **Research Question 2** is **yes**: lightweight compression schemes (LWCs) can be implemented in a highly maintainable way using scalar code, relying on compiler auto-vectorization. This is achievable through a careful redesign of the encoding layout to ensure that the decoding logic remains simple and free of control-flow complexity, which can inhibit vectorization. Furthermore, our design avoids relying on AVX-512-specific instructions that, while beneficial on supported CPUs, introduce significant performance cliffs on architectures without AVX-512 support.

Discussion on what programming paradigm we should choose for SIMD-friendly schemes. Out of the four paradigms described in Section 2—scalar code with auto-vectorization, compiler intrinsics, third-party libraries, and explicit SIMD intrinsics—we have implemented and benchmarked only the scalar code with auto-vectorization and the explicit SIMD versions. The performance of the scalar code proved so convincing—matching the speed of explicit SIMD, the most performant paradigm—while offering superior readability and maintainability, that we concluded the other two approaches were not even necessary to benchmark for FastLanes up to version v0.1. However, both compiler intrinsics and third-party libraries remain promising avenues for future work, particularly in cases where our current approach falls short. This could especially apply to more complex compression schemes.

Discussion on How to Generate Scalar Code. One topic we did not address in the design and implementation of SIMD-friendly encoding is the alternative to using Python scripts for generating highly efficient code: using template programming to let the compiler generate these functions. Both options have their pros and cons. Generating code with Python scripts is harder to debug, as it requires extra steps to validate the generated code, while template programming keeps everything within C++, allowing us to use all the features of IDEs to inspect and check the syntax of the code. On the other hand, generated code compiles faster, whereas templates force the compiler to generate these functions on every new build. We argue—and ultimately chose—to use Python scripts, since the code only needs to be generated once and can be reused many times, avoiding repeated overhead during compilation. With a bit of extra effort maintaining the Python generator, we save compiler time on every new build.

ALP. To answer **Research Question 3**, we designed and implemented *ALP*, a novel lightweight compression scheme for floating-point data. ALP adapts to the two prevalent types of floating-point data: decimal and high-precision. For decimal data, it maps values to integers and applies FastLanes FFOR to fully data-parallelize the encoding. For

high-precision data, ALP focuses on compressing the leftmost bits, as the rightmost bits are essential for maintaining precision. It leverages our 8-entry dictionary encoding to compress the left bits efficiently. ALP achieves compression ratios comparable to or better than heavyweight compression schemes, while being an order of magnitude faster.
We conclude that the answer to **Research Question 3** is **yes**: although the representation of floating-point numbers is complex, we observe that—at the granularity of vectors (i.e., batches of 1024 values)—floating-point data often exhibits exploitable patterns for compression. In particular, when the data represents decimal-like values, most values fall within a narrow range; and for high-precision data, the leftmost bits tend to come from a small domain. These insights were key to the design of ALP, which leverages this structure to enable efficient, data-parallel compression of floating-point values.

Discussion of ALP for High-Precision Data. While ALP is one of the few schemes capable of achieving any compression on high-precision floating-point values, as shown in Chapter 3, the results are often far from desirable. Saving just a few bits per 64- or 32-bit value typically has minimal impact—especially when weighed against the added complexity of encoding and decoding, particularly on GPUs [176]. In machine learning pipelines, high-precision values often result from feature engineering or system defaults rather than a strict need for such granularity [230]. Moreover, for many ML models—such as decision trees and neural networks—reducing precision (e.g., from float64 to float32 or even quantized formats) has little effect on predictive performance [231, 232].
This raises the question: why not compress them using a lossy approach? One option is to apply quantization to high-precision floating-point values [233, 234]. Techniques such as Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) can retain model accuracy while significantly reducing memory and compute costs [235, 236]. Another approach is to adopt recent work from CWI, such as LEP [237]—a lossy variant of ALP. Compared to linear quantization, LEP tunes parameters per vector rather than globally and filters out outliers using exceptions, achieving significantly higher compression than ALP. While ALP remains a lightweight option when we **must** retain full precision, we acknowledge that in many cases, quantization or lossy floating-point compression is likely the more suitable choice.

Benchmarking Data-Parallel Encodings on GPU. We benchmarked FastLanes data-parallel lightweight compression schemes (LWCs)—originally designed for CPUs—on GPUs by mapping the 32-way parallelism of FastLanes (for `int32` data types) such that each SIMD lane is assigned to a single thread. We observed that data-parallel encodings naturally lead to coalesced memory accesses, which are essential for maximizing throughput on GPUs. Compared to tile-based decoding [238], our approach yielded superior performance. We further analyzed the performance of FastLanes layouts when scanning multiple columns and found a sharp performance drop when more than one column was involved. This was due to excessive shared memory allocation per thread, leading to significantly reduced occupancy. To address this, we experimented with lowering the number of values decoded per thread from 32 to 16, 8, and eventually 4. The results showed that reducing per-thread memory pressure increased occupancy and improved performance.
We conclude that the answer to **Research Question 4** is **yes**: data-parallel layouts are more efficient than non-data-parallel approaches, primarily due to their memory access patterns.

However, these layouts do not perform well when scanning multiple columns simultaneously, as the memory pressure becomes significant—each column requires 1024 values to be materialized, which quickly exhausts shared memory or GPU registers. This limitation can be addressed by making the decoding API more fine-grained, reducing the number of values decoded per thread, thus increasing occupancy and enabling better performance in multi-column query scenarios.

G-ALP. We designed and implemented **G-ALP**, the GPU-optimized version of ALP, incorporating two key optimizations. First, we data-parallelized the exception patching phase by sorting exceptions—i.e., values that cannot be mapped to integers—by thread ID, allowing each thread to directly access the exceptions for the lane it is responsible for, without traversing the entire exception list. Second, we redesigned the decoding API to return a single value per thread, instead of decoding 32 values at once. This significantly reduces shared memory pressure and improves occupancy. G-ALP demonstrates superior performance compared to both the naive implementation of ALP and the compression schemes in `nvCOMP`, the state-of-the-art GPU compression library from industry.

Lessons Learned from Designing and Implementing GALP. GALP is the story of what ultimately worked. We experimented with many different configurations and implementations of G-ALP, which are further documented in [176]. From working with GPUs, we learned that optimizing an encoding scheme for the GPU is significantly more difficult and time-consuming than on the CPU—but it is achievable. As a result, a fully GPU-based reader for FastLanes was not completed within the scope of this thesis and may require at least one more PhD thesis to fully mature, given how sensitive and complex GPU optimization can be. Nevertheless, it remains a highly interesting and promising direction for future work.
We conclude that the answer to **Research Question 5** is **yes**: lightweight compression schemes (LWCs) on GPUs must be *fully data-parallel*—even in their smallest components—because any sequential operation can quickly become the performance bottleneck. Furthermore, their APIs must support and be efficient for a *one-value-per-thread* execution model to minimize memory pressure and maximize occupancy.

Expression Encodings. We designed and implemented the *Expression Encoding* compression model, which combines our fully data-parallelized lightweight encodings (LWCs) and represents each combination using a compact syntax that is interpreted at runtime—once per row group, and therefore amortized. This syntax can express intra-column encoding pipelines, inter-column relationships, cascaded encodings, and even whitebox compression strategies. Expression Encoding achieves a better compression ratio than the Parquet file format with Zstd, as demonstrated by compressing samples from the `public_bi` dataset. Additionally, we propose a novel compression API that delivers compressed vectors directly to the query execution engine.

Lessons Learned from the Implementation of the Hard-Coded Variant of Expression Encoding. We initially implemented Expression encoding in a fully hardcoded manner, meaning that different combinations were explicitly represented in the code without any interpretation layer. While this approach worked for single-column encoding, its static nature proved inadequate for MCC, where dynamic information—such as references to

correlated columns, which can be of any number—needs to be encoded in the file format and cannot be hardcoded.
From this experience, we learned the importance of flexibility in Expression encoding and reimplemented it with a more dynamic, fully interpretable design to make it future-proof and adaptable to more complex use cases like MCC. Thanks to this interpretive layer, we believe we can now more easily adapt to future advancements in the compression field by supporting additional code paths and tokens.

Segmented Page Layout. To support Expression Encoding in the storage level, we introduce a new page layout capable of storing the encoded output of each expression in a vectorized manner—meaning the query engine can decode or encode each batch independently. This is achieved using an offset array that points to the encoded data of each vector. The design is aligned with the vectorized execution paradigm, where scan operators process data in vectors.
The above two contributions—1) *Expression Encodings* and 2) *Segmented Page Layout*—emerged from our efforts to answer **Research Questions 5 and 6**.
We conclude that the answer to **Research Question 6** is **yes**: by compressing small and specific opportunities in simple ways—rather than using overgeneralized schemes that attempt to capture all patterns with a single, complex decoding process, as is common in heavyweight compression—it is indeed possible to achieve the best of both worlds: highly compressed data with fast decoding, and data that remains executable in its encoded form.

FastLanes File Format. We designed and implemented the FastLanes File Format from scratch, with expression encoding, segmented page layout, and data-parallelized encodings at its core. It brings together all the research we have done so far into a single specification—a major step toward the features of next-generation file formats.
We conclude that the answer to **Research Question 7** is: future file formats should be designed with the following principles in mind:

1. They should be composed of fully data-parallel encodings that are efficient on both CPUs and GPUs.

2. They should adopt *Expression Encoding* to support cascaded encodings, multi-column compression (MCC), and white-box compression within a unified compression model.

3. They should employ a *Segmented Page Layout* to encode and decode small batches of data independently, enabling vectorized decoding and avoiding the limitations of block-based formats.

4. They should provide flexible decoding APIs—an arbitrary compressed vector API for CPUs and a one-value-per-thread API for GPUs.

Lessons Learned from the Design and Implementation of FastLanes. We started the design of FastLanes as a simple master's thesis project, but it took nine months just to design and benchmark SIMDized bit-packing and an early version of data-parallel delta encoding. Soon, we realized there was much more to optimize, which led to the development of this thesis—ultimately taking more than four years. We learned that designing a comprehensive

file format is an exhaustive task, and we hope this work serves as a foundation for continued research and development in this area.

Open Source. Our final—and perhaps most impactful—contribution is that we open-sourced all our code and implementations at `https://github.com/cwida/FastLanes` and `https://github.com/cwida/ALP`. From the very beginning, guided by the engineering mindset fostered at CWI, we placed strong emphasis on code quality and followed best software development practices—an approach often overlooked in academic research. This high standard of implementation enabled ALP to be integrated into DuckDB with minimal modifications and contributed to it winning the Best Paper Award for Reproducibility.

7.2 Future Work

In this section, we outline several key areas for future work that could extend and enhance the FastLanes file format.

Adoption. The biggest question for the future of FastLanes is how to replace Parquet so that all OLAP databases can benefit from FastLanes' innovations. This task is extremely difficult—multiple new file formats have attempted and failed to achieve it, most notably ORC, as well as newer versions of Parquet itself. We believe the core reason lies in the widespread adoption of Parquet version 1, coupled with the fact that many databases have heavily optimized and deeply integrated their own implementations of Parquet into their execution engines—each with its own custom reader and writer. For these systems, supporting and implementing additional file formats is extremely costly, often requiring large teams of engineers to maintain.
This fragmentation stems from Parquet's decision to provide only a specification rather than a single reference implementation. In this light, having a single shared implementation sounds promising as a way to encourage adoption of FastLanes—but it comes with its own set of challenges. The most significant is that OLAP databases are implemented in a variety of programming languages—ranging from Rust and Go to C++ and others. As a result, the obvious question becomes: how can we adopt FastLanes across a wide range of engines, each written in a different language?

Machine Learning Data. Several file formats have emerged to address the needs of machine learning data workloads, including Bullion [160], which tackles the complexities of data compliance, optimizes the encoding of long-sequence sparse features, and efficiently manages wide-table projections. Nimble [239], a columnar file format developed by Meta, is designed for very wide tables commonly found in machine learning training datasets. LanceDB [191], another columnar data format, is optimized for machine learning workloads, offering high-performance random access and efficient handling of complex data types, including images and videos.
However, these file formats lack some of the key advantages that FastLanes provides—namely, achieving the highest compression ratios while delivering the fastest decoding times. This raises the following question: how can we extend FastLanes to support nested data types, as well as wide and sparse data—characteristic of machine learning workloads—while maintaining its OLAP-optimized core? More specifically, can we support

both OLAP and ML workloads in a single, unified format?

File Footer. For simplicity, we initially used JSON for the FastLanes v0.1 footer. While functional, this approach introduces unnecessary parsing and storage overhead. These drawbacks become increasingly problematic as data size and metadata grow.
To address these limitations, we are exploring alternative solutions such as FlatBuffers or using the FastLanes file format itself to store the footer. FlatBuffers offer an efficient binary representation, though they lack native support for compression. The FastLanes format, however, includes built-in compression, which becomes important if the footer size grows significantly. Using FastLanes to encode the footer would also eliminate an external dependency.
It is also worth noting that Parquet uses Apache Thrift to encode its metadata, which serializes the entire metadata structure in one go. This design becomes inefficient when only a small subset of metadata is needed—especially in AI workloads where tables may contain as many as 17,000 columns [160], leading to the decoding of metadata for many unnecessary columns.
Therefore, this remains an open area for future work: what is the most efficient way to represent metadata in the FastLanes file format—and, more broadly, in the next generation of big data file formats?
Expression Encoding on GPU. The state-of-the-art encoding model on GPU, tile-based decoding [240], proposes decoding data in small batches—called tiles—within a GPU's shared memory to avoid transferring data back to global memory, which is a primary bottleneck in GPU performance. Additionally, it supports cascaded encoding limited to `FOR`, `DELTA`, and `RLE`, with both the value and length arrays further bit-packed. `Expression Encoding` aligns with the concept of decoding a batch of data that fits in shared memory, while offering more cascaded combinations capable of achieving better compression ratios than `Zstd`, and supporting compressed tiles similar to compressed vectors in DuckDB [51] and Velox [50].
We speculate that a CUDA implementation of FastLanes could bring significantly higher decoding speeds and improved compression ratios to the GPU processing ecosystem. Initial results look promising [165]; however, as we learned from G-ALP—even optimizing an already data-parallel encoding on the CPU is challenging on the GPU—implementing expression encoding on GPU requires substantial effort and may involve novel techniques and new design decisions. This raises a clear question for the future development of FastLanes: how can we implement expression encoding efficiently on GPU?

Schema Evolution. Future file formats should support schema evolution. Parquet currently offers a limited form of schema evolution for changing types within a column [241], allowing only the promotion of a few specific types, rather than broader support for other data types. We believe that `Expression Encoding` enables file formats to support this feature seamlessly, thanks to the type information included in each expression. Storing the footer in a separate place allows to include new types and new columns by modifying, resp. generating new expressions in the footer, without having to rewrite existing rowgroup data.

Encryption. We believe that our proposed `Expression Encoding` is a perfect match for supporting encryption in a vectorized manner, in contrast to encryption in Parquet, which is block-based. In our vision, encryption becomes just one more operator at the end of an

expression—responsible for [en/de]crypting the compressed vector of an expression. This remains an open area for future work: how effective is vectorized encryption in practice, and how well does it integrate with expression encoding?

Tunable Rowgroup Size. The rowgroup size in FastLanes is always a multiple of 1024 records. We initially set the rowgroup size to 64×1024 vectors. However, we envision rowgroup size as a tunable parameter that can be adjusted to balance compression ratio, memory footprint, and metadata size. For example, if a rowgroup contains nested data types—such as lists with an average length of 10, which is common [242]—and FSST is used for strings within these lists, the FSST symbol table becomes 10 times less likely to capture sub-patterns effectively due to exposure to a higher diversity of strings. Therefore, we leave this as future work: what is the best rowgroup size, and how can it be determined?

Nested Data Types Nested data types—such as structs, lists, and maps—are widely used and natively supported by open big data file formats like Apache Parquet. Recent work at CWI on real-world JSON datasets [243] suggests that applying LWC schemes to flattened nested data types, which resemble columns, is less effective for compression than using HWC schemes. However, new nesting-specific encodings [242], such as "list dictionary" encoding—which assigns a single code to an entire list of values, rather than to individual values within the list—could significantly reduce this gap.
This leaves us with an open question: how can we support nesting-specific encodings within FastLanes' expression encoding compression model?

Predicate Pushdown Recent work at CWI proposes data-parallel predicate evaluation using bitmaps as the underlying data structure, rather than traditional selection vectors [244]. This approach enables faster intersection operations between selection structures by reducing them to a few SIMD bitwise `AND` operations on bitmap datasets—for example, two AVX-512 instructions. The work also explores how this technique can be applied directly to LWC-encoded data without requiring full decompression.
We clearly envision supporting this model in FastLanes and further investigating how to extend it to compressed vectors of expression-encoded data—rather than just LWC-encoded data. Therefore, we raise the following question as future work: how can bitmap-based, data-parallel predicate evaluation be integrated with the more sophisticated expression encoding pipeline in FastLanes, where data flows through multiple layers of encodings?

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

[1] Apache Software Foundation. Apache parquet. `http://parquet.apache.org/`, 2023. Version 1.12.3 or later.

[2] J. A. N. Lee. *The History of Computing*. Springer, 2008.

[3] C. G. Bell. The evolution of digital computers. *Scientific American*, 1970.

[4] *American National Standard Code for Information Interchange (ASCII)*. American National Standards Institute (ANSI), 1963.

[5] IBM. The ibm personal computer: Launch announcement (1981), 1981. Accessed April 2025.

[6] Paul E. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, MA, 2nd edition, 2003.

[7] Martin Campbell-Kelly and William Aspray. *Computer: A History of the Information Machine*. Westview Press, Boulder, CO, 2nd edition, 2004.

[8] dBASE LLC. A brief history of dbase, 2003. Accessed April 2025.

[9] Excel Dimensions. Microsoft excel's history from 1982 until today, 2022. Accessed April 2025.

[10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 specification. *World Wide Web Consortium (W3C)*, 1998.

[11] Douglas Crockford. The application/json media type for javascript object notation (json). *Internet Engineering Task Force (IETF)*, 2006.

[12] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale xml dissemination service. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 612–623. VLDB Endowment, 2004.

[13] Michael Stonebraker and Andrew Pavlo. What goes around comes around... and around... *SIGMOD Rec.*, 53(2):21–37, July 2024.

[14] Apache Software Foundation. Apache avro. https://avro.apache.org, 2009. Version 1.0 released in 2009.

[15] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 1199–1208, Hannover, Germany, 2011. IEEE Computer Society.

[16] Apache. Apache orc, 2023. https://orc.apache.org/.

[17] J. Dean and S. Ghemawat. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[18] Wikipedia contributors. Moore's law. https://en.wikipedia.org/wiki/Moore%27s_law, 2025. Accessed: 2025-04-11.

[19] Bingsheng He, Ruoyuan Fang, et al. Relational query co-processing on graphics processors. In *VLDB*, 2009.

[20] You Wu et al. Design and evaluation of an efficient gpu-aware storage system. In *SIGMOD*, 2014.

[21] Sebastian Breß et al. Gpu-accelerated database systems: Survey and open challenges. In *SIGMOD*, 2018.

[22] OpenJDK. Jep 338: Vector api (incubator), 2020. Introduced in Java 16, this JEP adds an incubating API for expressing vector computations that reliably compile to SIMD instructions at runtime. Accessed: 2025-04-09.

[23] JDK-6340864: Implement vectorization optimizations in HotSpot Server Compiler. Oracle Java Bug Database, 2012. Introduced HotSpot auto-vectorization based on SLP (Larsen & Amarasinghe). Resolved in JDK 7u40 (2013) and JDK 8.

[24] Azim Afroozeh. The limitations of parquet in simd and parallel processing. Master's thesis, Vrije Universiteit Amsterdam, 2020.

[25] Apache Software Foundation. Apache parquet format documentation - version 2. https://github.com/apache/parquet-format, 2018. Accessed: 2025-04-11.

[26] Apache Software Foundation. Apache parquet 2.4.0 release notes. https://github.com/apache/parquet-format/blob/master/CHANGES.md#version-240, 2017. Accessed: 2025-04-11.

[27] Apache Parquet Developer Mailing List. Discussion thread: Proposal for parquet 3.0. https://lists.apache.org/thread/5jyhzkwyrjk9z52g0b49g31ygnz73gxo, 2024. Accessed: 2025-04-11.

[28] Grafana Labs. Grafana tempo 2.3 release notes. https://grafana.com/docs/tempo/latest/release-notes/v2-3/, 2024. Accessed: 2025-04-11.

[29] Xiao Zeng and Others. Optimizing data formats for modern cpu architectures. *Proceedings of the VLDB Endowment*, 17:148–160, 2023.

[30] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[31] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1617–1632, 2020.

[32] T Glass. C3: Compressing correlated columns. Master's thesis, centrum wiskunde & informatica, 2023.

[33] Bogdan Guita, Diego Tomé, and Peter Boncz. White-box compression: Learning and exploiting compact table representations. 01 2020.

[34] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, 2006.

[35] Marcin Zukowski, Sándor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 59–59, Atlanta, GA, USA, 2006. IEEE Computer Society.

[36] xtensor team. Xsimd: C++ wrappers for simd intrinsics, 2024. https://github.com/xtensor-stack/xsimd.

[37] LLVM Project. Auto-vectorization in llvm, 2023. https://llvm.org/docs/Vectorizers.html.

[38] GNU Compiler Collection. Gcc auto-vectorization, 2023. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[39] Intel Corporation. Intel intrinsics guide, 2024. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[40] Agner Fog. Optimizing software in c++: An optimization guide for windows, linux and mac platforms, 2022. https://www.agner.org/optimize/.

[41] CWI DA. Public bi benchmark, 2025.

[42] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.

[43] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. Elf: Erasing-based lossless floating-point compression. *Proceedings of the VLDB Endowment*, 16(7):1763–1774, 2023.

[44] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[45] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[46] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 370–379, Orlando, FL, USA, 1998. IEEE Computer Society.

[47] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 59–70, Atlanta, GA, USA, April 2006. IEEE.

[48] A Afroozeh. Towards a new file format for big data: Simd-friendly composable compression. Master's thesis, centrum wiskunde & informatica, 2020.

[49] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. Procella: Unifying serving and analytical data at youtube. *PVLDB*, 12(12):2022–2034, 2019.

[50] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment*, 15(12):3372–3384, 2022.

[51] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam, Netherlands, 2019. ACM.

[52] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. Hardware-oblivious SIMD parallelism for in-memory column-stores. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[53] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 317–326, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45, 01 2015.

[55] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. Vectorized vbyte decoding. *ArXiv*, 2015.

[56] Wayne Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-yun Nie, Hongfei Yan, and Ji-Rong Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems*, 33, 02 2015.

[57] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using SIMD instructions. In *Proceedings of the 6th International Workshop on Data Management on New Hardware (DaMoN)*, pages 34–40, Indianapolis, IN, USA, June 2010. ACM.

[58] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. Conflict detection-based run-length encoding: AVX-512 CD instruction set in action. In *Proceedings of the 34th IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 96–101, Paris, France, April 2018. IEEE.

[59] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. Beyond straightforward vectorization of lightweight data compression algorithms for larger vector sizes. In *Grundlagen von Datenbanken*, 2018.

[60] Florian Lemaitre, Arthur Hennequin, and Lionel Lacassagne. How to speed connected component labeling up with simd rle algorithms. In *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'20, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, 2018.

[62] Dirk Habich, Patrick Damme, Annett Ungethüm, and Wolfgang Lehner. 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*, DBTest'18, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, 2017.

[64] Richard Michael Grantham Wesley and Pawel Terlecki. Leveraging compression in the tableau data engine. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 563–573, New York, NY, USA, 2014. Association for Computing Machinery.

[65] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM scalable vector extension. *CoRR*, abs/1803.06185, 2018.

[66] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. Make the most out of your simd investments: counter control flow divergence in compiled query pipelines. *The VLDB Journal*, 29(2):757–774, May 2020.

[67] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 145–156, New York, NY, USA, 2002. Association for Computing Machinery.

[68] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009.

[69] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 289–300, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelope of main memory data processing with a new storage layout. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 31–46, Melbourne, Victoria, Australia, 2015. ACM.

[71] Orestis Polychroniou and Kenneth A. Ross. Efficient lightweight compression alongside fast scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Wee Keong Ng and Chinya V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Trans. on Knowl. and Data Eng.*, 9(2):314–328, March 1997.

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

[74] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, dec 1986.

[75] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 46(6):723–749, jun 2016.

[76] Guy E. Blelloch. Prefix sums and their applications. *Carnegie Mellon University, Kilthub Repository*, May 2004. Originally published as CMU-CS-90-190 in 1990.

[77] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD*, pages 671–682. ACM, 2006.

[78] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *ACM SIGMOD*, pages 1493–1508. ACM, 2015.

[79] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 311–326, New York, NY, USA, 2016. Association for Computing Machinery.

[80] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, sep 2000.

[81] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1390–1403, New York, NY, USA, 2022. Association for Computing Machinery.

[82] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 858–869. VLDB Endowment, 2006.

[83] Adnan Alhomssi Maximilian Kuschewski, David Sauerwein and Viktor Leis. Btrblocks: Efficient columnar compression for data lakes. In *Proceedings of the 2023 ACM SIGMOD international conference on Management of data*. Association for Computing Machinery, 2023. In press. Accessed on: 2023-04-13.

[84] Azim Afroozeh and Peter Boncz. The fastlanes compression layout: Decoding > 100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, jul 2023.

[85] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006.

[86] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.

[87] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[88] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998.

[89] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869. Citeseer, 2006.

[90] Mark A Roth and Scott J Van Horn. Database compression. *ACM Sigmod Record*, 22(3):31–39, 1993.

[91] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[92] Yann Collet. Zstandard - fast real-time compression algorithm, 2015. Accesed on: 2023-04-13.

[93] Yann Collet. Lz4 - extremely fast compression, 2014. Accesed on: 2023-04-13.

[94] Seungyeon Lee, Jusuk Lee, Yongmin Kim, Kicheol Park, Jiman Hong, and Junyoung Heo. Efficient scheme for compressing and transferring data in hadoop clusters. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1256–1263, 2020.

[95] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[96] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment*, 15(11):3058–3070, 2022.

[97] DuckDB Labs. Patas compression: Variation on chimp, 2022. Accessed on: 2023-04-13.

[98] Boudewijn Braams. Predicate pushdown in parquet and apache spark. *MSc thesis*, 2018.

[99] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[100] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. Beyond straightforward vectorization of lightweight data compression algorithms for larger vector sizes. In *Grundlagen von Datenbanken*, pages 71–76, 2018.

[101] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[102] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016.

[103] Public bi benchmark. `https://github.com/cwida/public_bi_benchmark`, 2019. Accessed on: 2023-04-13.

[104] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In *Proceedings of the 2018 Workshop on Testing Database Systems (DBTest 2018)*, pages 1–6, Houston, TX, USA, 2018. Association for Computing Machinery.

[105] National Ecological Observatory Network (NEON). Barometric pressure (dp1.00004.001), 2021.

[106] National Ecological Observatory Network (NEON). Relative humidity above water on-buoy (dp1.20271.001), 2021.

[107] National Ecological Observatory Network (NEON). Ir biological temperature (dp1.00005.001), 2021.

[108] National Ecological Observatory Network (NEON). Dust and particulate size distribution (dp1.00017.001), 2021.

[109] National Ecological Observatory Network (NEON). 2d wind speed and direction (dp1.00001.001), 2021.

[110] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142, Snowbird, Utah, USA, 2006. IEEE, IEEE.

[111] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE transactions on computers*, 58(1):18–31, 2008.

[112] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[113] Fastlanes, 2023. Accesed on: 2023-04-13.

[114] Azim Afroozeh and P Boncz. Towards a new file format for big data: Simd-friendly composable compression, 2020.

[115] Mark Raasveldt and Hannes Muehleisen. Duckdb, 2019. Accesed on: 2023-04-13.

[116] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

[117] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. *CoRR*, abs/2104.14294, 2021.

[118] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. `https://openai.com/research/language-unsupervised`, 2019. OpenAI Blog.

[119] Vipul Raheja, Dhruv Kumar, Ryan Koo, and Dongyeop Kang. Coedit: Text editing by task-specific instruction tuning. *arXiv preprint arXiv:2305.09857*, May 2023.

[120] Vadim Engelson, Peter Fritzson, and Dag Fritzson. Lossless compression of high-volume numerical data from simulations, 2000.

[121] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[122] Nathaniel Fout and Kwan-Liu Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[123] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[124] Andrea Bruno, Franco Maria Nardini, Giulio Ermanno Pibiri, Roberto Trani, and Rossano Venturini. Tsxor: A simple time series compression algorithm. In *String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 217–223, Lille, France, 2021. Springer.

[125] Aliaksandr Valialkin. Victoriametrics: achieving better compression than gorilla for time series data. `https://faun.pub/victoriametrics-achieving-better-compression-for-time-series-data-than-gor` 2019. Accesed on: 2023-04-13.

[126] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[127] John L Gustafson and Isaac T Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing frontiers and innovations*, 4(2):71–86, 2017.

[128] Azim Afroozeh and Peter Boncz. The fastlanes compression layout: Decoding> 100 billion integers per second with scalar code. *Proceedings of the VLDB Endowment*, 16(9):2132–2144, 2023.

[129] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. An empirical evaluation of columnar storage formats. *arXiv preprint arXiv:2304.05028*, 2023.

[130] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM SIGMOD*, 1(2):118:1–118:26, 2023.

[131] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 1617–1632, 2020.

[132] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. Gpu database systems characterization and optimization. *Proceedings of the VLDB Endowment*, 17(3):441–454, 2023.

[133] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

[134] Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Comput. Graph.*, 111:155–165, 2023.

[135] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[136] Anil Shanbhag, Bobbi W Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1390–1403, 2022.

[137] Eyal Rozenberg and Peter Boncz. Faster across the pcie bus: a gpu library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–5, 2017.

[138] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

[139] Azim Afroozeh. Fastlanes v0.1, 2025. Accessed: 2025-03-07.

[140] Azim Afroozeh and Peter Boncz. The fastlanes compression layout: Decoding > 100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, jul 2023.

[141] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. Alp: Adaptive lossless floating-point compression. *Proc. ACM Manag. Data*, 1(4), dec 2023.

[142] Azim Afroozeh, Lotte Felius, and Peter Boncz. Accelerating gpu data processing using fastlanes compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*, DaMoN '24, New York, NY, USA, 2024. Association for Computing Machinery.

[143] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE, 2008.

[144] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

[145] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the nvidia hopper gpu architecture, 2024.

[146] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking, 2019.

[147] NVIDIA. Cuda c++ programming guide, 2025. NVIDIA Documentation.

[148] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[149] NVIDIA. How to implement performance metrics in cuda c/c++, 2025. NVIDIA Documentation.

[150] NVIDIA. nvcomp benchmarks, 2025.

[151] NVIDIA. Thrust documentation, 2025. NVIDIA Documentation.

[152] NVIDIA. nvcomp v2.0.0 now available with new compressors, 2025. NVIDIA Developer Blog.

[153] NVIDIA. Optimizing data transfer using lossless compression with nvcomp, 2025. NVIDIA Developer Blog.

[154] NVIDIA. Using fully redesigned batch api and performance optimizations in nvcomp v2.1.0, 2025. NVIDIA Developer Blog.

[155] NVIDIA. Accelerating lossless gpu compression with new flexible interfaces in nvidia nvcomp, 2025. NVIDIA Developer Blog.

[156] NVIDIA. nvcomp: Gpu-accelerated compression library, 2025. Available on conda-forge.

[157] NVIDIA. Gdeflate: Gpu-optimized lossless compression, 2025. NVIDIA Documentation.

[158] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[159] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, 2021.

[160] Gang Liao, Ye Liu, Jianjun Chen, and Daniel J. Abadi. Bullion: A column store for machine learning. arXiv preprint arXiv:2404.08901, 2024.

[161] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. An empirical evaluation of columnar storage formats, 2023.

[162] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. A deep dive into common open formats for analytical dbmss. *Proc. VLDB Endow.*, 16(11):3044–3056, aug 2023.

[163] Google. Snappy - a fast compressor/decompressor, 2014. Accesed on: 2023-12-04.

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

[165] Azim Afroozeh and Peter Boncz. Fastlanes on gpu: Analysing data-parallelized compression schemes. In -, pages –, -, 2023. -.

[166] Azim Afroozeh. Fastlanes end-to-end script, 2024. Accessed: 2024-11-29.

[167] Azim Afroozeh and Peter Boncz. Fastlanes: A simd-friendly composable compression library. In -, pages –, -, 2021. DBDBD.

[168] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1–2):670–680, sep 2010.

[169] Xi Lyu, Andreas Kipf, Pascal Pfeil, Dominik Horn, Jana Giceva, and Tim Kraska. Corbit: Leveraging correlations for compressing bitmap indexes. In *Proceedings of the Fifth International Workshop on Applied AI for Database Systems and Applications (AIDB 2023)*, volume 3462 of *CEUR Workshop Proceedings*, pages 1–10, Vancouver, Canada, 2023. CEUR-WS.org.

[170] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. Corra: Correlation-aware column compression, 2024.

[171] Mihail Stoian, Alexander van Renen, Jan Kobiolka, Ping-Lin Kuo, Josif Grabocka, and Andreas Kipf. Lightweight correlation-aware table compression. In *NeurIPS 2024 Third Table Representation Learning Workshop*, 2024.

[172] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. White-box compression: Learning and exploiting compact table representations. In *Proceedings of the 2020 Conference on Innovative Data Systems Research (CIDR)*, page 23, Amsterdam, The Netherlands, 2020. Very Large Data Base Endowment.

[173] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J. Elmore. Pids: Attribute decomposition for improved compression and query performance in columnar storage. *Proc. VLDB Endow.*, 13(6):925–938, feb 2020.

[174] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proceedings of the 2005 Conference on Innovative Data Systems Research (CIDR)*, pages 225–237, Asilomar, CA, USA, 2005. Very Large Data Base Endowment.

[175] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. Procella: Unifying serving and analytical data at youtube. *Proc. VLDB Endow.*, 12(12):2022–2034, aug 2019.

[176] S Hepkema. Fastlanes on gpu. Master's thesis, centrum wiskunde & informatica, 2025.

[177] cwida. Fast static symbol table (fsst). `https://github.com/cwida/fsst`, 2023. Accessed: 2025-02-25.

[178] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. *Proceedings of the VLDB Endowment*, 27(3):169–180, September 2001.

[179] Marcin Zukowski, Niels Nes, and Peter Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, page 47–54, New York, NY, USA, 2008. Association for Computing Machinery.

[180] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, page 268–279, New York, NY, USA, 1985. Association for Computing Machinery.

[181] DuckDB. Announcing duckdb 1.20, February 2025. Accessed: 2025-02-25.

[182] DuckDB. Parquet encodings, January 2025. Accessed: 2025-02-25.

[183] Maximilian Kuschewski and contributors. Btrblocks: Efficient columnar compression for data lakes, 2023. GitHub repository, accessed on February 18, 2025.

[184] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[185] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*, volume 3. Now Publishers Inc., Cambridge, MA, USA, 2013.

[186] Mark Raasveldt. Lightweight compression in duckdb. `https://duckdb.org/2022/10/28/lightweight-compression.html`, 2022. Accesed on: 2023-04-13.

[187] Animesh Kr Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 561–574, Boston, MA, USA, 2018. USENIX Association.

[188] Apache. Apache carbondata, 2023. `https://carbondata.apache.org/`.

[189] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 311–326, New York, NY, USA, 2016. Association for Computing Machinery.

[190] Mosha Pasumansky. Inside Capacitor, BigQuery's Next-Generation Columnar Storage Format. `https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format`, 2023. Accessed: 2023-10-10.

[191] LanceDB Developers. Lancedb: A modern vector database. `https://github.com/lancedb/lancedb`, 2024. Accessed: 2024-11-29.

[192] Todor Ivanov and Matteo Pergolesi. The impact of columnar file formats on sql-on-hadoop engine performance: A study on orc and parquet. *Concurrency and Computation: Practice and Experience*, 32, 09 2019.

[193] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. A general simd-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst.*, 33:15:1–15:28, 2015.

[194] Andrew Trotman and Jimmy Lin. In vacuo and in situ evaluation of simd codecs. In *Proceedings of the 21st Australasian Document Computing Symposium*, ADCS '16, page 1–8, New York, NY, USA, 2016. Association for Computing Machinery.

[195] Daniel Lemire and Christoph Rupp. Upscaledb: Efficient integer-key compression in a key-value store using simd instructions. *Information Systems*, 66:13–23, 2017.

[196] Dirk Habich, Patrick Damme, Annett Ungethüm, and Wolfgang Lehner. 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*, DBTest'18, New York, NY, USA, 2018. Association for Computing Machinery.

[197] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, page 34–40, New York, NY, USA, 2010. Association for Computing Machinery.

[198] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 317–326, New York, NY, USA, 2011. Association for Computing Machinery.

[199] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009.

[200] Maximilian Ungethum and Example Another Author. A title of the paper. In *Proceedings of the 10th International Conference on Example Research (ICER 2018)*, pages 123–134, Berlin, Germany, 2018. Springer.

[201] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. Milc: Inverted list compression in memory. *Proc. VLDB Endow.*, 10(8):853–864, apr 2017.

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

[203] Florian Lemaitre, Arthur Hennequin, and Lionel Lacassagne. How to speed connected component labeling up with simd rle algorithms. In *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'20, New York, NY, USA, 2020. Association for Computing Machinery.

[204] Martin Boissier and Max Jendruk. Workload-driven and robust selection of compression schemes for column stores. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*, pages 674–677, Lisbon, Portugal, 2019. OpenProceedings.org.

[205] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. Good to the last bit: Data-driven encoding with codecdb. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 843–856, New York, NY, USA, 2021. Association for Computing Machinery.

[206] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, sep 1993.

[207] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*, pages 401–410, Madrid, Spain, 2009. Association for Computing Machinery.

[208] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 858–869, Seoul, Korea, 2006. Citeseer, VLDB Endowment.

[209] Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)*, pages 283–294, Athens, Greece, 2014. OpenProceedings.org.

[210] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. Leco: Lightweight compression via learning serial correlations, 2023.

[211] Jiujing Zhang, Zhitao Shen, Shiyu Yang, Lingkai Meng, Chuan Xiao, Wei Jia, Yue Li, Qinhui Sun, Wenjie Zhang, and Xuemin Lin. High-ratio compression for machine-generated data. *Proc. ACM Manag. Data*, 1(4), dec 2023.

[212] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. Compressdb: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1655–1669, New York, NY, USA, 2022. Association for Computing Machinery.

[213] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. Evaluating lightweight integer compression algorithms in column-oriented in-memory dbms. In *Proceedings of the 11th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2021)*, pages 26–36, Copenhagen, Denmark, 08 2021. ADMS Workshop Organizers.

[214] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proc. VLDB Endow.*, 13(12):2396–2410, jul 2020.

[215] Richard Michael Grantham Wesley and Pawel Terlecki. Leveraging compression in the tableau data engine. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 563–573, New York, NY, USA, 2014. Association for Computing Machinery.

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

[217] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 283–296, New York, NY, USA, 2009. Association for Computing Machinery.

[218] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, pages 40–56, Venice, Italy, 03 2017. Springer.

[219] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. Adaptive compression for databases. In *Proceedings of the 27th International Conference on Extending Database Technology (EDBT)*, pages 143–149, Paestum, Italy, 03 2024. OpenProceedings.org.

[220] Yinan Li, Jianan Lu, and Badrish Chandramouli. Selection pushdown in column stores using bit manipulation instructions. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[221] Martin Prammer and Jignesh M. Patel. Rethinking the encoding of integers for scans on skewed data. *Proc. ACM Manag. Data*, 1(4), dec 2023.

[222] Bhavik Nagda. *CHuff: Conditional Huffman String Compression*. PhD thesis, Massachusetts Institute of Technology, 2021.

[223] C.G. Nevill-Manning and I.H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Data Compression Conference (DCC)*, pages 3–11, Snowbird, UT, USA, 1997. IEEE Computer Society.

[224] N.J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[225] Kennon J. Conrad and Paul R. Wilson. Grammatical ziv-lempel compression: Achieving ppm-class text compression ratios with lz-class decompression speed. In *Proceedings of the 2016 Data Compression Conference (DCC)*, page 586, Snowbird, UT, USA, 2016. IEEE Computer Society.

[226] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, pages 72–83, Venice, Italy, 2017. OpenProceedings.org.

[227] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3), jun 2019.

[228] NVIDIA. nvcomp. `https://github.com/NVIDIA/nvcomp`, 2023. Accessed on: 2023-4-12.

[229] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 476–487, New York, NY, USA, 1998. Morgan Kaufmann.

[230] Quix.io. The anatomy of a machine learning pipeline. Online; accessed 2025-04-11, 2022. `https://quix.io/blog/the-anatomy-of-a-machine-learning-pipeline`.

[231] Harsh Mittal and Ankit Suresh. Studying the effect of bitwidth reduction on decision forest classifiers. *arXiv preprint arXiv:2106.14340*, 2021.

[232] Paulius Micikevicius, Sharan Narang, Jonah Alben, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2018.

[233] Amir Gholami, Sehoon Kim, Zhen Dong Yao, et al. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[234] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[235] Ron Banner, Yaniv Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *arXiv preprint arXiv:1810.05723*, 2018.

[236] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.

[237] Elena Krippner. Rethinking vector embeddings search for analytical database systems. Master's thesis, Augsburg University, Augsburg, Germany, October 2024. Master's thesis in the Elite Graduate Program in Software Engineering.

[238] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1390–1403, New York, NY, USA, 2022. Association for Computing Machinery.

[239] Facebook Incubator Team. Nimble: A columnar file format for feature engineering. `https://github.com/facebookincubator/nimble`, 2024. GitHub Repository.

[240] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1390–1403, New York, NY, USA, 2022. Association for Computing Machinery.

[241] Apache Iceberg. Apache iceberg specification - writer requirements, 2023. Accessed: 2023-11-01.

[242] Ziya Mukhtarov. Nested data-type encodings in fastlanes. Master's thesis, Technical University of Munich, 2024.

[243] CWI Database Architectures Group. Realnest - a collection of nested data from real-world datasets, 2024.

[244] Raufs Dunamalijevs. Predicate pushdown in fastlanes. Msc thesis, Universiteit van Amsterdam, 2024. Supervisors: Azim Afroozeh, Peter Boncz, and Balder ten Cate.

List of Publications

1. **Azim Afroozeh**, Peter Boncz: "The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code." Proceedings of the VLDB Endowment, 16(9), 2132–2144, 2023.
2. **Azim Afroozeh**, Leonardo Kuffó Rivero, Peter Boncz: "ALP: Adaptive Lossless Floating-Point Compression." Proceedings of the ACM on Management of Data, 1(4), 1–26, 2023.
3. **Azim Afroozeh**, L. Felius, Peter Boncz: "Accelerating GPU Data Processing using FastLanes Compression." Proceedings of the 20th International Workshop on Data Management on New Hardware, 2024.
4. **Azim Afroozeh**, Peter Boncz: "The FastLanes File Format." Proceedings of the VLDB Endowment, 18(11), 4629–4643, 2025.
5. Sven Hepkema, **Azim Afroozeh**, Charlotte Felius, Peter Boncz, Stefan Manegold: "G-ALP: Rethinking Light-weight Encodings for GPUs." DaMoN '25: Proceedings of the 21st International Workshop on Data Management on New Hardware, Article No. 11, 1–10, 2025.

Included in this thesis.

SIKS Dissertation Series

2016
01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VUA), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VUA), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VUA), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UvA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VUA), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VUA), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VUA), Refining Statistical Data on the Web
19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UvA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VUA), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UvA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior

26 Dilhan Thilakarathne (VUA), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (TiU), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (TiU), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime

02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UvA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VUA), Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TU/e), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (TiU), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UvA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VUA), Logics for causal inference under uncertainty
23 David Graus (UvA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VUA), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VUA), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (TiU), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (TiU), The Power of Facial Expressions

	31	Ben Ruijl (UL), Advances in computational methods for QFT calculations
	32	Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
	33	Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
	34	Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
	35	Martine de Vos (VUA), Interpreting natural science spreadsheets
	36	Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
	37	Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
	38	Alex Kayal (TUD), Normative Social Applications
	39	Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
	40	Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
	41	Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
	42	Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
	43	Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
	44	Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
	45	Bas Testerink (UU), Decentralized Runtime Norm Enforcement
	46	Jan Schneider (OU), Sensor-based Learning Support
	47	Jie Yang (TUD), Crowd Knowledge Creation Acceleration
	48	Angel Suarez (OU), Collaborative inquiry-based learning
2018	01	Han van der Aa (VUA), Comparing and Aligning Process Representations
	02	Felix Mannhardt (TU/e), Multi-perspective Process Mining
	03	Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
	04	Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
	05	Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process
	06	Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
	07	Jieting Luo (UU), A formal account of opportunism in multi-agent systems
	08	Rick Smetsers (RUN), Advances in Model Learning for Software Systems
	09	Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
	10	Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
	11	Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks

12 Xixi Lu (TU/e), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (TiU), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OU), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (TiU), Emotion Recognition from Mandarin Speech
30 Wouter Beek (VUA), The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
06 Chris Dijkshoorn (VUA), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction

11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VUA), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TU/e), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VUA),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VUA), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VUA), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OU), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VUA), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VUA), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TU/e), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TU/e), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OU), Gamification with digital badges in learning programming
36 Kevin Ackermans (OU), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OU), Learning visually grounded and multilingual representations

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour

02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TU/e), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OU), Towards a software architecture for reusable game components
08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
10 Alifah Syamsiyah (TU/e), In-database Preprocessing for Process Mining
11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
12 Ward van Breda (VUA), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
15 Konstantinos Georgiadis (OU), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
17 Daniele Di Mitri (OU), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VUA), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VUA), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TU/e), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality

	29	Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
	30	Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst
	31	Gongjin Lan (VUA), Learning better – From Baby to Better
	32	Jason Rhuggenaath (TU/e), Revenue management in online markets: pricing and online advertising
	33	Rick Gilsing (TU/e), Supporting service-dominant business model evaluation in the context of business model innovation
	34	Anna Bon (UM), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
	35	Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production
2021	01	Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
	02	Rijk Mercuur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
	03	Seyyed Hadi Hashemi (UvA), Modeling Users Interacting with Smart Devices
	04	Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
	05	Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
	06	Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
	07	Armel Lefebvre (UU), Research data management for open science
	08	Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
	09	Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
	10	Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
	11	Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
	12	Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
	13	Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
	14	Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
	15	Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
	16	Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
	17	Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
	18	Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
	19	Roberto Verdecchia (VUA), Architectural Technical Debt: Identification and Management
	20	Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
	21	Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
	22	Sihang Qiu (TUD), Conversational Crowdsourcing

	23	Hugo Manuel Proença (UL), Robust rules for prediction and description
	24	Kaijie Zhu (TU/e), On Efficient Temporal Subgraph Query Processing
	25	Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
	26	Benno Kruit (CWI/VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
	27	Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
	28	Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs
2022	01	Judith van Stegeren (UT), Flavor text generation for role-playing video games
	02	Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
	03	Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
	04	Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
	05	Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
	06	Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
	07	Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
	08	Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
	09	Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
	10	Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
	11	Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
	12	Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
	13	Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
	14	Michiel Overeem (UU), Evolution of Low-Code Platforms
	15	Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
	16	Pieter Gijsbers (TU/e), Systems for AutoML Research
	17	Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification
	18	Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
	19	Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
	20	Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media

	21	Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
	22	Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
	23	Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
	24	Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
	25	Anna L.D. Latour (UL), Optimal decision-making under constraints and uncertainty
	26	Anne Dirkson (UL), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
	27	Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
	28	Onuralp Ulusoy (UU), Privacy in Collaborative Systems
	29	Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
	30	Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
	31	Konstantinos Traganos (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
	32	Cezara Pastrav (UU), Social simulation for socio-ecological systems
	33	Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management
	34	Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
	35	Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction
2023	01	Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
	02	Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
	03	Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
	04	Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval
	05	Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
	06	António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
	07	Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning
	08	Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
	09	Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
	10	Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing

	11	Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
	12	Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
	13	Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
	14	Selma Čaušević (TUD), Energy resilience through self-organization
	15	Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models
	16	Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters
	17	Meike Nauta (UT), Explainable AI and Interpretable Computer Vision – From Oversight to Insight
	18	Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
	19	George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
	20	Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
	21	Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
	22	Alireza Shojaifar (UU), Volitional Cybersecurity
	23	Theo Theunissen (UU), Documentation in Continuous Software Development
	24	Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
	25	Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
	26	Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
	27	Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
	28	Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts
	29	Tim Draws (TUD), Understanding Viewpoint Biases in Web Search Results
2024	01	Daphne Miedema (TU/e), On Learning SQL: Disentangling concepts in data systems education
	02	Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
	03	Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
	04	Mike Huisman (UL), Understanding Deep Meta-Learning
	05	Yiyong Gou (UM), Aerial Robotic Operations: Multi-environment Cooperative Inspection & Construction Crack Autonomous Repair
	06	Azqa Nadeem (TUD), Understanding Adversary Behavior via XAI: Leveraging Sequence Clustering to Extract Threat Intelligence
	07	Parisa Shayan (TiU), Modeling User Behavior in Learning Management Systems

08 Xin Zhou (UvA), From Empowering to Motivating: Enhancing Policy Enforcement through Process Design and Incentive Implementation
09 Giso Dal (UT), Probabilistic Inference Using Partitioned Bayesian Networks
10 Cristina-Iulia Bucur (VUA), Linkflows: Towards Genuine Semantic Publishing in Science
11 withdrawn
12 Peide Zhu (TUD), Towards Robust Automatic Question Generation For Learning
13 Enrico Liscio (TUD), Context-Specific Value Inference via Hybrid Intelligence
14 Larissa Capobianco Shimomura (TU/e), On Graph Generating Dependencies and their Applications in Data Profiling
15 Ting Liu (VUA), A Gut Feeling: Biomedical Knowledge Graphs for Interrelating the Gut Microbiome and Mental Health
16 Arthur Barbosa Câmara (TUD), Designing Search-as-Learning Systems
17 Razieh Alidoosti (VUA), Ethics-aware Software Architecture Design
18 Laurens Stoop (UU), Data Driven Understanding of Energy-Meteorological Variability and its Impact on Energy System Operations
19 Azadeh Mozafari Mehr (TU/e), Multi-perspective Conformance Checking: Identifying and Understanding Patterns of Anomalous Behavior
20 Ritsart Anne Plantenga (UL), Omgang met Regels
21 Federica Vinella (UU), Crowdsourcing User-Centered Teams
22 Zeynep Ozturk Yurt (TU/e), Beyond Routine: Extending BPM for Knowledge-Intensive Processes with Controllable Dynamic Contexts
23 Jie Luo (VUA), Lamarck's Revenge: Inheritance of Learned Traits Improves Robot Evolution
24 Nirmal Roy (TUD), Exploring the effects of interactive interfaces on user search behaviour
25 Alisa Rieger (TUD), Striving for Responsible Opinion Formation in Web Search on Debated Topics
26 Tim Gubner (CWI), Adaptively Generating Heterogeneous Execution Strategies using the VOILA Framework
27 Lincen Yang (UL), Information-theoretic Partition-based Models for Interpretable Machine Learning
28 Leon Helwerda (UL), Grip on Software: Understanding development progress of Scrum sprints and backlogs
29 David Wilson Romero Guzman (VUA), The Good, the Efficient and the Inductive Biases: Exploring Efficiency in Deep Learning Through the Use of Inductive Biases
30 Vijanti Ramautar (UU), Model-Driven Sustainability Accounting
31 Ziyu Li (TUD), On the Utility of Metadata to Optimize Machine Learning Workflows
32 Vinicius Stein Dani (UU), The Alpha and Omega of Process Mining
33 Siddharth Mehrotra (TUD), Designing for Appropriate Trust in Human-AI interaction
34 Robert Deckers (VUA), From Smallest Software Particle to System Specification - MuDForM: Multi-Domain Formalization Method

35 Sicui Zhang (TU/e), Methods of Detecting Clinical Deviations with Process Mining: a fuzzy set approach
36 Thomas Mulder (TU/e), Optimization of Recursive Queries on Graphs
37 James Graham Nevin (UvA), The Ramifications of Data Handling for Computational Models
38 Christos Koutras (TUD), Tabular Schema Matching for Modern Settings
39 Paola Lara Machado (TU/e), The Nexus between Business Models and Operating Models: From Conceptual Understanding to Actionable Guidance
40 Montijn van de Ven (TU/e), Guiding the Definition of Key Performance Indicators for Business Models
41 Georgios Siachamis (TUD), Adaptivity for Streaming Dataflow Engines
42 Emmeke Veltmeijer (VUA), Small Groups, Big Insights: Understanding the Crowd through Expressive Subgroup Analysis
43 Cedric Waterschoot (KNAW Meertens Instituut), The Constructive Conundrum: Computational Approaches to Facilitate Constructive Commenting on Online News Platforms
44 Marcel Schmitz (OU), Towards learning analytics-supported learning design
45 Sara Salimzadeh (TUD), Living in the Age of AI: Understanding Contextual Factors that Shape Human-AI Decision-Making
46 Georgios Stathis (Leiden University), Preventing Disputes: Preventive Logic, Law & Technology
47 Daniel Daza (VUA), Exploiting Subgraphs and Attributes for Representation Learning on Knowledge Graphs
48 Ioannis Petros Samiotis (TUD), Crowd-Assisted Annotation of Classical Music Compositions

---

2025 01 Max van Haastrecht (UL), Transdisciplinary Perspectives on Validity: Bridging the Gap Between Design and Implementation for Technology-Enhanced Learning Systems
02 Jurgen van den Hoogen (JADS), Time Series Analysis Using Convolutional Neural Networks
03 Andra-Denis Ionescu (TUD), Feature Discovery for Data-Centric AI
04 Rianne Schouten (TU/e), Exceptional Model Mining for Hierarchical Data
05 Nele Albers (TUD), Psychology-Informed Reinforcement Learning for Situated Virtual Coaching in Smoking Cessation
06 Daniël Vos (TUD), Decision Tree Learning: Algorithms for Robust Prediction and Policy Optimization
07 Ricky Maulana Fajri (TU/e), Towards Safer Active Learning: Dealing with Unwanted Biases, Graph-Structured Data, Adversary, and Data Imbalance
08 Stefan Bloemheuvel (TiU), Spatio-Temporal Analysis Through Graphs: Predictive Modeling and Graph Construction
09 Fadime Kaya (VUA), Decentralized Governance Design - A Model-Based Approach
10 Zhao Yang (UL), Enhancing Autonomy and Efficiency in Goal-Conditioned Reinforcement Learning

11 Shahin Sharifi Noorian (TUD), From Recognition to Understanding: Enriching Visual Models Through Multi-Modal Semantic Integration
12 Lijun Lyu (TUD), Interpretability in Neural Information Retrieval
13 Fuda van Diggelen (VUA), Robots Need Some Education: on the complexity of learning in evolutionary robotics
14 Gennaro Gala (TU/e), Probabilistic Generative Modeling with Latent Variable Hierarchies
15 Michiel van der Meer (UL), Opinion Diversity through Hybrid Intelligence
16 Monika Grewal (TU Delft), Deep Learning for Landmark Detection, Segmentation, and Multi-Objective Deformable Registration in Medical Imaging
17 Matteo De Carlo (VUA), Real Robot Reproduction: Towards Evolving Robotic Ecosystems
18 Anouk Neerincx (UU), Robots That Care: How Social Robots Can Boost Children's Mental Wellbeing
19 Fang Hou (UU), Trust in Software Ecosystems
20 Alexander Melchior (UU), Modelling for Policy is More Than Policy Modelling (The Useful Application of Agent-Based Modelling in Complex Policy Processes)
21 Mandani Ntekouli (UM), Bridging Individual and Group Perspectives in Psychopathology: Computational Modeling Approaches using Ecological Momentary Assessment Data
22 Hilde Weerts (TU/e), Decoding Algorithmic Fairness: Towards Interdisciplinary Understanding of Fairness and Discrimination in Algorithmic Decision-Making
23 Roderick van der Weerdt (VUA), IoT Measurement Knowledge Graphs: Constructing, Working and Learning with IoT Measurement Data as a Knowledge Graph
24 Zhong Li (UL), Trustworthy Anomaly Detection for Smart Manufacturing
25 Kyana van Eijndhoven (TiU), A Breakdown of Breakdowns: Multi-Level Team Coordination Dynamics under Stressful Conditions
26 Tom Pepels (UM), Monte-Carlo Tree Search is Work in Progress
27 Danil Provodin (JADS, TU/e), Sequential Decision Making Under Complex Feedback
28 Jinke He (TU Delft), Exploring Learned Abstract Models for Efficient Planning and Learning
29 Erik van Haeringen (VUA), Mixed Feelings: Simulating Emotion Contagion in Groups
30 Myrthe Reuver (VUA), A Puzzle of Perspectives: Interdisciplinary Language Technology for Responsible News Recommendation
31 Gebrekirstos Gebreselassie Gebremeskel (RUN), Spotlight on Recommender Systems: Contributions to Selected Components in the Recommendation Pipeline
32 Ryan Brate (UU), Words Matter: A Computational Toolkit for Charged Terms
33 Merle Reimann (VUA), Speaking the Same Language: Spoken Capability Communication in Human-Agent and Human-Robot Interaction
34 Eduard C. Groen (UU), Crowd-Based Requirements Engineering
35 Urja Khurana (VUA), From Concept To Impact: Toward More Robust Language Model Deployment

36 Anna Maria Wegmann (UU), Say the Same but Differently: Computational Approaches to Stylistic Variation and Paraphrasing
37 Chris Kamphuis (RUN), Exploring Relations and Graphs for Information Retrieval
38 Valentina Maccatrozzo (VUA), Break the Bubble: Semantic Patterns for Serendipity
39 Dimitrios Alivanistos (VUA), Knowledge Graphs & Transformers for Hypothesis Generation: Accelerating Scientific Discovery in the Era of Artificial Intelligence
40 Stefan Grafberger (UvA), Declarative Machine Learning Pipeline Management via Logical Query Plans
41 Mozhgan Vazifehdoostirani (TU/e), Leveraging Process Flexibility to Improve Process Outcome - From Descriptive Analytics to Actionable Insights
42 Margherita Martorana (VUA), Semantic Interpretation of Dataless Tables: a metadata-driven approach for findable, accessible, interoperable and reusable restricted access data
43 Krist Shingjergji (OU), Sense the Classroom - Using AI to Detect and Respond to Learning-Centered Affective States in Online Education
44 Robbert Reijnen (TU/e), Dynamic Algorithm Configuration for Machine Scheduling Using Deep Reinforcement Learning
45 Anjana Mohandas Sheeladevi (VUA), Occupant-Centric Energy Management: Balancing Privacy, Well-being and Sustainability in Smart Buildings
46 Ya Song (TU/e), Graph Neural Networks for Modeling Temporal and Spatial Dimensions in Industrial Decision-making
47 Tom Kouwenhoven (UL), Collaborative Meaning-Making. The Emergence of Novel Languages in Humans, Machines, and Human-Machine Interactions
48 Evy van Weelden (TiU), Integrating Virtual Reality and Neurophysiology in Flight Training
49 Selene Báez Santamaría (VUA), Knowledge-centered conversational agents with a drive to learn
50 Lea Krause (VUA), Contextualising Conversational AI
51 Jiaxu Zhao (TU/e), Understanding and Mitigating Unwanted Biases in Generative Language Models
52 Qiao Xiao (TU/e), Model, Data and Communication Sparsity for Efficient Training of Neural Networks
53 Gaole He (TUD), Towards Effective Human-AI Collaboration: Promoting Appropriate Reliance on AI Systems
54 Go Sugimoto (VUA), MISSING LINKS Investigating the Quality of Linked Data and its Tools in Cultural Heritage and Digital Humanities
55 Sietze Kai Kuilman (TUD), AI that Glitters is Not Gold: Requirements for Meaningful Control of AI Systems
56 Wijnand van Woerkom (UU), A Fortiori Case-Based Reasoning: Formal Studies with Applications in Artificial Intelligence and Law
57 Syeda Amna Sohail (UT), Privacy-Utility Trade-Off in Healthcare Metadata Sharing and Beyond: A Normative and Empirical Evaluation at Inter and Intra Organizational Levels

	58	Junhan Wen (TUD), "From iMage to Market": Machine-Learning-Empowered Fruit Supply
	59	Mohsen Abbaspour Onari (TU/e), From Explanation to Trust: Modeling and Measuring Trust in Explainable Decision Support
	60	Marcel Jurriaan Robeer (UU), Beyond Trust: A Causal Approach to Explainable AI in Law Enforcement
	61	Shuai Wang (VUA), Links in Large Integrated Knowledge Graphs: Analysis, Refinement, and Domain Applications
	62	Khaleel Asyraaf Mat Sanusi (OU), Augmenting a learning model within immersive learning environments for psychomotor skills
	63	Rashid Zaman (TU/e), Online Conformance Checking on Degraded Data
	64	Jens d'Hondt (TU/e), Effective and Efficient Multivariate Similarity Search
	65	Aswin Balasubramaniam (UT), Disentangling Runner Drone Interaction Potentialities
2026	01	Pei-Yu Chen (TUD), Human-Agent Alignment Dialogues: Eliciting User Information at Runtime for Personalized Behavior Support
	02	Hezha Hassan Mohammedkhan (TiU), Estimating Body Measurements of Children from 2D Images: Towards the Automatic Detection of Malnutrition
	03	Kyriakos Psarakis (TUD), Democratizing Scalable Cloud Applications: Transactional Stateful Functions on Streaming Dataflows
	04	Boyu Xu (UU), Exploring Indirect Relations Between Topics in Neuroscience Literature Using Augmented Reality to Inform Experimental Design
	05	Koen Minartz (TU/e), Stochastic Simulation with Geometric Deep Generative Models
	06	Azim Afroozeh (CWI, VUA), FastLanes: A Next-Gen File Format
	07	Inès Blin (VUA), Narrative Understanding with Knowledge Graphs