TOWARDS ARCHITECTURE-AWARE DATA ANALYTICS

by

Martin Prammer

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: August 7th, 2024

The dissertation is approved by the following members of the Final Oral Committee:
Jignesh M. Patel, Professor, Computer Science, Carnegie Mellon University
Paris Koutris, Associate Professor, Computer Science, University of Wisconsin-Madison
Xiangyao Yu, Assistant Professor, Computer Science, University of Wisconsin-Madison
Karthikeyan Sankaralingam, Professor, Computer Science, University of Wisconsin-Madison
José F. Martínez, Professor, College of Engineering, Cornell University

To my family; by birth and found along the way.

ACKNOWLEDGMENTS

I am fortunate to have collaborated with many groups over the years, all of whom have influenced me for the better. My identity as a person is built upon my experiences; I would not be who I am today without these memories.

Professor Jignesh M. Patel: one whose impact is difficult to convey. Good advisors naturally accompany their students through the highs and lows, the struggles and triumphs. Great advisors have the foresight to plan through these individual events to shape cohesive identities for their students, allowing them to flourish into subject matter experts. Jignesh was more. Looking back, I now see the depth of his insight and guidance. In some ways, pursuing a Ph.D. is like theater, where students both playwright and perform the story of their research; the best advisors are directors, who leave an indelible mark that elevates their advisee's research to so much more than it would be without them. All of Jignesh's students sing praise for his research acumen, technical insight, and groundedness; I join this chorus as well.

Next, I thank my family. Growing up in the presence of academics certainly influenced my decision to pursue further academic studies beyond an undergraduate degree. Over the course of my Ph.D. studies, I believe that I have grown into a worthy peer. Likewise, my family's non-academic members have had a profound impact on my development as well; deep technical competence and insight are all around us, ready to be shared once we open our hearts and minds.

Of course, my "family away from home" also deserves recognition; the other students within the UW-Madison database group, especially those also mentored by Prof. Patel, were a constant source of inspiration and insight over the multiple years we interacted. Similarly, the database and computer architecture faculty members at UW-Madison have, time after time, showcased their research and technical acumen. Even further away are the phenomenal students and faculty I've collaborated with across the world; I hope we will collaborate once again.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With rapidly increasing data sizes, data analytic tasks are increasingly hindered by the memory wall: the performance disparity between processing data already present in CPU caches vs. data retrieved from memory. Thus, while the data processing capabilities of modern computers continue to improve, database management systems are still limited by the fundamental bottleneck of moving data to the CPU.

In response, this work poses the following question: Must the entirety of the data be processed to perform analytics? More precisely, must a given machine process every single bit of every single value in a single column from a database table? The answer to this question is "No," as demonstrated by three distinct mechanisms that each provide a unique solution to bypass processing portions of data. First, this work explores data layouts, demonstrating that performance must be rooted in storage formats that facilitate efficient and sequential access patterns. Next, data representations are explored to demonstrate that particular integer encodings can be leveraged to process only some bits per value, enabling partial-value compute and reducing the column's effective bit width. Finally, metadata and its relationship to underlying columnar data is leveraged to bypass entire horizontal data partitions. Taken together, many data analytic tasks only require processing a fraction of the underlying data, resulting in speedups by virtue of both reduced computational costs and data movement.

LAY SUMMARY

Big data is the fuel for modern businesses: predicting future trends requires knowledge of the past. However, finding insights in massive amounts of data is akin to finding needles in haystacks. My work is focused on finding these needles faster.

In particular, my work focuses on looking for these "needles" within lists of items; for example, given a cumbersomely large receipt, find items that cost more than $100 (USD). While seemingly a very specific task, "searching through a list" is a fundamental operation within data science. If we know nothing about a list and its contents, the only course of action is to sequentially "scan" it; without guarantees regarding ordering (such as if the list is sorted), knowing the first element gives us no information about the second element, the second about the third, and so on.

A natural way to perform a list-searching task would be to read out each item, row-by-row, checking the price of the item after reading its name. However, it is also possible for someone to read "down the column" of the price of every item; each time an item over $100 is found, then the rest of the receipt line can be read. Further still, we recognize that all prices in the receipt are aligned to the right side of the receipt paper, always containing two decimal points for cents. For items under $100, the price can be represented using 4 digits and a decimal point. For items at or over $100, they must occupy a 5th digit space, one place to the left of the tens amount of dollars. Thus, we now have a new mechanism to find the $100 or more items even faster by simply looking down the receipt at the exact digit where that 5th digit must be placed; usually empty for most items, but sometimes occupied, only when a value is $100 or more. This example is a metaphor for the differences between three different families of techniques used to store data: row-stores, column-stores, and *bit-stores*. This last area is one that I have researched extensively, where my work changes the way we represent numbers to enable "5th digit" and related cases to occur more frequently.

My other works can be thought of using the receipt metaphor as well. First, we need to print the receipt so that we can easily scan over each price; if we're wasting time trying to find the price for each item on each row, then we should

find a better way to print the receipt, such as lining up the prices at the right-hand side of the receipt. This work emphasizes the importance of using a column-store for analytics tasks: We created an in-memory column-store layer for a file-based row-store database management program.

Second, if we cut the large receipt into smaller "partial receipts," a technique known within databases as horizontal partitioning, we could write the largest and smallest price on the back of each partial receipt. This technique is highly useful if we ever want to look at the receipts again, where we can skip looking at the front of receipts with a maximum price of less than $100. By using the notes on the back, we avoid spending effort checking the price of many receipt items.

To implement these kinds of improvements within computers that process massive amounts of data, care must be taken to use each computer component to the fullest extent. This is a hard challenge, as the way that underlying systems work and interact with large-scale data tasks is not always obvious. Thus, we have to design our software in a way that is aware of the underlying computing resources; hence, the title of this thesis: Towards Architecture-Aware Data Analytics.

1.1 Introduction

Oft-cited as a motivation for improving the performance of data analytics is the monumental explosion of data. With forecasts that over 180 zettabytes of data will be produced in 2025, these motivations are very real [81].

Data analytics uses this vast amount of data to produce actionable insights, such as finding patterns to forecast future events from past data. Naturally, the first step in performing such an analysis is filtering out records irrelevant to the particular query at hand. For example, if a store wished to evaluate the success of a promotional campaign run in a single geographic area targeting certain demographics, it would want to filter its sales records to the specific area in question. This filtering operation is generally performed via a scan with minimal indexing. This is because building an index for such a query is difficult, if not impossible, for three general reasons:

1. The query predicate is generally not known until run-time. Thus, precomputing work is out of the question.

2. Similarly, queries may filter over multiple data attributes, such as our area and demographic filter. Sorting data over multiple attributes is similarly inflexible, as even our attributes for filtering might not be known until run-time.

3. Finally, the data may not even be present until run-time, such as in the case of real-time data analytics.

Thus, we find ourselves in an environment where scanning over the entirety of a dataset seems inevitable. However, this is also not always the case. Data analytic benchmarks regularly classify particular workloads by selectivity: the portion of the underlying data remaining after all filtering steps are performed by a particular query [63; 82]. Also represented as a "filter factor" metric, we quickly find that only small portions of data impact a query's end result. Unfortunately, this introduces a large performance bottleneck, as massive amounts of values must

be processed before we can find the few that impact the query output. This issue is magnified even further by the *memory wall*, which introduces significant costs when moving data from its at-rest location to a processing element that can evaluate the filter [11; 92; 93].

Thus, my body of work has posed a fundamental question: How can we be more efficient when analyzing data? We have found answers to this question in three major areas:

1. We can optimize the underlying representation of data, such as by leveraging columnar-first data storage structures (Chapter 2).

2. We can reorganize data to better enable partial-value compute. Under this construction, certain queries can be answered by only loading a fraction of the data; thus, partial-value compute effectively shrinks the original dataset to a much more manageable size (Chapter 3).

3. We can leverage lightweight metadata structures to perform predicate push-down over one or more horizontal partitions of data. By avoiding processing large portions of the underlying data, we have effectively shrunk the original dataset (Chapter 4).

The remainder of this chapter is dedicated to a breadth of background information, including databases, computer architecture, and a few other areas. Finally, my closing thoughts and plans for future work are included in Chapter 5.

1.2 Background: Databases

In this section, we highlight two important considerations for this work from the perspective of databases.

1.2.1 Analytics-focused Databases

Attribute-based filters are ubiquitous in data analytic workloads. All queries in the Star Schema Benchmark (SSB) [63] include a filtering operation (`SELECT ... WHERE ...`), while almost all TPC-H [82] queries do as well. To this end, we find it prudent to accelerate such a common data analysis primitive.

Columnar storage formats accelerate attribute-filtering by rearranging traditional row storage formats via vertical partitioning along attribute boundaries [71; 80]. In many cases, these vertical partitions are then horizontally partitioned, resulting in large columnar "chunks" or "blocks," regularly 1M+ rows each [5; 6; 75]. Columnarizing data is patently useful for such filtering operations, as now row-exclusion may occur without processing the entirety of each row.

1.2.2 Declarative Interfaces

A crucial, fundamental property of SQL is that there is no singular implementation of the underlying storage layer. In fact, Codd's original 12 rules were intended as a measure of how faithfully DBMS offerings at the time were implementing the relational model; rule 12 (nonsubversion) is expressly intended to be difficult for "born-again" systems to satisfy [14; 15]. Thus, the question of implementation has always been left open-ended.

The impact of declarative interfaces is most clearly seen in column stores [5; 6; 71; 75; 80], and by extension, *bit-parallel stores* [24; 51; 64; 70]. The significant differences in "under the hood" storage-layer implementations lead to specialization-based performance improvements.

1.3 Background: Computer Architecture

In this section, we discuss computer architecture topics that fundamentally impact the practicalities of database systems.

1.3.1 Cache Lines

Modern CPUs store data locally in a hierarchical arrangement of caches, each able to store many cache lines [16]. In particular, the cache line size is 64 bytes. While this size could theoretically change, the practical reality is that significant amounts of hardware must be changed to accommodate such a shift. Thus, for the foreseeable future, cache lines are 64 bytes.

While a cache line is not atomic, performance may rapidly degrade if operations occur on non-cache-aligned boundaries. Thus, cache alignment is a crucial consideration for database techniques that optimize data structure performance [4; 32].

1.3.2 Row Buffers

Even though data is generally presented to the CPU one cache line at a time, memory operates at a much larger scale [16]. While dependent on the implementation of the memory system, memory "rows" are generally at least 1KB in size, resulting in at least 16 cache lines per row. Before a cache line within a row can be transmitted to the CPU, the data is retrieved from the underlying storage system and placed into one or more buffers. Once in this buffering mechanism, individual cache lines can be extracted and transmitted to the CPU. However, there is a non-zero cost to perform such buffering. While the technical details of row switching are beyond the scope of this work, it is important to understand that placing a row in the row buffer requires hundreds of CPU cycles [17]. This is partly why sequential and random accesses to underlying memory systems usually have significantly different performance characteristics.

1.3.3 SIMD

It is well established that database algorithms are "embarrassingly parallel" [51; 67; 77; 80; 88]. Unsurprisingly, *single instruction, multiple data* (SIMD) instructions present in modern CPUs are incredibly useful for database-related compute. Re-

cent years have seen a perpetual trend toward vectorizing compute for impressive performance improvements [67; 88].

The general usage of "SIMD instructions" usually refers to an x86 ISA SIMD extension, such as SSE, AVX2, or AVX512. In this context, vectorizing an application usually involves rewriting the program to use SIMD intrinsics, either directly or through a library that improves the portability of the low-level intrinsics. Further, many modern compilers can automatically recognize when scalar instructions can be replaced with SIMD instructions in a process known as *auto-vectorization* [62].

1.4 Background: Miscellaneous

This section begins this thesis's analysis of the totality of the works discussed herein. While the previous background sections are directly relevant to each individual work discussed within this document, this section introduces background information that is used together to contextualize the entire work as a cohesive whole.

1.4.1 Information Theory

While information theory captures many foundational topics, we highlight only a few in this section [78]. In particular, we focus on the framing of a *general communication system* which consists of five parts:

1. The *information source*, which produces the *message* to be communicated.
2. The *transmitter*, which operates on the message to produce a signal suitable for transmission over the channel.
3. The *channel*, which is the medium of communication between transmitter and receiver.
4. The *receiver*, which inverts the operation applied by the transmitter.
5. The *destination*, which receives the *message*.

We focus on the discrete and error-free case of communication, where the transmitter encodes the message as a sequence of *symbols* from a known messaging *alphabet*. For example, if we were to send a 32-bit binary value from source to destination, the transmitter may send a sequence of 32 symbols from the alphabet $\{0, 1\}$. Or, the transmitter could send a sequence of 4 symbols from the alphabet consisting of integers in the inclusive range $[0, 255]$.

While a seemingly trivial formulation, this structuring is incredibly useful when analyzing existing database literature focusing on *bit-parallelism* [24; 51; 64; 70]. This foundation allows for a more general construction of *early stopping*, which we introduce in Section 3.2 and extend in Section 3.7.

1.4.2 Interval Arithmetic

Interval arithmetic allows for traditional variables to be replaced by numeric ranges in arithmetic expressions [34]. Below is the notation of an interval:

$$[x] = [x_1, x_2] \equiv \{x \in \mathbb{R} | x_1 \leqslant x \leqslant x_2\}$$

Where x_1 and x_2 are referred to as the infimum and supremum, respectively.

Of the defined arithmetic operations, we highlight the procedure to multiply intervals $[x]$ and $[y]$:

$$[x] * [y] = [\mathit{min}(x_1y_1, x_1y_2, x_2y_1, x_2y_2), \mathit{max}(x_1y_1, x_1y_2, x_2y_1, x_2y_2)]$$

In fact, the construction of multiplication is close to the general form of any binary operation on two intervals. The verbosity of the expression for multiplication is due to the unknown signs of the infimum and supremum.

For our particular uses, we find the following shorthand notation useful: We observe that it is regularly convenient to express $[x_1, x_2]$ as an endpoint and an offset from that endpoint rather than two endpoints. In this case, it is useful to define $o = x_2 - x_1$ and represent the interval as $[x_1, x_2] = [x_1, x_1 + o]$. Significantly, this

form can be much easier to perform arithmetic with, especially when o is simple or small compared to a much larger or complex x_1.

We do not intend to leverage interval arithmetic too heavily in this work. However, this construction is necessary for the expansion of partial-value compute from Chapter 3 into more general forms (see Section 3.7).

1.4.3 Sampling Distributions

As many topics within databases involve collections of data with some underlying distribution, it is beneficial to have a strong grasp on statistics, specifically the behavior of drawing samples from distributions. Consider rolling a six-sided die [68]. In this case, the die itself is a uniform random distribution of values, where each face is equally likely to be rolled. We can express the value of the average roll as the sum of faces divided by the cardinality of the die: $\frac{1+2+3+4+5+6}{6} = 3.5$. If we were to roll the same die twice and sum the results, our average outcome would be $2\frac{1+2+3+4+5+6}{6} = 7$.

However, when rolling two dice, rolling a sum of 7 is more likely than any other combination; by sampling from two independent uniform random distributions, we find our outcome distribution is no longer uniformly random. Intuitively, 7 being the most common value is not particularly surprising, as there are 6 different ways to sum integers in the inclusive range $[1, 6]$ to reach 7 ($\{1{+}6, 2{+}5, 3{+}4, 4{+}3, 5{+}2, 6{+}1\}$), the most of any possible outcome.

This observation stems from the central limit theorem (CLT) [12; 68]. In plain English, the CLT represents the following idea: If we draw samples from a distribution in repeated trials, given enough samples per trial, the distribution of the average sample per trial will nearly approach a normal distribution. Within databases, the CLT creates a push-pull relationship: normalcy leads to more predictable behavior, which can be optimized for, but reduces the distinctness of any given distribution, which could have been used to create additional performance opportunities. We discuss these performance opportunities in Chapter 4, though the impact of the CLT is visible throughout this work.

2 SQLITE3/HE

2.1 Introduction

SQLite [35], a monumentally successful relational database management system (DBMS), serves as the database of choice for embedded systems. SQLite's high performance on these systems is driven by a number of deliberate design decisions, which together emphasize hardware compatibility and low software overhead. However, these decisions come at a significant cost: while SQLite functions well for transaction-focused workloads in a lightweight environment, these same optimizations significantly impede its analytical query performance.

One can imagine a small computer placed inside a wind turbine generator ("windmill"), connected to an array of sensors that monitor performance, maintenance, and other concerns. Similar to many other real world applications that benefit from the local availability of compute resources, this example of a windmill is a member of a broad class of edge devices applications [79]. Naturally, the sensor data would benefit from being stored in a query-accessible format, allowing seamless integration of both the recording of sensor readings and metadata. This in turn would facilitate real-time analysis of the data, enabling the tracking of long-term electricity generation trends. However, edge computing cases have three key differences when compared to traditional SQLite-powered embedded systems.

First, the data analysis tasks in the edge computing domain require highly efficient performance for both transaction- and analytics-focused workloads, as the data logging and analysis tasks are usually performed concurrently. This change in use case reflects a significant divergence from the historical needs of embedded systems: edge devices usually need to perform a number of concurrent tasks, usually through the cooperation of an operating system scheduler and hardware parallelism.

Next, the compute resources available to the devices in this domain have grown significantly over the last decade, blurring the boundary between a small personal computer and a traditional embedded device. These devices are now regularly

equipped with multiple processing cores and relatively large amounts of memory. Likewise, each of these cores comes equipped with hardware support for vectorized instructions. Together, these advancements have drastically improved the computing power available to even the smallest of devices in this domain.

Finally, many edge devices are powered by a traditional computational stack, complete with an operating system and utility software. While these devices do share many constraints with the older class of traditional embedded devices, other factors such as executable image size are significantly less of a problem. These edge devices demonstrate the shifting of computational paradigms; for example, they prioritize portability and maintainability, to the point where a performance benefit may not be worth the cost of sacrificing either.

Still, SQLite has demonstrated that it deserves a place in modern DBMS offerings, usually as the database of choice for transactional workloads run on embedded devices. Thus, we find it highly relevant to provide a mechanism for SQLite to perform comparably to the state-of-the-art when processing analytical queries. To this end, we have developed SQLite3 Hustle Edition (SQLite3/HE), a query acceleration path that is able to maximize machine resource utilization leading to significant improvements in SQLite's analytical query processing performance [38]. This performance improvement is driven by the usage of both an in-memory, column-store and a set of analytics-optimized query operators. Most notably, the analytical query acceleration path is only taken by queries that benefit from such acceleration. Thus, SQLite3/HE is able to function as a drop-in augmentation for SQLite3 without fear of performance degradation in existing workloads.

In this paper, we make the following contributions:

1. An analysis of the technical behavior of SQLite3. This analysis explores the deliberate design decisions that, while appropriate for the older class of traditional embedded devices, have created the need for SQLite3/HE.

2. An exploration into the implementation of SQLite3/HE, detailing the mechanisms that support both its in-memory database and its analytics-optimized query operators.

3. An analysis of SQLite3/HE's performance, both in analytical and transactional workloads, which demonstrates its effectiveness as a drop-in add-on for existing SQLite3 implementations.

2.2 Background

SQLite3/HE utilizes many existing, state-of-the-art techniques to improve analytical query performance. Most notably is its usage of a secondary storage layer, facilitating a fundamentally different set of performance behaviors than that of SQLite. We explore this and related topics in this section.

The practice of optimizing a database for read-focused queries has been well established in recent years [52; 80]. One of the most prominent mechanisms to achieve high read performance is to "transpose" the database, a process in which fields of a single column are placed adjacent in the storage layer instead of preserving locality within a single record. This storage format provides two major optimizations:

1. The tuple members that are unrelated to the current query are not accessed (which would be the case if an entire row needs to be read, e.g., due to hardware limitations).

2. The tuple members that are related to the current query are placed consecutively in storage, naturally aligning their sequential access with existing hardware-based optimizations.

Given these advantages, this storage format is more amenable towards calculating aggregate values from a subset of the data. Likewise, this storage format is also highly compatible with machine specific optimizations, such as vector instructions and hardware prefetching.

However, the two previously mentioned optimizations are inverted when discussing either writes or accessing entire tuples. Further, when considering the writing of tuples, we find that we achieve these optimizations by utilizing a row-

based storage method, allowing for a write to act similarly to a simple append operation. Thus, we realize that for our particular windmill example, there is a need for both kinds of storage formats. To this end, and as part of a larger body of research, we see the benefit of, broadly, "hybrid stores" [65]. Of course, preserving the benefits of each storage solution simultaneously is both a difficult and an application-specific feat, especially without causing a significant amount of software-related overhead. To this end, only recently have these systems begun to enter the embedded device space, which imposes not only the aforementioned limitations but also further restrictions by nature of limited machine resources.

2.3 SQLite

In order to fully explore the purpose of specific design choices within SQLite3/HE, we must first examine SQLite.

2.3.1 SQLite3 Implementation

As SQLite3/HE is implemented as an acceleration path for specific queries, it is important to understand the behavior of SQLite and how SQLite3/HE fits into the existing query processing pipeline. To accomplish this, we explore specific SQLite3 components related to SQLite3/HE's acceleration path.

B-Tree Storage Format

SQLite primarily utilizes a single, monolithic database file; while this approximation is not entirely accurate, SQLite can be thought of as a sort of "SQL Interface" for a flat file database. This design paradigm benefits resource-constrained environments, allowing SQLite to fine-tune its data storage mechanisms without placing a significant burden on the operating system's file management system. However, given the unwieldiness of using a single file as a database, SQLite3 utilizes a page-based, B-tree structure: rows are stored as individual pages at the leaf level, which are then organized into a B-tree. This implementation functions incredibly well for

the core set of applications that SQLite targets. However, as scanning a column requires a large number of page walks, this fundamental design choice creates a significant performance bottleneck for the analytical query execution path.

The Query Pipeline

Similar to other DBMS solutions, SQLite utilizes a query pipeline structure composed of a number of familiar steps: specifically, the tokenizer, parser, planner, optimizer, and executor. Internally, the Lemon parser generator [36] creates a parser that performs both the parsing and the planning tasks, receiving tokens from the tokenizer to construct a parse tree. However, the query optimizer is heavily connected to the translation unit for the Virtual Database Engine, and thus is discussed separately. In general, this compartmentalization of work creates the opportunity to intercept queries as they progress through the query processing pipeline. SQLite3/HE intercepts queries once the parse tree has been constructed and before they reach the optimization-translation unit.

Virtual Database Engine

SQLite transforms queries into a format reminiscent of assembly instructions. Virtual Database Engine (VDBE) instructions consist of an opcode and a number of operands (5 in SQLite3, though previous versions had fewer). These VDBE instructions assist in maintaining a high degree of portability for SQLite, as a system-specific implementation need only implement the machine instructions for each VDBE instruction. Unfortunately, this design creates a number of problems when integrating modern high-performance query optimization methods into SQLite3.

SQLite3 performs major query optimizations during the process of query translation, as the "code generator" component handles both tasks simultaneously. While the blurring of program responsibilities usually does not present a problem, it prevents other components from accessing the optimized query plan; the code generator only outputs the optimized VDBE instructions, preventing access to an

optimized version of the input query plan. Thus, SQLite3's compartmentalization of query steps posed a significant design challenge for SQLite3/HE, and was one of the most important considerations for how SQLite3/HE works in tandem with SQLite3. Furthermore, as the instructions have already broken down the query plan into a number of discrete steps, it is difficult to recombine these components into a less optimized query plan. There is no feasible way to improve or reorient the existing level of query optimization, which prevents the application of analytics targeted optimizations after the query has been translated to VDBE instructions.

Additionally, optimizations are applied throughout the VDBE instruction generation process. Traditional optimizations such as constant propagation and push-down mechanisms are introduced before and during the VDBE instruction translation process. However, index-based optimizations occur later on, after the VDBE instructions are generated. Most notably is the usage of indices during nested loop joins, the join algorithm of choice for SQLite3. Of course, the decision to exclusively use nested loop joins creates additional problems for analytical queries. In general, while SQLite's optimizations are suitable for the general use case of embedded applications and related systems, some decisions that SQLite makes are contrary to the existing research involving high performance analytical query optimizations. These optimizations generally favor columnar methods with access patterns that are amenable to vectorization, further establishing the benefits of the presence SQLite3/HE's secondary columnar storage layer.

2.3.2 SQLite Optimizations

In addition to the aforementioned design choices, SQLite3 also utilizes a number of smaller optimizations that are relevant to analytical query acceleration. For example, the "record format" used to store each row makes a number of optimizations. One such optimization involves the header specifying the most efficient type possible for each value in the body of the record. Not only does SQLite3 include 24-bit and 48-bit integer types to assist in saving space, some types represent a field having an integer value of 1 or 0. In the value-specified type case, the body does not contain a

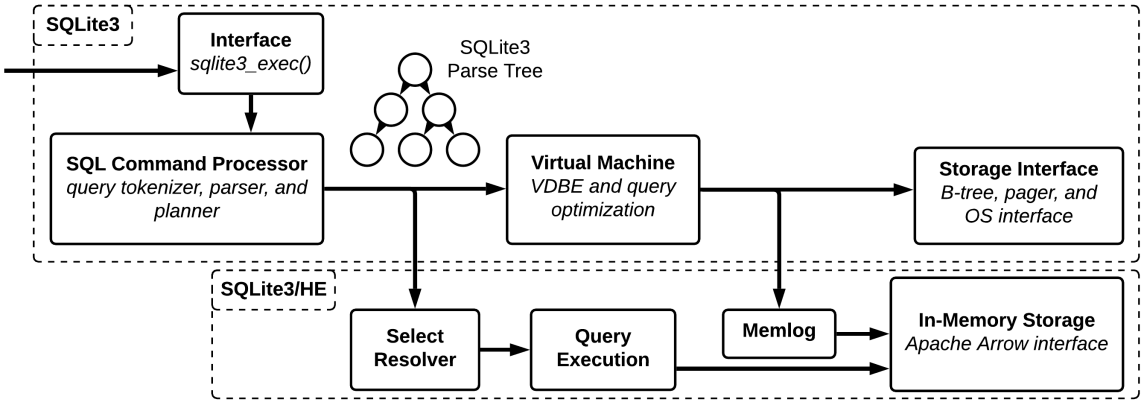

Figure 2.1: The SQLite3/HE Acceleration Path. After SQLite3 has generated the query parse tree, analytical queries are branched off from the existing SQLite3 execution path while transactional queries continue to be executed by SQLite3.

value; thus, it is possible for some records to have a header and no body.

SQLite's row format is part of a larger set of optimizations that reduce the size of values stored. Another example is the usage of variable sized integers elsewhere, encoded as the "varint" type. In general, SQLite3 makes a significant effort towards compressing the values it stores. This storage paradigm is effective in append-only, transaction-recording workloads; however, the added levels of interdiction and conversion causes significant amounts of overhead for reads. Thus, a new storage mechanism must be introduced to bypass this fundamental incompatibility with state-of-the-art analytical query optimizations.

2.4 SQLite3/HE Acceleration Layer

Given the previous exploration of SQLite3's design, we now have the context to understand the existing obstacles towards using SQLite3 as a high-performance database platform for analytical queries.

2.4.1 SQLite3/HE Integration

SQLite3/HE is implemented as an alternate query execution path, escaping the existing execution path between the SQLite3 Query Planner and VDBE Code Generator (Figure 2.1). SQLite3/HE functions primarily by providing an in-memory columnar database that is used as the foundation for analytical query acceleration. Due to this separate storage layer, a new set of operators were developed to facilitate query execution on a secondary query execution path, which we explore in detail in the following subsections.

Optional Acceleration Path

While Figure 2.1 depicts the acceleration path, the mechanisms that fully enable this behavior must be detailed at a finer degree. The SQLite3 query parser and planner behave as expected, with the addition of a limited set of optimizations that they apply such as nested sub-query flattening. These optimizations are all performed in-place on the parse tree, which can then be intercepted by SQLite3/HE before it is transformed into VDBE instructions. SQLite3/HE intercepts all queries before they are sent to the SQLite3 code generator unit. If the query is determined to be SQLite3/HE compatible (generally, a read/analytics-focused query), the query will instead be routed through the SQLite3/HE query acceleration path, and will no longer execute in SQLite3.

In particular, there are three rules that dictate if a query will be accelerated by SQLite3/HE:

1. The query is a supported SELECT or JOIN query.
2. All source tables are present in memory.
3. All operators used in the parse tree are supported by SQLite3/HE.

These rules allow for common analytical queries to be processed by SQLite3/HE. Parser flags for the SELECT query and the structure of the JOIN query determine if SQLite3/HE supports accelerating each respective query type. Specifically, JOIN

queries must fall under the general structural archetypes of "star-join" or "chain-join" based on how the tables are joined together. We utilize existing methods to both categorize and optimize these kinds of joins [30]. Once a JOIN query has been placed in the acceleration path, we utilize Lookahead Information Passing (LIP) [97] to optimize the JOIN query plan beyond the optimizations of SQLite3, resulting in a more robust and efficient JOIN query implementation.

Write queries are not supported by the SQLite3/HE acceleration path, and thereby fall back to the original SQLite3 query execution path.

In-Memory Columnar Storage Layer

SQLite3/HE utilizes Apache Arrow [8] as an in-memory columnar storage layer. This external dependency enables future cross-application compatibility, but is chiefly motivated due to Apache Arrow's strong implementation of efficient memory access and vectorization-friendly compute kernels. Thus, SQLite3/HE is able to focus on query optimization while still preserving all the benefits of a machine-optimized set of compute kernels.

Apache Arrow arrays are implemented as "chunked arrays," —arrays partitioned into a number of chunks of contiguous memory. This design choice enables a fairly simple process for resizing, either by deleting excess chunks or appending additional chunks. While the chunking of an array is accompanied by additional overhead, it also bolsters overall scalability and is an acceptable trade-off for the robust set of features that Apache Arrow brings. In addition, the chunked array approach facilitates an extra layer of parallelism from the perspective of operators, as allocating work based on chunks presents simple and efficient opportunities for parallelism.

Query Operators

SQLite3/HE's utilization of Apache Arrow for its storage layer allows for a significantly expedited engineering effort. Apache Arrow is primarily concerned with high-performance data storage and retrieval, providing a number of performant

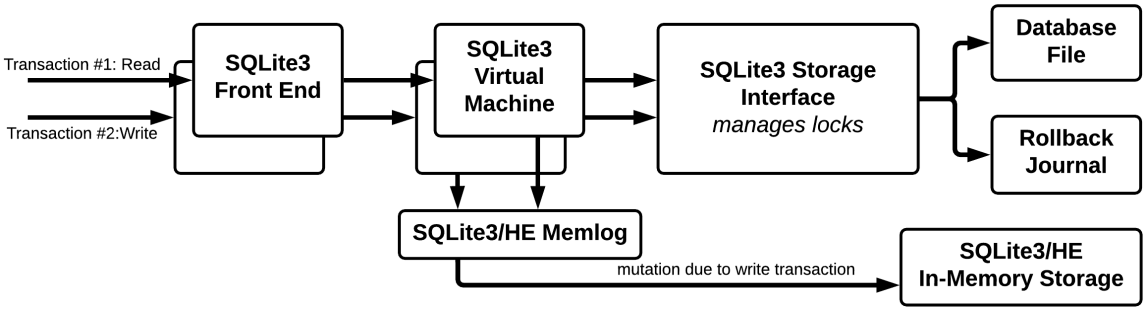

Figure 2.2: Transaction Flow in SQLite3/HE, depicting the movement of data during the processing of multiple concurrent queries.

compute kernels. By utilizing these kernels and the Apache Arrow API as a whole, SQLite3/HE capitalizes on the performance and long term support guarantees that Apache Arrow provides. Likewise, because of this existing body of work, the engineering effort behind SQLite3/HE has been able to focus on optimizing query operators, further improving SQLite3/HE as a whole.

2.4.2 Data Consistency

The implementation of a secondary storage layer requires mechanisms to maintain coherence between the SQLite3 and SQLite3/HE storage layers, especially in the case of the SQLite3 database receiving a write transaction. The primary mechanism in SQLite3/HE that facilitates this behavior is the "memlog."

The memlog records the writes that are made to the SQLite database file whenever transactions are committed by the SQLite3 query execution path. The memlog structure is depicted in Figure 2.2, within the context of two concurrent queries being processed at the same time. Note that due to SQLite3's database file-locking mechanisms, the memlog is presented with an ordered collection of changes to propagate from SQLite3 to the SQLite3/HE storage layer. Thus, SQLite3/HE is able to exploit the already existing ordering of mutations as they are made to the SQLite3 database.

The memlog is constructed as a collection of ordered write logs that must be propagated to SQLite3/HE columns, where each table is linked with a queue-like

structure to contain its write logs. Each write log includes information regarding the particular mode of the write (INSERT, UPDATE, or DELETE), the targeted row ID, and the relevant data if appropriate. These writes are accumulated over time, to eventually be processed in bulk depending on the mode of write propagation used by SQLite3/HE. We define these modes as "eager," "lazy," and "async background," each of which behaves as follows: eager writes perform updates during the transaction, lazy writes only make changes when the table is next read by a query, and async background writes use a secondary thread to perform a lazy write (as the worker threads also process SQLite3 lock contention, these background writes are considered asynchronous). We evaluate the behavior of each respective write mode as part of the evaluation of SQLite3/HE's transactional performance.

In the case of a crash or some other failure, the SQLite3/HE tables are regenerated from the SQLite3 database. This behavior is unavoidable due to SQLite3/HE's existence as an in-memory storage layer. That said, the reliance on SQLite3's error recovery mechanisms to regenerate data does provide the standard set of benefits which accompany a well-supported library.

2.5 Evaluation

Our evaluation of SQLite3/HE focuses on three primary questions:

1. What is the performance gain of SQLite3/HE for analytical queries?
2. Do any of the optimizations implemented carry over to transactional workloads? In particular, does the write propagation strategy have a significant impact on performance?
3. The secondary acceleration path incurs additional data movement costs when writing data to the SQLite3/HE columns –what is the overhead?

We utilized two different benchmarks as part of our evaluation: TATP [59] for write-focused, transactional workloads, and SSB [63] for read-focused, analytical

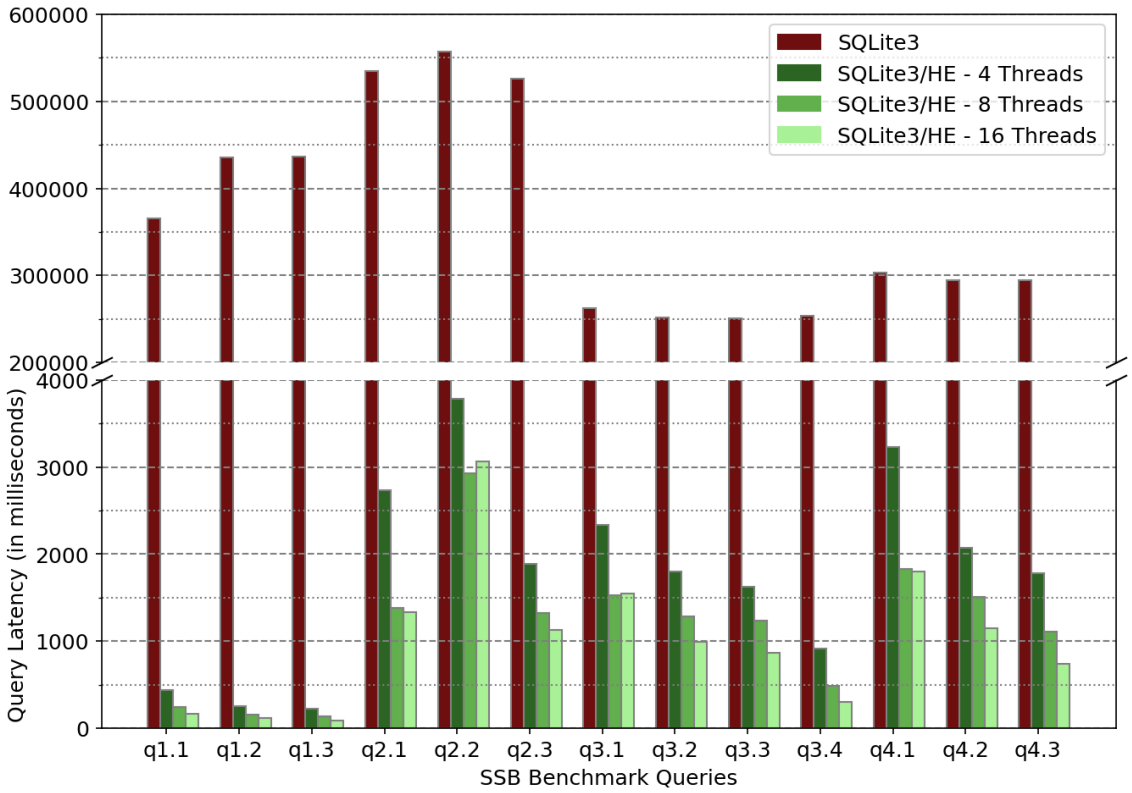

Figure 2.3: Analytical workload performance improvement due to Hustle, as compared between different amounts of thread-level parallelism.

workloads. All experiments were performed using SQLite3's default configuration parameters.

The experiments were run by a CloudLab "c220g5" machine [19]. This node consists of two Intel Xeon Silver 4114 processors each with a clock speed of 2.2 GHz. Each processor has 10 cores with two threads, for a total of 20 cores and 40 threads.

2.5.1 Analytical Workload Evaluation

We depict the SQLite3/HE results for each SSB query in Figure 2.3. SQLite3/HE completes all SSB (scale factor 10) queries in under 4 seconds, as compared to SQLite3 which requires at least 250 seconds at best. Query to query, SQLite3/HE

boasts a speedup consistently over 100x, though some queries are closer to a 1000x improvement. The previously mentioned bottlenecks of SQLite3 have a significant influence on the results. Likewise, these queries are where SQLite3/HE is able to fully utilize the cutting edge of analytical query optimization, such as its in-memory columnar store, operation vectorization, LIP, and intra-operator parallelism. As these features are not supported by SQLite3, SQLite3/HE's demonstrated performance gain is a reflection of its employment of state-of-the-art techniques. As SQLite3/HE supports thread-level parallelism, we evaluate SSB performance using 4, 8, and 16 threads. Unsurprisingly, the execution time of the queries is significantly reduced as more threads are utilized, due to large amounts of intra-operator parallelism. However, the speedup from additional threads has diminishing returns. As the provisioned CloudLab machine is a two-socket, 10 cores (20 threads) per socket system, we begin to see the impact of increased coordination and data movement between cores, exacerbated by the need for the CPUs to more regularly utilize L3 cache coherence mechanisms. This performance behavior is perfectly reasonable, as using a fixed thread count to divide the work into a number of partitions is expected to require some amount of optimization specific to both the workload and the underlying machine.

2.5.2 Transactional Workload Evaluation

To evaluate our integration with SQLite3 for transactional workloads, we use the TATP benchmark. In both TATP experiments, we use a database containing 2 million subscriber records and the recommended default parameters. We vary the percentage of write transactions present to generate a read-heavy workload (80% reads, 20% writes) and a write-heavy workload (50% reads, 50% writes). These results are depicted in Figures 4 and 5 respectively. Once again, we use SQLite3's query throughput as a performance baseline. As previously mentioned, all write transactions primarily interact with the SQLite DB file. After this initial write, we perform a secondary write to the SQLite3/HE storage at some (write mode-determined) point preceding the next read to the modified field. Due to the

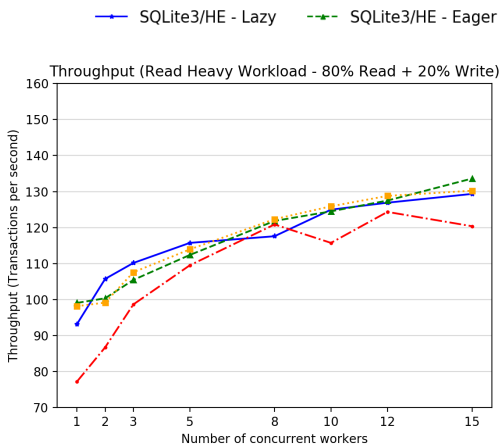

Figure 2.4: Transactional workload performance improvement due to Hustle, as compared between different write modes. This experiment utilized TATP in the default configuration.

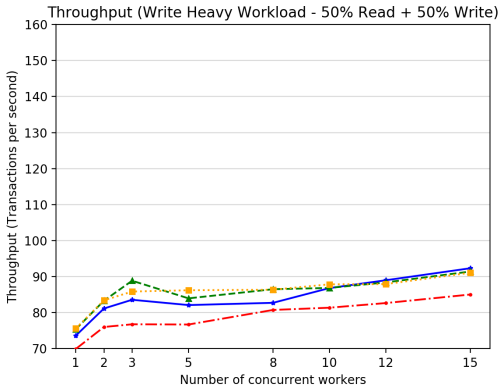

Figure 2.5: Transactional workload performance improvement due to Hustle, as compared between different write modes. This experiment altered the ratio of reads and writes in TATP, increasing the number of writes per read.

secondary write being performed, we expected the writes to have slightly higher latency compared to the naive SQLite3. However, given that reads in SQLite3/HE will be significantly faster than that in SQLite3, overall performance is expected to increase.

These expectations were justified, as SQLite3/HE offers a reasonable improvement over baseline SQLite3. While dependent on the number of workers and the particular write mode used, an improvement of up to 20% is possible. Most configuration and workload combinations will experience a 5%-10% improvement. Due to significant lock contention issues inherent in the SQLite3 write process, there is little room for performance gain in the presence of this heavy bottleneck, and thus the proportion of reads to the overall workload dictates the maximum expected performance improvement.

2.6 Discussion

Our evaluation demonstrates that SQLite3/HE accomplishes all performance improvements set forth. SQLite3 is purpose-built to handle transactional-query workloads in the resource constrained environment of an embedded device, and the SQLite3/HE acceleration path augments SQLite3 with state-of-the-art analytical query processing capabilities. By relaxing many of SQLite3's hardware-related constraints, SQLite3/HE is able to fully leverage the underlying hardware capabilities. This hardware-conscious behavior has become increasingly important as modern edge devices continue to diverge from the older class of traditional embedded devices.

A critical aspect of our design is the ability for SQLite3/HE to function as a drop-in replacement for SQLite3. As demonstrated by the TATP experiments, the combination of careful write management and significant read-performance improvements allow SQLite3/HE to be used as an analytical query accelerator in existing SQLite3 implementations at no performance cost. Likewise, SQLite3/HE's impressive analytical workload performance facilitates a wider variety of database-driven applications, such as on-site, live data analytics (as in the original windmill example).

Many modern databases have both transactional and analytical query processing needs. Significantly optimizing one category "for free" by more efficiently leveraging existing machine resources demonstrates SQLite3/HE's widespread applicability. A true cost that SQLite3/HE does impose on the host system is a slightly larger executable image size. While this is a valid consideration, the SQLite3/HE does not target applications and devices where minimizing executable image size is a primary concern.

2.7 Related Work

The use of secondary acceleration paths has become increasingly relevant in recent years across the breadth of computing research. These acceleration paths exist at

all levels, driven by advancements in both software and hardware [52; 69]. These paths consistently demonstrate the benefits of acceleration add-ons for existing systems, leveraging specialization to improve performance.

Improving the performance of an existing database structure to handle analytical queries has become a common "growing pain" within the field of database research. Significant amounts of prior work surround SQLite3/HE, each focusing on their own unique innovations [52; 72; 80]. Within this broad space, DuckDB [72] and SQLite3/HE target a similar need. While SQLite3/HE introduces an alternate query execution path, DuckDB implements much of the DBMS architecture "from scratch," opting to recreate or rely on external modules to facilitate their own query processing pipeline (such as *libpg_query* [25], the PostgreSQL Parser). DuckDB is an alternative approach to the overall problem that both SQLite3/HE and DuckDB address.

2.8 Future Work

While the benchmarks presented demonstrate wide-spread feasibility of SQLite3/HE, a larger discussion could be had on the inner workings of the implementation of SQLite3/HE's in-memory, columnar database. An examination into the particulars of how data consistency is achieved between the SQLite3 and SQLite3/HE databases would provide a deeper understanding of the true performance characteristics of SQLite3/HE's acceleration path. Specifically, while "write" and "read" are sufficient for the general case, evaluating the differences between INSERT, UPDATE, and DELETE operations would assist in the exploration of SQLite3/HE's low-level behavior.

2.9 Conclusion

Through the implementation of an alternative query execution pathway, SQLite3/HE augments SQLite3 with state-of-the-art analytical query processing capabilities, generating speedups at the 100x-1000x scale. These performance benefits come

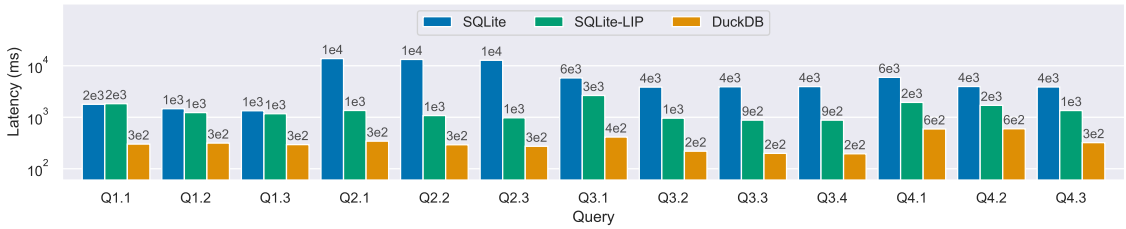

Figure 2.6: SSB query latency on a Raspberry Pi, evaluated using SQLite, SQLite with LIP, and DuckDB.

at no cost to the transactional query performance, enabling SQLite3/HE to function as a drop-in replacement for existing SQLite-powered applications. We view SQLite3/HE as a strong candidate for consideration in existing database platforms. As the technological needs of computing applications ever grows, so too do the performance demands placed on the underlying database systems. SQLite3/HE's implementation of state-of-the-art techniques satisfies these demands, advancing the ability for embedded systems to perform high-performance data analytics.

2.10 Continuation: Beyond SQLite3/HE

The paper has had a significant impact in the years following SQLite3/HE. While SQLite3/HE demonstrated that an in-memory cache format could improve data analytics tasks in SQLite, the reality of SQLite is that implementing an in-memory column store layer to accelerate analytics is a massive increase of complexity over the existing implementation of SQLite. Similarly, changing the underlying data format is entirely out of the question, partly because the SQLite file format is a recommended US Library of Congress standard because of its stability [29]. Thus, we required an alternative approach to improve analytics within SQLite.

The results of improving SQLite are in a separate paper [29]. Figure 2.6 showcases one of the notable results. These results were collected by running SSB on a Raspberry Pi 4 Model B, equipped with an ARM Cortetx-A72 CPU and other hardware typical of a low-power edge device. Overall, augmenting SQLite with LIP [97] speeds up the average SSB query latency by around 4.2x. While our exact implemen-

tation was not directly merged into SQLite (in line with the SQLite "Open-Source, not Open-Contribution" policy [35]), bloom filter-based query optimization was added to SQLite in version 3.38.0.

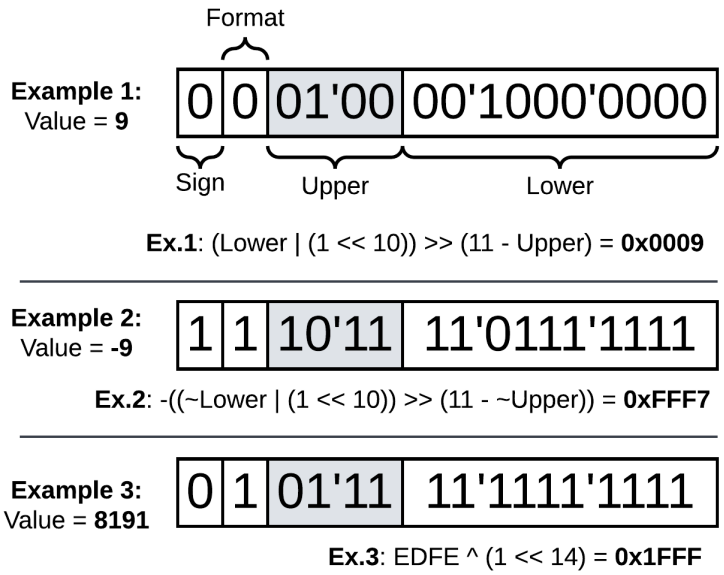

Figure 3.1: The values 9 (Example 1), -9 (Example 2), and 8191 (Example 3) encoded in EDFE, along with the process to decode each value back to an INT (explained in detail later) using C program syntax. We omit masking operations applied to the upper and lower fields to make the examples more concise. The upper field of each encoded value is highlighted in light gray. In Example 2 (-9), because the sign bit is set, the upper and lower fields are inverted during the decoding process, and the final output is negated. In Example 3 (8191), because the sign bit is cleared and the format bit is set, the only step to decode is to invert the format bit.

3.1 Introduction

A critical property of an integer data type is its binary representation, which has largely remained unchanged since the start of the computing field. The natural method to represent an integer in a machine-amenable format is to use a base-2 positional representation and then interpret the number using a signed notation, usually two's complement [44]. The two's complement representation has been

a crucial component of modern computing systems since it was proposed as part of the von Neumann architecture [86]. As we will see in this paper, this encoding plays a critical role in the performance of predicate-based columnar scan techniques – a common operation in data platforms that has received much attention in the community [3; 11; 24; 41; 42; 45; 51; 52; 64; 76; 89].

First, we observe that numbers may have multiple machine representations. As a more complex example than the aforementioned integer representation, consider the IEEE 754 floating point number representation. When read from left to right, it defines a set of discrete fields at fixed bit positions (sign, exponent, significand) [2].

Second, building on this observation, we reimagine the encoding of integers to more resemble a floating-point representation, but without losing precision or the natural ordering of integers. Our reimagined integers are also compatible with existing binary comparison logic, allowing their use as a replacement for existing integer types without significant changes to existing data processing kernels. Due to these properties, our alternative integer encodings are a significant departure from existing floating-point formats, as they do not call for dedicated floating-point processors [2; 55; 87].

One such method we consider is illustrated in Figure 3.1. This new format is called the **E**xtended **D**ata **F**orward **E**ncoding (EDFE). EDFE re-encodes a two's complement integer to minimize the number of leading zeroes. Thus, EDFE is particularly amenable to bit-parallel columnar-scan methods that rely on a property called *early pruning* to efficiently evaluate a predicate scan on an integer-typed column. EDFE extends the simpler **D**ata **F**orward **E**ncoding (DFE) by adding a sign and a format bit.

Example 1 in Figure 3.1 illustrates the EDFE decoding process (Section 3.3 presents the technique in full). To decode a value in EDFE to a two's complement integer, the value in the lower field is shifted as a function of the upper field. The bit pattern in the lower field is always able to reconstruct the entirety of the original value. This simplicity of encoding allows EDFE to be efficiently processed using simple bit-manipulation operations instead of relying on dedicated hardware, facilitating a lightweight encoding and decoding process that modern CPUs can

perform in a few clock cycles.

We emphasize the broad property of EDFE shifting bits towards the most significant bit (MSB). This property is common to all encodings in this family, which we identify as *forward encodings* (FEs). When using FEs instead of two's complement representations, the bits likely to distinguish between values are shifted toward the MSB. As shown in Figure 3.1, evaluating $-9 < 9 < 8191$ is represented in INT as `0xFFF7` $<$ `0x0009` $<$ `0x1FFF`, while in EDFE it becomes `0xEF7F` $<$ `0x1080` $<$ `0x5FFF`.

Forward encoding works particularly well with *bit-parallel columnar scan* methods [24; 42; 47; 51; 64; 76]. These methods often slice the bits in a sequence/column of integer values at the bit level and store the column by the bit position. For example, in both the classic bit-sliced index [64] and the BitWeaving/V [51] storage organizations, retrieving the first 64 bits of an integer column returns the 64 MSBs of the first 64 column values.

Predicate evaluation using bit-parallel organizations benefits from *early pruning*, which allows a scan to safely (early) terminate when sufficient bit slices have been evaluated to guarantee a correct result. The simplest case is that of an equality predicate. A bit-parallel method compares a slice of data bits with the corresponding slice in the predicate literal. If any data bits do not match, the corresponding column value (and hence the record) will not match the predicate, irrespective of additional data bits. However, integer columns often have many leading zeros, especially in the case of skewed datasets. These leading zeros delay early pruning, as more bit slices must be scanned before processing a prunable bit. Forward encodings are designed to address this weakness in two ways: First, the set bits from the original value are shifted toward the MSB. Second, the upper field captures the magnitude of a value while the remaining bits capture the details; as both fields can be used for early pruning, many scans prune early after evaluating the first few bits of the upper field.

The improved columnar scan performance of bit-parallel techniques is based on rearranging the bits of each value in a column. As the performance benefit of early pruning is based upon accessing a portion of the underlying value, this access pattern must be facilitated by the column's data layout. However, because of this

modified layout, retrieving a value from the column requires additional memory reads. Broadly, bit-parallel techniques improve columnar scan performance at the cost of fetch performance.

Previous bit-parallel scan acceleration techniques have primarily been evaluated using synthetic benchmarks that usually generate uniform-random data [63; 82]. While uniform-random data is useful for comparing techniques, it obfuscates the exact origin of scan performance improvements that are influenced by the underlying data. Is a uniform distribution of bits necessary for early pruning to function effectively? Does reliance on a uniform bit distribution necessitate that bit-parallel techniques must always be used to store integers using the two's complement representation? To approach these questions, we perform our evaluation using multiple skewed, real-world datasets, which we describe during our evaluation.

The encoding method of a value is orthogonal to the storage organization of a column. One could take a column/batch of integer values and slice them at the bit level. Indeed, this is what methods like bit-sliced indices [64] and BitWeaving/V [51] do. But, one could also slice at a different "vertical" boundary – such as the byte level. This latter approach is adopted by ByteSlice [64], the current state-of-the-art bit-parallel method. We evaluate our proposed encodings using BitWeaving/V and ByteSlice to understand the impact of the choice of encoding on multiple storage organizations.

Collectively, the contributions of this paper are as follows:

1. We propose reimagining the encoding of integer data types and present a new general method of encoding integers called forward encoding. Besides the specific use cases discussed in this paper, we intend for the paper to initiate a new way of thinking about the encoding of integer data.

2. We propose two new forward encodings, DFE and EDFE, and explore their theoretical properties.

3. We note the orthogonality between encoding and storage organization. In this framework, BitWeaving/V and ByteSlice use existing integer encodings

with bit-sliced and byte-sliced boundaries. We provide an intuitive method to characterize different ways of organizing the bits in a column of integers. Using this characterization, we also explore each storage organization's trade-offs between columnar scan and fetch operation performance.

4. We demonstrate the performance advantage of DFE and EDFE across multiple storage organizations using skewed, real-world data. Averaging across several cases, we observe geometric mean speed-ups of 1.47x and 1.33x for scan and fetch operations using the state-of-the-art bit-parallel technique (ByteSlice). The performance of BitWeaving/V is improved as well (1.55x and 1.19x), elevating it to be comparable with ByteSlice.

The remainder of this paper is organized as follows. First, we explore key background works in Section 3.2 to establish a set of standard parameters that describe bit-parallel techniques (Section 3.2.2). Next, we use these parameters to examine runtime discovered early stopping (a more precise definition of early pruning) in Section 3.2.3. We discuss related work in Section 3.2.4. Section 3.3 describes the properties of forward encodings (Section 3.3.1) before discussing the specifics of DFE (Sections 3.3.2) and EDFE (3.3.3). Then, we introduce a new concept called FE-enabled early stopping in Section 3.3.4. In Sections 3.3.6 and 3.3.7, we explore usage considerations and trade-offs between our proposed and existing encodings. We evaluate our proposed encodings in Section 3.4 and discuss the results of our experiments in Section 3.5. Finally, Section 3.6 contains our concluding remarks.

3.2 Background and Related Work

This section describes key bit-parallel techniques and identifies a set of standard parameters to compare bit-parallel methods. Using these parameters, we explore the critical components that make early pruning effective. We also explore a number of related works in this space, including hardware implementations of early pruning and alternative bit representations (codes) of integers.

3.2.1 Existing Bit-Parallel Techniques

Bit-parallel techniques have achieved significant reach since their inception [24; 42; 47; 51; 64; 76]. Broadly, these methods recognize that reorganizing the bits of multiple values can lead to increased storage and processing efficiency.

We consider the bit-sliced index [64] the starting point of many bit-parallel techniques. This initial technique was extended in a variety of directions [24; 41; 51; 76]. In particular, we emphasize the later contribution of *early pruning* [51]. Early pruning allows bit-parallel techniques to identify opportunities to compute results without accessing every bit of each processed value. The properties behind early pruning are crucial to the behavior of modern bit-parallel, predicate-based, columnar scan techniques.

We select two background works to emphasize fundamental mechanisms common to bit-parallel techniques: BitWeaving/V [51] and ByteSlice [24]. We choose these two particular techniques for two reasons: First, ByteSlice is the current state-of-the-art bit-parallel technique. Second, the authors of the ByteSlice paper evaluate ByteSlice against BitWeaving/V to demonstrate the differences between literal bit-parallel (bit width $b = 1$) and byte-parallel ($b = 8$) storage layers. These two techniques serve as a baseline to compare multiple forms of bit-parallelism when performing columnar scan and fetch operations.

As our proposed encodings are heavily influenced by early pruning, we explore the behavior of early pruning in a dedicated portion of this section. However, before we can thoroughly investigate early pruning in prior work, we need to establish a standard set of parameters to describe existing bit-parallel techniques. From this foundation, we can explore the impact of early pruning.

3.2.2 Generalization of Techniques

Prior bit-parallel works use different terms to identify their bit-parallel techniques. We unify the existing terminology using a precise set of terms, which we tabulate in Table 3.1.

Variable Name	V.	Value	Definition
Bit Width	b	Parameter	The bit-width of each data value in the original column. This parameter is data-dependent.
Group Size	g	Parameter	The number of segments of data values that are processed in parallel. This parameter is specified by the implementation of the bit-parallel technique.
Strata Width	s	Parameter	The smallest number of actionable bits per data value. This parameter is defined by the bit-parallel technique.
Strata Count	c	$c = b/s$	The number of strata that each data value is broken into.
Parallelism Size	w	$w = g \times s$	The number of data bits that are processed in parallel. This value is influenced by the available compute resources.

Table 3.1: Variables used to categorize bit-parallel techniques.

We define a bit-parallel technique as a technique that processes s bits of g values in a combined processing step. The parameter s identifies the "strata width," which indicates the number of bits processed per value in a parallel context. A "byte-parallel" technique specifically identifies a bit-parallel method with a strata width parameter $s = 8$. The parameter g (group size) identifies how many individual segments of data values are processed in parallel. We define g as an implementation detail of each method; in many cases, the theory behind the overall technique remains the same when g is modified.

When a bit-parallel method loads a column of data, it processes values with a bit width of b. All other terms that describe bit-parallel techniques can be derived from b, g, and s.

The number of strata each value is broken into is defined as the "strata count" $c = b/s$. A higher value of c allows for more early stopping (early pruning) opportunities during processing. However, higher values of c generally incur a higher overhead cost when recombining strata to restore the original value.

The number of bits processed in parallel at each step is the parallelism size $w = g \times s$. The parallelism size of bit-parallel techniques is generally implementation dependent, influenced by not only the same considerations that influence the group size g but also the available compute and storage resources.

Using this framework, we can systematically characterize various terms used to describe bit-parallel techniques.

"Bit-parallel," "byte-parallel," and "N-bit-parallel" all refer to a bit-parallel storage technique with a strata width s denoted by the name. We still use bit-parallel and bit-stratified as general descriptors for bit-stratified storage organizations. The strata width will be explicitly stated when necessary for clarity.

"Early stopping," "early pruning," and "early termination" all refer to an algorithm that stops before processing all bits in every processed value. Because the early stopping condition is evaluated between strata to determine if a stop can occur, we describe these behaviors as "runtime discovered early stopping." In contrast, an algorithm that determines the bit-location of an early stop before scanning begins has performed a "planned early stop."

We note that a stratum is intended as the smallest width of actionable bits in the context of early stopping. Strata width is usually, but not necessarily, linked to the particular implementation of the storage layer orchestrated by a bit-parallel method. For example, BitWeaving/V implemented using a "bit group size" of 4 is identified as having a strata width of $s = 4$, while ByteSlice has a strata width of $s = 8$ [24; 51].

The performance of columnar scans and fetches is significantly impacted by the strata width of a given bit-parallel technique. Reducing the strata width makes it possible to perform an early stop after processing a smaller number of bits. Depending on the group size parameter and the exact query, the average number of bits processed by the columnar scan may be reduced. However, reducing the strata width will usually increase the latency of fetching values from a bit-parallel column, as each stratum retrieved may require its own memory access.

3.2.3 (Runtime Discovered) Early Stopping

Having identified a standard set of parameters to describe bit-parallel techniques, we can now more deeply discuss the behavior of runtime discovered early stopping.

A bit-parallel technique processes g (group size) elements at each processing step. A discovered early stop can only occur when the early stopping condition for the entire group g is met. For all integer comparison operations, this condition

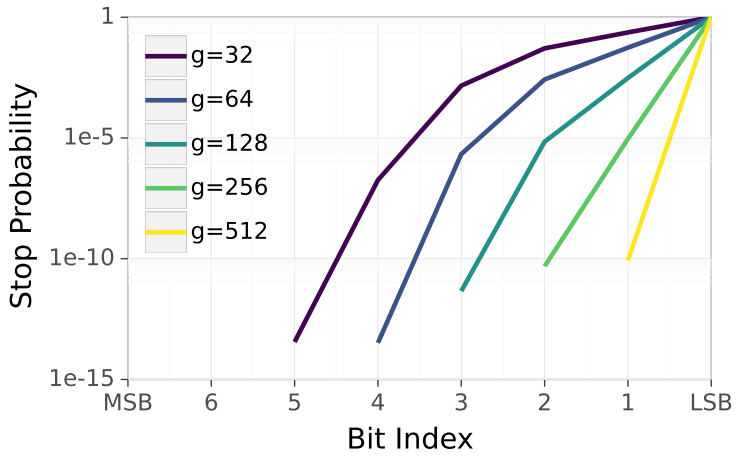

Figure 3.2: Stopping probabilities when performing a predicate-based scan on the SSB column LO_QUANTITY, using a value of 15, at multiple group sizes (g). A (non-early) stop occurs when all bits of a value have been processed.

is "data strata not equal to the corresponding predicate strata." A model of this behavior has been explored in previous work [24; 51], which we summarize here. Assuming a set of uniformly distributed values of b bits in length, the probability of a value to allow for early stopping after i_b bits have been evaluated is $\mathcal{P}(i_b) = 1-(\frac{1}{2})^{i_b}$. This behavior is plainly understood, as each additional bit examined will remove half of the remaining values. This construction can be expanded to include g and a "fill factor" term f, which corresponds to the fraction of values present compared to the total cardinality of the number of bits. This expanded form expresses the previous probability as $\mathcal{P}(i_b, g, f) = (1 - (\frac{1}{2})^{i_b})^{gf}$.

While this model is accurate, its assumptions carry a significant caveat that impedes its general case usage: The fill factor reflects a random removal of elements from the overall cardinality. In practice, fill factors are regularly skewed. For example, consider the LO_QUANTITY column from the Star Schema Benchmark (SSB) [63], which contains values in the range $[1, 50]$ and is stored using 6 bits. The skewed fill reduces the value of evaluating an early stopping condition at some bit indices. To represent this impact, we replace the $\frac{1}{2}$ term with $I(b)$, where I represents a function mapping the probability of bit b in the predicate being equal to bit b across

all values in the scanned column's data. For example, given the LO_QUANTITY column and a query value of 15, $I(b) = \{0.62, 0.62, 0.48, 0.48, 0.5, 0.5\}$. After this modification, our early stopping model becomes: $\mathcal{P}(b, g) = (1 - \prod_{i=1}^{b} I(b))^g$.

Our model allows for an arbitrary extension of the bit representation of values with "leading zero" padding, which are bits that do not allow for any discovered early stopping opportunities. For example, extending the LO_QUANTITY column from 6 to 8 bits impacts the original example of a query predicate value of 15 as follows: $I(b) = \{1, 1, 0.62, 0.62, 0.48, 0.48, 0.5, 0.5\}$. These leading zeroes pose a fundamental challenge to existing bit-parallel methods, as they require processing but do not contribute to runtime discovered early stopping opportunities.

To illustrate the impact of our early stopping model, we apply the query "LESS THAN 15" to the previously discussed LO_QUANTITY column sized at 8 bits. We vary the group size g and depict the results in Figure 3.2. Points below 1e−15 are omitted from the figure. In general, large values of g negatively impact runtime discovered early stopping. By the next to last bit (bit index 1), only 0.3% of values have been early stopped when using a parallelism group size of $g = 128$. Once large values of g are reached, runtime discovered early stopping rarely finds opportunities to stop, significantly reducing its overall impact.

This early stopping behavior has been identified by the authors of ByteSlice and serves as part of the motivation to use byte-parallel ($b = 8$) storage instead of smaller bit-parallel ($b < 8$) storage configurations [24]. We agree that, as is, there are significant limitations to runtime discovered early stopping when working with large group sizes g. These results motivate an analysis into techniques that can complement runtime discovered early stopping.

3.2.4 Related Work

Bit-parallel computing techniques are deeply intertwined with our proposed encodings [24; 42; 47; 51; 52; 64]. These techniques have resulted in several successful implementations [9; 28; 66; 73]. While we focus on two specific implementations of bit-parallel techniques in this work (BitWeaving/V [51] and ByteSlice [24]), our

generalization of bit-parallelism and the concept of a bit-stratified storage layer is a broadly applicable framework that could be applied to many existing bit-parallel implementations.

Further, recent work has investigated specialized hardware designed to perform bit-parallel compute [49; 77; 89; 90; 91; 96; 98]. Some of these recent works have implemented bit-parallel techniques within compute-capable memory units [49; 77; 90]. We find that our generalization of bit-parallel techniques can accurately describe these methods. As these techniques operate on parallelism sizes (w) significantly wider than those used by CPUs, our analysis of runtime discovered early stopping (Section 3.2.3) is also pertinent to these techniques.

Application-specific encodings have found success in a variety of environments [20; 26; 31; 50; 55; 74; 87]. These encodings were proposed to address the needs of a specific application, just as DFE and EDFE are focused on improving bit-parallel techniques when processing skewed data. Further, methods that automatically select an optimal type or encoding amplify the benefits of additional encoding options [37; 41].

3.3 Encoding

In this section, we explore the motivations behind the design of forward encodings (FEs). Then, we propose two FEs: Data Forward Encoding (DFE) and Extended Data Forward Encoding (EDFE). DFE is intended as a simple implementation of a forward encoding to demonstrate the properties of an FE. In contrast, EDFE is usable as a general-purpose encoding that extends the DFE. After describing the encodings, we explore how forward encodings enable new opportunities for early stopping. Finally, we examine the trade-offs between existing representations of integers, DFE, and EDFE.

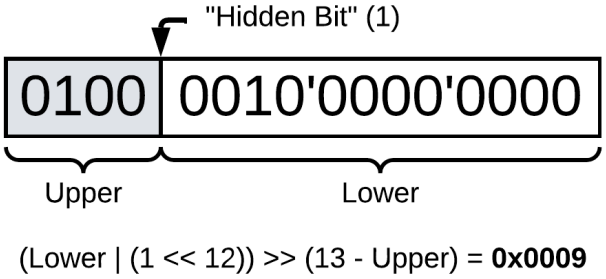

Figure 3.3: The value 9 encoded in DFE. The upper field is highlighted in light gray. Note that the natural bit representation of 9 is `1001`, which is present (shifted) in the lower field once the hidden bit (1) is made explicit.

3.3.1 Forward Encodings

In this section, we explore the concept of forward encodings. Then, we demonstrate how forward encodings enable a new set of early stopping opportunities.

A forward encoding (FE) is broadly intended to shift the bits that could be used for runtime discovered early stopping towards the most significant bit. Our design of forward encodings takes significant inspiration from floating-point formats that demarcate regions of a longer word as independent fields. However, in contrast to floating-point formats, FEs are still fundamentally integer encodings: forward encodings maintain full number precision and compatibility with integer comparison operations. Thus, while FEs share properties with floating-point encodings, they have a fundamentally different set of behaviors.

Previously, we demonstrated decoding several values from EDFE to their INT form (Figure 3.1). We use one of these values (9) as an example to showcase the decoding process of the data forward encoding (DFE). The DFE of 9 is shown in Figure 3.3. Note that the DFE of 9 has an upper field value of $\mathtt{upper} = 4$.

First, the bit pattern representing 9 (`1001`) is always readily available once the hidden bit (1) is made explicit. This allows for a simple decoding process: $(\mathtt{lower}|(1 \ll 12)) \gg (13 - \mathtt{upper}) = 9$.

Second, we note that the upper field decremented by one ($\mathtt{upper} - 1 = 3$) is the number of *salient* bits (starting from the MSB) in the lower field: Outside of these first three bits, all other bits will be shifted out during the decoding process

and thus have no impact on the decoded value. Thus, in effect, DFE has trailing zeroes instead of leading zeroes.

The trailing zeroes of DFE are common to all forward encodings. FEs have *non-impactful trailing zeroes* instead of *impactful leading zeroes*. In the context of bit-parallel techniques, this has a profound impact: Information from the upper field can be used to set a maximum bound on the total number of strata that must be processed. Further, this maximum bound can also be used for comparisons, allowing the scan predicate to apply a maximum strata boundary on a columnar scan (described in Section 3.3.4). In the following sections, we explore how DFE and EDFE implement the properties of forward encodings.

3.3.2 Data Forward Encoding

In this section, we present the Data Forward Encoding (DFE). The algorithm to encode a value from an integer to its DFE encoded form is shown in Algorithm 1. We include examples of encoded values in Table 3.2.

Before we sketch our proof for Algorithm 1, we walk through an example of encoding and decoding. We also use this opportunity to link DFE to the properties of forward encodings illustrated previously.

We present the DFE representation of the value 9 in both Figure 3.3 and Table 3.2. Before we explain DFE, consider a simple idea: Imagine shifting a non-zero UINT

Value	INT	DFE	EDFE
8191	0001111111111111	1101111111111111	0101111111111111
2048	0000100000000000	1100000000000000	0100100000000000
2047	0000011111111111	1011111111111100	0010111111111111
9	0000000000001001	0100001000000000	0001000010000000
3	0000000000000011	0010100000000000	0000101000000000
2	0000000000000010	0010000000000000	0000100000000000
1	0000000000000001	0001000000000000	0000010000000000
0	0000000000000000	0000000000000000	0000000000000000

Table 3.2: Examples of integer values and their 16 bit representations using INT, DFE, and EDFE. The upper fields of DFE and EDFE are highlighted in light gray.

Algorithm 1: DFE Encoding Algorithm

input: Integer n represented using b bits.

```
1  if n = 0 then
       /* Encoding zero.  Return zero.                          */
2      return 0;
3  else
4      clz ← CountLeadingZeroes(n);
5      len(u) ← ⌈log2(b)⌉;
6      len(l) ← b − ⌈log2(b)⌉;
7      if clz < len(u) − 1 then
           /* Number out of bounds for encoding.                */
8          throw ErrorOutOfBounds;
9      else
           /* Upper field records the shift.                    */
10         u ← (b − clz) ≪ len(l);
           /* Lower field preserves the value.                  */
11         l ← n ≪ (clz − len(u) + 1);
12         l_mask ← (1 ≪ len(l)) − 1;
           /* Combine fields with a bitwise OR.                 */
13         return u | (l & l_mask);
14     end
15 end
```

(unsigned integer) value to the left until the MSB of the shifted value is `1`. As the shifting process has guaranteed that this (new) MSB bit is always `1`, we can assume this bit's value without storing it.

This process is precisely what occurs when encoding a value in DFE. The original value is shifted so that the first set bit is aligned with the MSB of the lower field. As this bit is always known to be `1`, it can be omitted. Thus, for the DFE value of 9, by including the "hidden bit," we see that the base-2 representation of 9 `1001` has been shifted to the MSB of the lower field. By hiding this bit, we double the representable range.

Conceptually, the hidden bit resides between the lower and upper fields. In this context, the value stored in the upper field is similarly intuitive: the upper

field records the number of salient bits in the lower field, starting with the hidden bit. Consider the example of encoding 9: the upper field has a value of 4; thus, the lower field and hidden bit together are 4 bits long. The first three bits of the lower field are `001`, which, when augmented with the hidden bit, becomes `1001`. It is trivial to decode 9 in DFE back to the original INT encoding through the use of bit manipulation: $(lower|(1 \ll 12)) \gg (13 - upper) = 9$.

To support Algorithm 1, we provide sketches of proofs that identify the key components of this alternative encoding.

Definition 3.1. *A value of* 0 *has a DFE of* 0*.*

Theorem 3.2. *Given a word length of* b *bits, DFE is a reversible encoding for all values* n *where* $n \leqslant 2^{b-\lceil \log_2 b \rceil + 1} - 1$ *and* $n \geqslant 0$*.*

Sketch of Proof. Given a value n that is represented using a word length of b, we note that $\lceil \log_2 b \rceil = len(u)$ is the number of bits required to represent b. In this construction, $len(u)$ bits are used to express the "upper" field (u) using a subset of fixed position bits in the existing word. The rest of the word is used for the "lower" field (l), the length of which is $b - len(u) = len(l)$. The values of u and l are stored using $len(u)$ and $len(l)$ bits from the word, respectively. We place these bits at positions relative to a word of size b using a few logical shift operations.

$u = (b - CLZ(n)) \ll len(l)$

$l = (n \ll (CLZ(n) - len(u) + 1))\&((1 \ll len(l)) - 1)$

$DFE(n) = u|l$

As also seen in Algorithm 1, prior to combining u and l into a single size b value, we apply a mask to l. This mask removes the leading bit of l; per Definition 3.3.1 and the behavior of the CLZ function, this removed leading bit is always 1. DFE can represent values of up to length $b - len(u) + 1 = len(l) + 1$, equal to a maximum value of $2^{b-\lceil \log_2 b \rceil + 1} - 1$.

To decode a value in DFE, we determine a shift and a value by reversing the process to construct the upper and lower fields.

$n_s = DFE(n)\&((1 \ll len(l)) - 1)|(1 \ll len(l))$

$n_v = (len(l) + 1) - (DFE(n) \gg len(l))$

$n = n_v \gg n_S$

Note that n_v is restored bit pattern of the original value n: applying the shift n_s to n_v results in n.

We require that all shifts are logical shifts that shift in zeroes. If an arithmetic shift is used, then masks must be applied to set the shifted-in bits to zero.

As n has been defined to have an upper bound of $2^{b-\lceil \log_2 b \rceil + 1} - 1$, n can always be represented using $len(l) + 1$ bits.

The value n has never been lost, only shifted around within the existing b bits of the word. This behavior demonstrates that DFE is as much of a bit-representation remapping as it is a unique encoding. This mapping purposefully reduces the representable space of values to create additional early stopping opportunities closer to the MSB. As the mapping has been demonstrated to be reversible for values within the predefined bounds, DFE is a reversible encoding within the given bounds of n. □

Theorem 3.3. *Given two values* x *and* y *and the DFE encoding function* $DFE()$*, for any comparison operation* $\bullet \in \{<, \leqslant, >, \geqslant, =, \neq\}$*,* $DFE(x) \bullet DFE(y) \leftrightarrow x \bullet y$.

Sketch of Proof. First, we note that we have defined the encoded value of 0 to be 0. We identify the encoded value of 0 as our base case.

As u is constructed using $b - CLZ(n)$, we note that as $CLZ(n)$ decreases, u increases (Theorem 3.3.2). Further, l is constructed using a shift of $CLZ(n) - len(u) + 1$. By observation, incrementing n by one has one of two effects: Either u is incremented by one and $l = 0$ (as the only set bit in l is now the hidden bit) or the pre-shift value of l is incremented by one. Given these cases, we see that the sequence of possible values of $DFE(n)$ forms a sequence of monotonically increasing numbers, provided that the consecutive values of n are also monotonically increasing. Thus, for any comparison operation $\bullet \in \{<, \leqslant, >, \geqslant, =, \neq\}$, $DFE(x) \bullet DFE(y) \leftrightarrow x \bullet y$. □

We note that DFE is an encoding that can be used with various bit widths. The significant change between each width is the size of the upper field ($\lceil \log_2(b) \rceil$), as

the upper field must contain enough bits to fully record any applied shift for the bit width in question. For example, DFE16, which is used in Table 3.2, has an upper field size of 4, while DFE32 would have an upper field size of 5.

Further, while the upper field must meet a minimum size requirement to support lower fields of increasing size, there is no requirement that the combined field size is always a machine word. This leads to the ability to form columns encoded in DFE9, where the last 7 bits of the lower field of DFE16 have been removed.

Bit packing at these exotic bit sizes is common in many storage formats, including Parquet [6], making the FE methods compatible with existing bit-based methods that store values using arbitrary bit sizes based on the representation needs of the stored column.

3.3.3 Extended Data Forward Encoding

As identified in Algorithm 1, DFE cannot represent values with fewer than $\lceil \log_2(b) \rceil - 1$ leading zeroes or negative numbers.

To remedy these shortcomings, we propose the "Extended Data Forward Encoding" (EDFE) as an extension to DFE. The EDFE addresses these issues by modifying the encoding process. This algorithm is shown in Algorithm 2.

Examining Algorithm 2, we see many similarities with the process to encode values in DFE (Algorithm 1). Table 3.2 and Figure 3.1 both show the result of encoding 9 in EDFE. This encoded form is intentionally similar to the encoding of 9 in DFE. EDFE includes two extensions to DFE: a "sign indicator" bit and a "format" bit.

To give an intuitive understanding of these two additions, we explore some of the values in Figure 3.1 (some of which are also included in Table 3.2). We emphasize how the additions to EDFE preserve its integer comparison functionality.

First, we examine the EDFE of 8191. For values that require more salient bits than what can be represented using the lower field, we bypass the encoding process and toggle the format bit. In this case, decoding is trivial: the format bit is inverted, restoring the integer to the original two's complement representation. As setting

Algorithm 2: EDFE Encoding Algorithm

input: Integer n represented using b bits.

```
1  if n = 0 then
       /* Encoding zero.  Return zero.                                  */
2      return 0;
3  else
4      abs ← AbsoluteValue(n);
5      clz ← CountLeadingZeroes(abs);
6      len(u) ← ⌈log2(b)⌉;
7      len(l) ← b − ⌈log2(b)⌉;
8      if clz ⩽ 1 then
           /* Number out of bounds for encoding.                        */
9          throw ErrorOutOfBounds;
10     else if clz ⩽ len(u) then
           /* Large number case.  Invert format bit (using an XOR
              operation).                                               */
11         return n ^ (1 ≪ (b − 2));
12     else
           /* Small number case.  Similar to DFE.                       */
13         u ← (b − clz) ≪ (len(l) − 2);
14         l ← abs ≪ (clz − len(u) − 1);
15         l_mask ← (1 ≪ (len(l) − 2)) − 1;
16         if n < 0 then
               /* Negative Number case.  Invert bits.                   */
17             return ~(u | (l & l_mask));
18         else
19             return u | (l & l_mask);
20         end
21     end
22 end
```

the format bit is based on a value threshold and the format bit is placed before the upper field, comparison functionality is preserved.

Next, we examine the values for 9 and -9. The EDFE encoded value of -9 is the bit inversion of the EDFE of 9. This is intentionally similar to the ones' complement

integer representation, as the bit inversion extends the comparison properties of the positive number range to the negative number range [44].

As EDFE represents two significant changes from the DFE, it requires a separate proof sketch.

Theorem 3.4. *Given a word length of* b *bits, EDFE is a reversible encoding for all values* n *where* $-1 \times (2^{b-2} - 1) \leqslant n \leqslant 2^{b-2} - 1$ *that obeys the comparison behavior of DFE (for any comparison operation* $\bullet \in \{<, \leqslant, >, \geqslant, =, \neq\}$*,* $\mathrm{DFE}(x) \bullet \mathrm{DFE}(y) \leftrightarrow x \bullet y$*).*

Sketch of Proof. EDFE extends DFE. This extension is implemented via a right shift by 2 on the upper and lower fields, creating space for two additional bits. The most significant bit is identified as the "sign indicator" bit. The bit following the most significant bit is identified as the "format" bit. Both the sign indicator bit and the format bit are cleared for positive values small enough to be encoded similarly to DFE.

If a value is negative, the EDFE is computed using the magnitude of the value to encode. As the last step in the encoding, all bits of the encoded value are inverted, setting the sign indicator bit. In this way, the sign indicator bit allows for compatibility with signed integer comparison hardware.

If a value's magnitude is too large for a DFE-like encoding (i.e. $\mathrm{CLZ}(\mathrm{ABS}(n)) \leqslant \mathrm{len}(u)$), the value is encoded by inverting the format bit. As this is a magnitude-based threshold, EDFE values still form a monotonically increasing sequence, preserving the properties of the DFE for comparisons.

To prove that EDFE maintains the comparison properties of DFE, we sketch a proof by exhaustion. There are four cases, based on both the sign and the magnitude of the value to be encoded.

Case 1: Positive, inactive format bypass (Algorithm 2, lines 12, 19). This case is the DFE-like case; no behavior is changed.

Case 2: Positive, active format bypass (lines 10, 11). As the format bit is closer to the MSB than the upper field, integer comparisons interpret format-set EDFE values as larger than format-unset EDFE values. As the format bit is set by a threshold

based on magnitude (via the leading zero count), it holds true that format-set EDFE values will always be larger than format-unset EDFE values.

Case 3: Negative, inactive format bypass (lines 12, 17). We note that the bit-inversion step is shared between EDFE and the process to encode values as a ones' complement negative integer. Due to this similarity, using the MSB as the sign bit allows for using existing signed integer comparison hardware.

Case 4: Negative, active format bypass (lines 10, 11). As the value is already a negative integer (MSB is set), toggling the format bit applies the same thresholding as before. This toggling still obeys integer comparison rules, as the toggling is based on the magnitude of the value: all EDFE values where the format bit has not been toggled (is set for negative values) are greater than EDFE values where the format bit has been toggled (is not set for negative values).

Thus, while the feature set of the DFE has been expanded into the EDFE, all of the previously existing properties regarding the correct semantics of comparison operations are preserved. □

3.3.4 FE-enabled Early Stopping

The upper field can be used to generate two kinds of stopping opportunities: planning a stop before performing a predicate-based scan and discovering a stop while fetching a value.

FEs exchange the padded leading zeroes for trailing zeroes, a behavior seen in Table 3.2. However, these zeroes are not equivalent: Trailing zeroes in DFE do not impact the result of comparisons. This is the foundation for our definition of FEs having *non-impactful trailing zeroes*, as opposed to the *impactful leading zeroes* present in existing integer encodings.

For any value encoded in DFE, $l_t = len(u) + u - 1$ bits, starting from the most significant bit, must be processed to reconstruct the original value. We reiterate that u contains the number of salient bits in the lower field (starting with the hidden bit). Thus, $u - 1$ is the number of bits that must be read in addition to the upper field. For any two values a and b in an FE, they can be compared using $min(l_t(a), l_t(b))$

bits. In the case of a columnar scan, l_t can be calculated for the predicate before any values are loaded, allowing a bit-parallel scan to plan how many strata must be processed. This behavior enables the planning of early stops before a scan begins, reducing the number of strata evaluated without relying on a runtime discovered early stopping condition.

While impactful leading zeroes cannot be skipped without impacting the result of a comparison, non-impactful zeroes will not change the result of a comparison, regardless of whether or not they have been processed. This property is the basis for *FE-enabled early stopping*, which is complementary to the existing early stopping technique. FE-enabled early stops can be applied to both predicate-based scans and fetch operations. First, the FE-enabled stop for scans is implemented as a *planned stop*, where the predicate value determines a stop location using l_t before the scan begins. Second, the FE-enabled stop for fetches is implemented as a runtime discovered early stop with a modified stopping condition: l_t determines the maximum number of strata required to reconstruct a value. We explore both kinds of FE-enabled early stops in this section.

First, we present an example of an FE-enabled planned stop when evaluating a predicate-based columnar scan. Consider a column containing uniform-random distributed values $\{0, 1, 2, 3, 9\}$ encoded in DFE, where we apply a filtering operation of "LESS THAN 2." The minimum number of bits to reconstruct a value in DFE is $l_t = len(u) + u - 1$, as we must read both the upper field and all salient bits of the lower field. As seen in Table 3.2, our predicate value of 2 has an upper field of $u = 2$ and thus requires $len(u) + 1 = 5$ bits to represent the value fully in 16-bit form. However, for a comparison between any two DFE encoded values a and b, the number of bits required from each value is $\min(l_t(a), l_t(b))$. For example, when comparing our predicate value 2 against a data value of 9, only the first $\min(5, 7) = 5$ bits from each value are required to perform the computation. While 9 is not fully representable using only 5 bits, existing binary comparisons will still identify that $00100 < 01000$, leading to a correct comparison result. Similarly, using 7 bits to represent the value 2 does not change the comparison result ($0010000 < 0100001$). This example showcases the core property of non-impactful trailing zeroes and

how the usage of DFE/EDFE allows for reducing the reliance on discovered early stopping to reduce the number of processed strata.

We now show how the properties explored by the example can be broadly applied to create additional early stops in bit-parallel predicate-based scan and fetch operations.

Given some data array d encoded in DFE16 ($b = 16$) and the query $d > c$, we iterate over each element in d. Due to the properties of non-impactful trailing zeroes, we know that the maximum number of bits required to perform all comparisons is $l_t(c)$. In the case of DFE16 ($len(u) = 4$) and $c = 2$ ($l_t = 5$), 5 bits of each 16 bit value are processed. In contrast, $c = 9$ determines that 7 bits of each value are processed. We identify this behavior as predicate-determined early stopping, where the maximum number of bits to process per value is known before the scan begins.

Minimizing the number of loaded bits can also be adapted to fit fetch operations, using a runtime discovered early stop, albeit in a different form than existing runtime discovered early stops. By loading u before loading any bits in l, the number of l bits to be read can be determined during the fetch process. As $len(u)$ and $len(l)$ are determined by the type of the array, a bit-parallel fetch can be broken down into a set of required reads and optional reads depending on the strata width, where the optional reads are performed based on u. This process also functions when reading multiple u in parallel. For example, an upper bound of u across many values can either be calculated at runtime (possibly by applying a vertical bit-wise OR across a group of u values) or be stored as metadata during column construction. As only trailing zeroes reside after the salient bits of a forward encoded value, reading a few more bits of each value, while not perfectly efficient, still reduces the average number of bits processed per value.

3.3.5 Encoding/Decoding Implementation

The algorithms that describe the encoding process for (E)DFE are purposefully designed to map directly to modern CPU instructions. Using scalar operations,

GCC 12.3 compiles decoding DFE32 to INT27 to 9 instructions (4 if the zero branch is taken). Decoding EDFE32 to INT31 is 10, 17, or 18 instructions depending on branching (format, sign). Encoding INT27 to DFE32 is at most 21 instructions, and encoding INT31 to EDFE32 is at most 26. Further, these scalar kernels can be directly mapped to SIMD; for example, the AVX512 instructions that perform "SRLV"/"SLLV" and "LZCNT" operations can be directly applied to the discussed kernels.

3.3.6 Usage of FEs

Forward encodings are lossless encodings. To facilitate a lossless transformation, the upper field requires allocating additional bits. For example, a column that requires 28 bits to be represented as a two's complement integer would require 32 bits in DFE and EDFE. The additional bits of overhead are used by the upper field to store the salient bit count before the data bits, which benefits bit-parallel scan and fetch operations on skewed data.

The upper field must grow to accommodate the number of salient bits (Section 3.3.2). However, this leads to smaller integer widths allocating a proportionally larger fraction of their overall size to the upper field (DFE8: $\mathtt{len}(\mathtt{u}) = 3$), while larger integer widths require proportionally fewer upper field bits (DFE64: $\mathtt{len}(\mathtt{u}) = 6$). The format bit in EDFE assists in mitigating this drawback.

3.3.7 Trade-offs of FEs

DFE and EDFE preserve the validity of existing comparison techniques. Techniques based on these properties, such as existing bit parallel work, can perform both scan and fetch operations using DFE and EDFE without technique-level modifications.

Comparing DFE and EDFE, we find that the extensions that EDFE provides come at the cost of increasing encoding and decoding complexity. However, supporting signed values is necessary for usage outside of constrained environments. Thus, while the DFE is more performant in applicable circumstances, these situations may not align with the needs of an application looking to better leverage bit-parallel

techniques. Similarly, DFE reduces the number of scanned bits compared to EDFE in situations that do not benefit from the usage of a format or sign bit. Thus, we find that both encodings have their own respective use cases depending on the needs of the target application. One could imagine a single system using existing integer encodings, DFE, and EDFE on the same platform, where the system selects the most appropriate encoding to use in each situation.

The concept of automatic typing and data encoding is well explored by other work [37; 41] and is broadly related to block-based storage layers [6; 13; 75]. There has been a recent surge in block-based storage formats, such as Parquet [6] and Arrow [75]; even one of the datasets we use as part of the evaluation of DFE and EDFE is distributed using the Parquet format [61]. One could envision that the choice of encoding (INT, DFE, or EDFE) is added as block-level metadata. As DFE and EDFE can be used with a variety of bit widths, these alternative encodings can be freely applied on top of existing data. Further, in data systems where in-place updates are allowed, a block could be converted between encodings if beneficial.

FE-native arithmetic is expensive when using integer hardware; a limitation shared by floating-point representations. However, encoding to (and decoding from) FEs is demonstrably lightweight (see Section 3.4.6), minimizing the need for dedicated FE compute units.

Finally, we recognize that using an alternative integer representation is a significant departure from existing computing ideas. Given how the existing integer implementation has not changed in over 50 years [86], this change opens a new set of challenges. However, we emphasize that forward encodings are intended for use when deemed beneficial by the data platform, similar to other alternative encodings used by modern systems [26; 50; 55; 87].

3.4 Evaluation

Throughout this paper, we have emphasized that DFE and EDFE are intended to complement existing bit-parallel data processing techniques. We selected two background works to establish a core set of properties shared between bit-parallel

techniques (Section 3.2.1). These background techniques use different strata widths ($s = 4$ for BitWeaving/V using a bit group size of 4, and $s = 8$ for ByteSlice). We compare these two bit-parallel methods with a columnar baseline, which we implement using a contiguous in-memory array. Consistent with existing results, ByteSlice outperforms BitWeaving/V in all cases [24]. However, as our results will demonstrate, both DFE and EDFE can significantly reduce the performance gap between these two bit-parallel techniques.

Our experiments aim to address the following questions:

Q1. What is the performance benefit of using forward encodings to accelerate bit-parallel scans? How impactful is setting an early stopping bound using l_t? (Sections 3.4.2 and 3.4.3.)

Q2. What is the fetch performance of forward encoded columns? Does the salient bit count optimization outweigh the cost of decoding values? (Section 3.4.4.)

Q3. What is the impact of the usage of forward encodings on the compressibility of a column? (Section 3.4.5.)

Q4. What is the cost to encode to or decode from an FE? (Section 3.4.6.)

Each experiment shares several standard parameters, which we identify in the remainder of this section.

Our experiments were run on a CloudLab c220g5 machine [19]. All experiments were run using a single CPU (an Intel Xeon Silver 4114). Unless otherwise stated, our experiments use all 20 threads of the CPU. All datasets used in our evaluation fit in the available main memory of this machine (192 GB). Each experiment was run ten times to generate an average result. The variance between experiments was negligible; thus, to aid readability, we do not include error bars in our figures.

3.4.1 Datasets

To evaluate the impact of forward encodings, we perform a series of microbenchmarks on selected columns from three real-world datasets: individual contributions

Column	Units	90%	99%	99.9%	99.99%	Max	INT	DFE	EDFE
FEC: Transaction Amount (Amt)	Dollars	112	2k	5.8k	100k	125M	27	31	32
Home: Milliwatts per Half Hour (mW)	Milliwatts	481k	1.49M	2.8M	4.67M	10.76M	24	28	30
Taxi: Trip Distance (Dist)	0.01 Miles	896	2.02k	2.92k	5.8k	38.97M	26	30	32
Taxi: Fare Amount (Fare)	Cents	3.1k	6.46k	11.85k	25k	40.11M	26	30	32
Taxi: Tip Amount (Tip)	Cents	575	1.49k	2.5k	6k	140.02k	18	22	24
Taxi: Tolls Amount (Tolls)	Cents	0	655	1.9k	2.83k	91.19k	17	21	23
Taxi: Total Amount (Total)	Cents	4.33k	8.18k	14.43k	28.44k	40.11M	26	30	32

Table 3.3: The evaluated columns from each dataset. The 90th, 99th, 99.9th, and 99.99th percentile values of each column and the maximum value per column are included. We also include the minimum number of bits required to represent each column using each evaluated encoding (INT, DFE, EDFE), based on each column's maximum value.

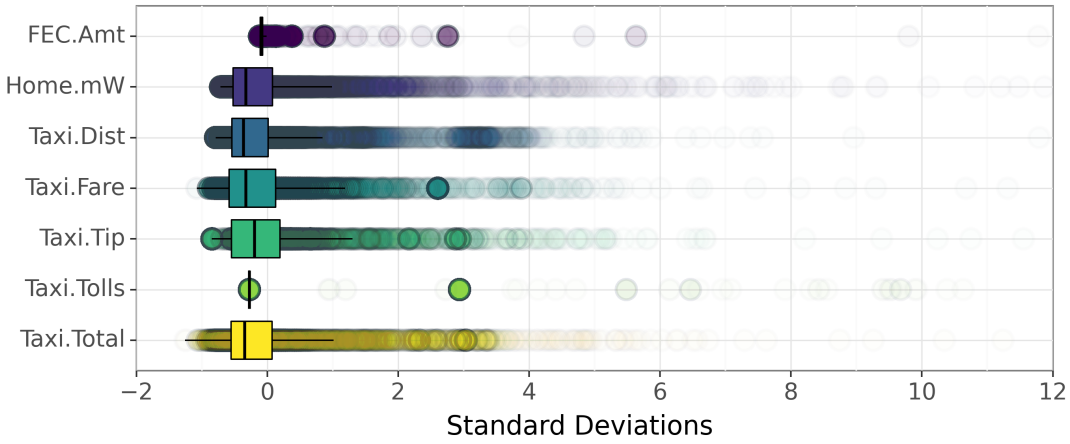

Figure 3.4: The distribution of 10,000 samples from each explored column across the three datasets. Each sample is drawn as a mostly transparent circle; visible points indicate the overlap of samples.

to political campaigns in the US [23], taxi data from New York City, USA [61], and energy consumption readings from households in London, UK [84]. We refer to these datasets as the "FEC," "Taxi," and "Home" datasets respectively. We describe the selected columns from these datasets in Table 3.3 and depict the distribution of each column in Figure 3.4. These datasets represent real-world data with skewed values, which is the target for forward encodings. The seven selected columns, from across the three datasets, facilitate a quantitative evaluation of how effectively forward encodings improve bit-parallel techniques when processing skewed data. As an example of the skewed nature of the chosen columns, while many taxi rides are short trips within NYC, some riders arrange long-distance (over 100 miles) trips with drivers for negotiated rates. Similarly, of the approximately 63 million individual political campaign contributions from 2021 to 2022, there were 778 individual contributions of 1 million USD or more.

We apply a minimal amount of cleaning to the datasets. As we evaluate both DFE and EDFE, we remove all negative values from the selected columns (the Home dataset did not have negative values). We also removed all rows from the Taxi dataset that were missing values. To store the columns as unsigned integers,

we apply unit transformations to fixed precision decimals (such as dollars and cents to cents). We replicated each dataset multiple times, resulting in each column having a size of about 400 million elements.

Note that removing negative elements from the datasets impacts the performance of EDFE, as branch prediction is able to quickly learn that the sign bit is never set when processing an EDFE encoded value. However, this impact is minor due to the small number of instructions required to encode to and decode from (E)DFE.

The distributions of the skewed columns are shown in Figure 3.4. This figure was generated by normalizing and then sampling each column 10,000 times. We use sampling to showcase the heavily skewed nature of the entire dataset, as opposed to each column being fairly normal except for a few outliers. For example, the tolls column of the Taxi dataset (Taxi.Tolls) is skewed due to about 92% of the values in the column being equal to zero. Of the remaining 8% of values, we clearly see a number of repeated values; one such value is $6.55 (represented as 655), the current "E-ZPass" fare rate for multiple bridges and tunnels around NYC [60]. The other columns showcase similar behavior, with varying degrees of skew and multimodalness. As we demonstrate throughout the rest of our evaluation, forward encodings allow bit-parallel columns to process skewed data more efficiently.

The three real-world datasets we use differ from the previous evaluations of bit-parallel techniques, usually performed using TPC-H or SSB [63; 82]. Earlier, we explored the behavior of runtime discovered early stopping when processing the "LO_QUANTITY" column from SSB (Section 3.2.3), which contains values in the range of $[1, 50]$. As previously shown by the early stopping model of the LO_QUANTITY column, each additional bit evaluated causes many records to discover a stop, irrespective of the exact query predicate. This behavior does not extend to columns containing skewed data using the two's complement integer representation, as the large outlier values require a large representation range and thus necessitate the usage of leading zeroes in the non-outlier data values. Thus, while each column is stored using the minimum number of bits per encoding (Table 3.3), the representation ranges of these bit counts far exceed the median

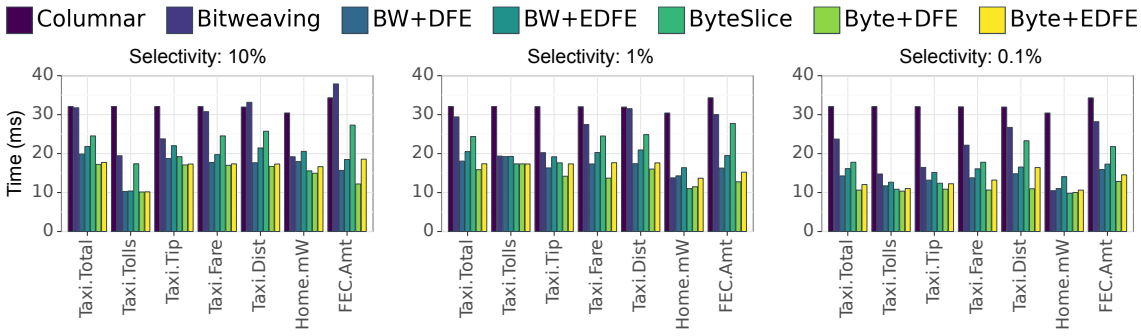

Figure 3.5: Average scan performance of the "data greater than constant" query applied to each column using 20 threads, separated into individual figures by filter selectivity.

value of each column. In these cases, runtime discovered early stopping (traditional early pruning) is significantly less effective, necessitating alternative mechanisms to provide stopping opportunities for bit-parallel columns.

3.4.2 Scan Performance

To evaluate the performance impact of forward encodings on columnar scans, we perform multiple scans using the following query:

```
SELECT count(*) FROM column WHERE (column > threshold)
```

The threshold is a constant query parameter that results in a query with 10%, 1%, or 0.1% selectivity. For example, the 10%, 1%, and 0.1% selectivity thresholds for the campaign contribution transaction amount (FEC.Amt) are $112, $2000, and $5800 USD, respectively. We specifically use the "greater than" operator because we are concerned with processing the outlier data from the aforementioned datasets.

We include our overall scan results in Figure 3.5, where each scan was performed using 20 threads. Overall, ByteSlice with DFE is the best-performing technique. This is not particularly surprising, as it combines the state-of-the-art bit-parallel technique with the forward encoding that emphasizes performance over wider applicability. In the 20-thread configuration, ByteSlice with DFE has an average

geometric mean speed-up over ByteSlice of 1.47x. In contrast, ByteSlice with EDFE has an average geo. mean speed-up of 1.29x. This is primarily due to the extra two bits, which do not benefit runtime discovered early stopping for the selected datasets.

BitWeaving/V, implemented with a strata width ("bit group size") of $s = 4$ requires more memory reads than ByteSlice ($s = 8$) to process the upper field of each value. Because of this difference in storage organization, ByteSlice+DFE usually outperforms BitWeaving/V+DFE. In addition, the smaller strata width is also more impacted by the skewed data, as the increased number of strata can incur more memory accesses when relying solely on runtime discovered early stopping. BitWeaving/V+DFE has an average speed-up of 1.54x over BitWeaving/V (both using 20 threads), while BitWeaving/V+EDFE has a respective speed-up of 1.35x. These results match our expectations; due to BitWeaving/V's smaller strata width, the planned stop skips more BitWeaving/V strata than ByteSlice strata.

We also vary the number of threads used to perform each scan-based filter. The results from these experiments are shown in Figure 3.6, which depicts the geometric mean performance of each columnar scan across multiple thread configurations at each selectivity threshold. The performance stagnation between 10 and 20 threads is due to the used CPU only having 10 physical cores with two logical cores each; using the second logical core of each physical core does not improve the scan performance of the evaluated techniques. This result is expected,

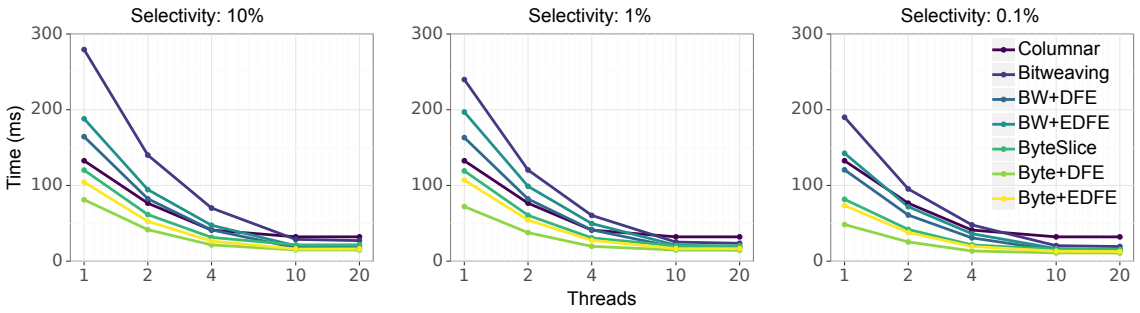

Figure 3.6: Geometric mean of scan performance of all evaluated columns, across all selectivity and threading configurations

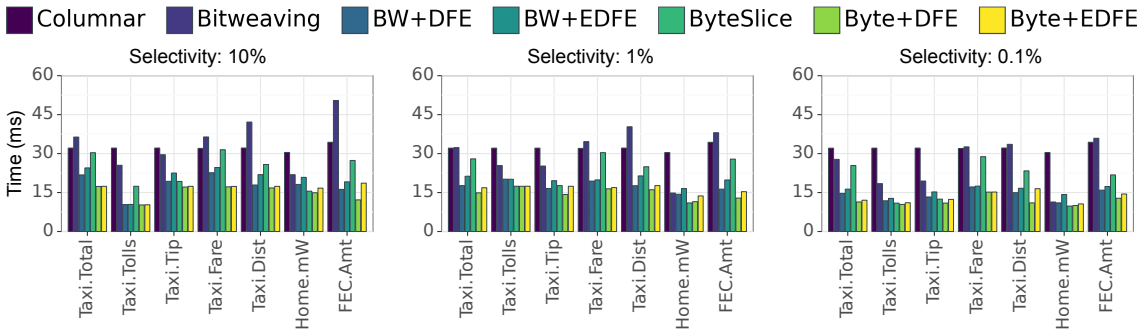

Figure 3.7: Average scan performance of each column using 20 threads when the alternative "lower than" query is applied, separated into individual figures by filter selectivity.

as scan operations are generally limited more by memory performance than CPU performance. Further, this behavior results in "free" CPU cycles that can be used for tasks such as decoding while waiting for memory transfers. We measure the utilization of these otherwise unused cycles in Section 3.4.6.

We note that the evaluated scans selected values above the 90%, 99%, and 99.9% percentiles, as opposed to queries that select beneath the 10%, 1%, and 0.1% percentiles. We depict the results of these "lower than" queries in Figure 3.7. While smaller values have fewer salient bits, the skewed distributions of the columns result in many of these small values being present in a tight cluster. This clustering

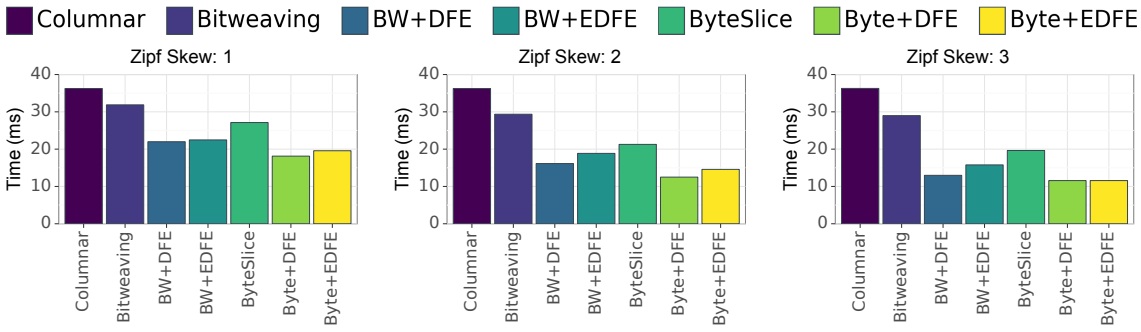

Figure 3.8: Average scan performance of each skewed, synthetic dataset using 20 threads, separated into individual figures by the skew parameter for each Zipf distribution.

of values significantly impacts the performance of existing bit-parallel techniques as it impairs traditional runtime discovered early stopping. In contrast, the salient bit count prevents a lack of discovered early stopping opportunities from significantly decreasing scan performance.

3.4.3 Synthetic Data Scan Performance

We also perform scans on synthetic columns with skewed distributions. We generate columns of 400M elements using one uniform-random and three Zipf (skew of 1, 2, and 3) distributions of values in the range [1,1M]. We perform a scan-based filter operation on the column with the query "greater than 100."

The results from the experiments on synthetic, skewed columns are shown in Figure 3.8. The speed-up of using (E)DFE increases as the skew increases. Across all storage configurations, DFE speed-ups ranged from 1.4x to 2.2x, while EDFE speed-ups ranged from 1.4x to 1.8x. These results align with our expectations due to the impact of parallelism group size g on scans: the less skewed distributions have a larger percentage of distribution occupied by values equal to the scan predicate (100), and thus require a larger fraction of the groups to be evaluated without the benefit of runtime discovered early stopping. However, while the speed-up of applying (E)DFE to the uniform random column was lower (about 1.7x for BitWeaving/V and 1.5x for ByteSlice), the time to complete the query for FE-encoded columns containing the uniform-random distribution was similar to the Zipf-3 distribution, at around 12ms; the speed-up was reduced because BitWeaving/V and ByteSlice performed significantly better when the columns were not skewed (20ms and 17ms vs. 32ms and 27ms for BitWeaving/V and ByteSlice, respectively). These results mirror the initial exploration of runtime discovered early stopping, as illustrated by our original model (Section 3.2.3).

We also scan a column of uniform-random distributed values in the range $[1, 50]$ using the same experimental setup as before (Section 3.4.2). This scan is similar to experiments performed in existing work [24; 51]. The distribution of this column is the same as the previously explored SSB column "LO_QUANTITY" (Section 3.2.3).

We perform a scan-based filter operation at 10% selectivity (column > 45). This result is depicted in Figure 3.9. We note that the bit width of the column is 6, 9, and 8 for INT, DFE, and EDFE, respectively.

ByteSlice and ByteSlice+EDFE have similar performance, as both methods use the same stratification (one 8-bit stratum), in contrast to the two strata that ByteSlice+DFE uses. As DFE requires an additional stratum to be processed when using both BitWeaving/V and ByteSlice, it cannot outperform the INT encoding. Further, BitWeaving/V+EDFE incurs significant overhead when testing the format bit to determine whether the second stratum should be loaded. This behavior incurs branch misses and randomizes the strata access pattern, resulting in additional overhead that does not improve performance over streaming the full 8 bits of all values from memory. Prior work has identified this behavior as a source of decreased performance [51].

Broadly, we include these results because forward encodings are designed for the specific case of processing skewed datasets with bit-parallel techniques. On a uniform-random dataset, ByteSlice using EDFE offers no performance improvement over ByteSlice using the existing integer encoding, though we note that it does not decrease performance either. Existing bit-parallel techniques have contributed

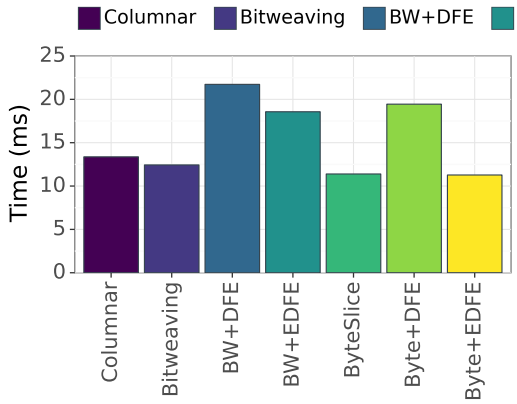

Figure 3.9: Average scan performance of the synthetic uniform-random column using each storage format.

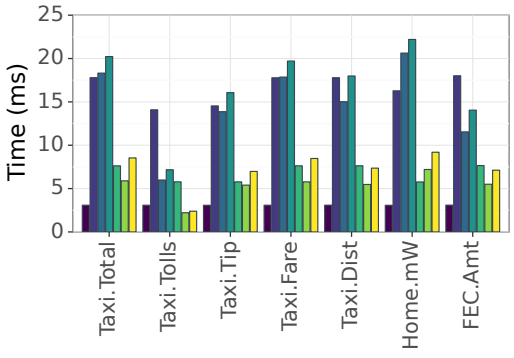

Figure 3.10: Average time to perform one million random fetch operations on each column using each storage format.

numerous ways to process uniform-random datasets; bit-parallel techniques currently lack the ability to efficiently process skewed columns, as showcased by our other results. These results on a uniform-random dataset demonstrate that our encodings improve the performance of scanning and fetching from skewed datasets without a significant decrease in performance when applying the same operations to columns without skew.

3.4.4 Fetch Performance

Our fetch microbenchmark is implemented using the following steps: randomly generate a row ID, fetch the corresponding element, and then decode the value to INT (if necessary). We randomly fetch one million elements from each column to evaluate the fetch performance of forwarded encoded columns. This benchmark poses a significant challenge for bit-parallel storage formats: by rearranging bits in memory to be more amenable to columnar scans, additional memory reads must be performed to reconstruct the original values. Thus, storage formats that use more strata have a higher average fetch latency. However, as we will demonstrate, the usage of the salient bit count allows for some strata to be bypassed during the fetch operation, as those strata do not contain salient bits for the particular fetched value. Explicitly flushing the CPU cache did not significantly impact the fetch benchmark results.

The results of our fetch microbenchmark are depicted in Figure 3.10. We emphasize the fetch results from the "Taxi.Tip" column. Note that 99% of the values in this column are under approximately $15 (Table 3.3, 1493), which requires 11, 15, or 17 bits to represent using INT, DFE, or EDFE respectively. Thus, while the INT column requires fewer bits to represent all values (18 vs. 22/24 for DFE/EDFE), 99% of fetches using DFE/EDFE require only 15/17 bits instead of 18 bits. However, because BitWeaving/V and ByteSlice use strata widths of 4 and 8, respectively, increasing the bits fetched from 15 to 17 causes an additional stratum to be accessed. Thus, DFE, but not EDFE, improves the fetch performance of the Taxi.Tip column.

While ByteSlice using DFE is always the best-performing fetch configuration,

the exact results depend on the average number of strata required to reconstruct the original value. Broadly, the salient bit count of forward encodings can significantly improve the fetch performance of skewed bit-parallel columns. DFE provides a geometric mean fetch performance improvement of 1.33x and 1.19x for ByteSlice and BitWeaving/V, respectively. While EDFE does not offer a significant performance improvement (2% and 4% for the respective techniques), these results are expected due to the cost of retrieving the sign and format bit.

3.4.5 Compressiblity

We evaluate the compressibility of the proposed encodings when using both bit-parallel and byte-parallel storage formats. We compress each of the used dataset columns using Zstandard (zstd) at compression level 3 [54]. These results are shown in Table 3.4. There is no clear-cut choice of storage format most suitable for compression when using zstd alone. Note that we calculate our compression ratio against the baseline columnar representation, which advantages the columns that use fewer bits to represent the uncompressed values. Overall, the best choice of compression format relies on the complex interactions between a column's data, bit-parallel storage format, and encoding.

As a second experiment, we also compress the columns by applying either run-length (RLE) or delta encoding to the columns before zstd, the results of which are

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	4.25	4.14	3.58	3.67	4.27	**4.42**	4.19
H.mW	**2.34**	2.01	1.83	1.82	2.11	2.03	2.07
T.Dist	2.41	**3.09**	2.53	2.55	3.02	3.02	2.86
T.Fare	**3.60**	2.79	2.54	2.53	3.27	3.29	3.40
T.Tip	3.10	3.38	2.81	2.87	**3.66**	3.42	3.33
T.Tolls	**24.11**	11.06	9.36	10.67	12.29	17.76	17.23
T.Total	2.45	2.54	2.27	2.29	**2.62**	2.61	2.50

Table 3.4: The compression ratio of each evaluated column, using each storage format when compressed using zstd. "BL" is the baseline columnar method, "b" refers to BitWeaving/V-stored columns, and "B" refers to columns stored in the ByteSlice format. The best compression technique for each column is in bold.

shown in Tables 3.5 and 3.6, respectively. Applying run-length or delta encoding before compression does not significantly impact the results. We note that the evaluated datasets are fairly antagonistic to RLE due to the large number of unique values in each dataset. For example, the average run length of the Taxi.Distance column is about 1.01, which effectively doubles the column size by repeatedly storing runs of length 1. Both RLE and Delta compression demonstrate two related behaviors of alternative integer encodings. First, RLE and other techniques that compress a series of integers as a stream of symbols (dictionary, move-to-front, etc.) are not impacted by the exact value of the underlying integer but rather by the repetition of the symbol. Thus, changing the bit representation of a value does not significantly impact the overall performance of RLE compression. Second, while delta compression is impacted by the distance between values, the recorded distances can be interpreted as a pattern of symbols, regardless of the exact bit representation of each delta.

Overall, applying run-length and delta encodings to each of the evaluated columns before compressing with zstd does not lead to one technique being more compressible than the others.

3.4.6 Encode and Decode Performance

We compare our proposed FEs with existing integer encoding techniques to better frame the performance cost of (E)DFE encode and decode operations. We evaluate

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	**6.88**	6.24	5.59	5.49	6.69	6.02	6.10
H.mW	**4.52**	3.52	3.33	3.33	3.33	3.25	3.26
T.Dist	4.31	**5.44**	4.59	4.51	4.92	4.53	4.68
T.Fare	5.36	4.87	4.56	4.51	4.85	4.92	**5.37**
T.Tip	4.76	**5.52**	4.86	4.84	5.30	5.32	5.34
T.Tolls	10.86	7.93	14.78	16.03	8.49	**34.87**	29.52
T.Total	4.28	**4.51**	4.16	4.18	4.21	4.16	4.10

Table 3.5: The compression ratio of each column after compressing using run-length encoding then zstd.

DFE and EDFE against a unary code, the Elias δ, γ, and ω codes [20], as well as Rice codes with turntable parameters of 4 and 8 [74]. We evaluate a uniform random distribution with values in the range $[1, 50]$, similar to the previously described LO_QUANTITY column. First, we record the time it takes to encode the entire column. Then, we randomly fetch 10M (2.5% of the column) values.

For lightweight encodings like (E)DFE, the encoding and decoding process can often be "free" when it is performed in otherwise wasted cycles, such as during the time spent waiting for memory. Compared to an unencoded baseline, DFE and unary encodings incurred no significant amount of overhead during decoding. γ and Rice 4/8 both had decoding overheads in the range of 2.5% to 3.5%. δ, EDFE, and ω had the largest encoding overheads, of 12%, 35%, and 108% respectively. Encoding performance is similar, where γ, unary, and Rice 8 encodings did not incur a significant overhead. δ, DFE, and Rice 4 added an additional 5%, 6%, and 9% encoding overhead, respectively. EDFE and ω incurred the largest encoding overheads of 19% and 55%. Overall, while encoding and decoding DFE is faster than EDFE, both are lightweight codes with costs comparable to other integer codes.

However, these alternative encodings are not FEs. The Elias codes do not preserve compatibility with integer comparison hardware. The unary component of Rice codes limits their applicability to columns with skewed values; while using large divisors limits the size of the unary component, the increased number of remainder bits results in more integer-like early stopping behavior. Thus, while we compare the cost to encode and decode (E)DFE against a number of existing

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	3.44	2.46	3.33	3.30	2.78	3.70	**3.76**
H.mW	**2.32**	1.83	1.84	1.75	2.09	1.98	1.97
T.Dist	2.19	2.19	2.51	2.43	2.35	**2.68**	2.61
T.Fare	**3.15**	2.07	2.48	2.32	2.59	2.79	3.00
T.Tip	2.45	2.61	2.46	2.38	**2.76**	2.73	2.75
T.Tolls	**22.66**	6.34	6.34	6.14	11.51	16.65	16.00
T.Total	1.91	1.95	2.27	2.12	2.14	2.35	**2.29**

Table 3.6: The compression ratio of each column after compressing using delta encoding then zstd.

encodings, we also note that each is tailored for its own specific use cases.

3.5 Discussion

Before we performed our evaluation, we established four questions that our experiments were intended to explore:

A1. Forward encodings limit the maximum number of strata that must be evaluated when performing a scan-based filter operation using a constant predicate. When the filtering predicate is small compared to the maximum value that can be represented by the bit width of the column, many strata can be skipped.

A2. The salient bit count is applicable to both scan and fetch operations. As l_t controls the number of strata retrieved, the memory accesses required to perform a fetch on a forward encoded column are based on the value fetched and the strata width of the column rather than the overall strata count. In the case of significantly skewed datasets, the additional strata that non-FE columns must process harm performance, a penalty that is not incurred by (E)DFE-encoded columns.

A3. Forward encodings are not significantly more or less compressible than existing integer encodings.

A4. Encoding and decoding (E)DFE has a performance cost similar to other integer codes.

As the scan-related results demonstrate, forward encodings improve the performance of bit-parallel scanning techniques by reducing their data sensitivity. The previously explored early stopping model (Section 3.2.3) predicts that early pruning may be ineffective depending on the bit-distributions of values in a column. These predictions held when exploring our selected skewed datasets (both real-world and synthetic), as the bits composing each value did not equally contribute opportunities for runtime discovered early stopping. Forward encodings address

this data sensitivity by expanding early stopping to include planned early stops. The salient bit count l_t sets an upper bound on the number of strata that must be evaluated, reducing the average number of bits examined during the scan. Beyond reducing the execution time of scan-based filtering, this upper bound also helps stabilize the performance of scans on bit-parallel columns by reducing predicate-based variability in query execution time.

By selecting a strata size to use when splitting data vertically, bit-parallel techniques introduce a complex relationship between the bit-representation of the data and the performance of database operators. Forward encodings reduce the variability of many bit-parallel storage configurations by using the salient bit count, providing a speed-up by elevating the performance of non-ideal cases. While we focus on skewed distributions, our previously explored early stopping model is an effective tool when exploring other data distributions.

3.6 Conclusions and Future Work

Runtime discovered early stopping is a core component of modern bit-parallel techniques. In this work, we explored a new family of integer encodings (forward encodings) that shift the salient bits of existing integer representations closer to the MSB, which enables more efficient early stopping in bit-parallel methods. Further, forward encodings (FEs) also enable additional opportunities for early stopping, such as planning a stop before performing a predicate-based scan or discovering an early stop during a fetch operation. These alternative stopping methods significantly improve the performance of bit-parallel techniques when processing skewed data, where the existing runtime discovered early stopping technique (known as early pruning) is ineffective.

We have also proposed a systematic way to think about encodings and storage organization. Using this framework, we proposed two FEs: DFE and EDFE. These forward encodings allow for a new set of interactions with bit-parallel techniques while preserving compatibility with existing integer comparison operations. Further, all optimizations performed by existing techniques are applicable when

using FEs, allowing for DFE and EDFE's use as replacements for existing integer representations in bit-stratified methods.

In our evaluation, we demonstrated how forward encodings improve the performance of two existing bit-parallel storage formats when processing skewed data. While the research directions of encoding and bit-parallel techniques are orthogonal, the choice of encoding profoundly impacts bit-parallel methods due to the influence encodings have on early stopping.

Besides the two FEs we introduce, it seems possible to design additional forward encodings. By rethinking the encoding of integer types, we hope to encourage future research into reimagining how FE-based methods might be applicable in speeding up computations in data applications beyond just the predicate-based columnar scans and fetch operations that we consider. Given the complex interactions between encoding, storage format, and data distribution, there is ample opportunity for database engines to create significant performance improvements through automated optimization techniques that select the best combination of these orthogonal, yet connected, parameters.

3.7 Continuation: Generalizing Early Stopping

Figure 3.2 showcases the early stopping probability for one particular query for one particular data distribution. After this work was published, a straightforward yet deceptively complex question was asked: What is the early stopping probability of a forward-encoded value? To generalize the two's complement solution to other representations, we require two additional tools: interval arithmetic and information theory.

3.7.1 Framing

From a practical perspective, early stopping is an intuitive way to compare representations of natural numbers. Consider comparing two natural numbers written as right-aligned text, with one value per line (such as some receipts); the larger

number extends further to the left and is thus trivially identifiable. This is the case because when using a written positional number system, the infinite sequence of leading zero digits is omitted [44, Section 4.1]. A machine equivalent of this "comparison by eye" operation is the count leading zeroes (CLZ) instruction. However, because the value must be loaded from memory to CPU to perform a CLZ operation, the usefulness of CLZ instructions as a direct analog is limited. Instead, the following section is built around the concept of iteratively processing the digits of a number from left (most significant) to right (least significant).

3.7.2 Generalized Integer Early Stopping

Given a positional numeral system of base B, each value is represented by a sequence of N symbols from the alphabet containing B symbols. This alphabet ranges from 0 to $B-1$, where B itself is represented using the two symbol sequence $\{1,0\}$, irrespective of B.

We introduce a new symbol, X, as the unknown symbol to represent values that are not yet known. For example, an unsigned 8-bit integer "half-loaded" value may be $\{1,0,1,0,X,X,X,X\}$. It is useful to be able to operate on this half-loaded value. Thus, we make two assumptions:

- A minimum bound usually is when all X symbols are assumed to be 0.
- A maximum bound usually is when all X symbols are assumed to be $B-1$.

Under this construction, the previous example has a min/max of:

$$\{1,0,1,0,0,0,0,0\}/\{1,0,1,0,1,1,1,1\}$$

Thus, we have now represented a partially loaded value as an interval, to which we can apply interval arithmetic. In the following example equations, we replace unknown symbol representations with matching intervals.

- Base 2: $1010XXXX = [160, 175]$

- Base 10: $16X = [160, 169]$

Similarly, we can also represent the min-max range as a minimum and an offset: $1010XXXX = [160, 160 + 15]$. And once again, both forms can be used for compute via interval arithmetic rules.

- $1010XXXX * 2 = [160 * 2, 175 * 2] = [320, 350]$
- $1010XXXX^2 = [160^2, (160 + 15)^2] = [25600, 30625]$

Under this construction, offsets have the benefit of ignoring signed-ness, though we must recognize that some operations may invert a given interval's infimum or supremum.

Given an N-symbol integer x, the maximum uncertainty is when all N symbols in each integer are unknown. If all symbols are known, then there is 0 uncertainty. The number of uncertain symbols in an integer is n_x. The number of certain symbols in an integer is n_c. To simplify calculations, we assume that uncertain symbols start at the LSB and replace symbols towards the MSB. Under this construction, the interval offset of a partial integer is either: $B^{n_x} - 1$; or $B^{N-n_c} - 1$. These two formulations represent a symmetry in representation: We can either "count unknown symbols remaining," or we can "count total symbols known."

Once again, the interval offset can be used to simplify interval arithmetic.

Example 1: Add two fully unknown 32-bit values (base-2).

$$[0, 0 + (2^{32} - 1)] + [0, 0 + (2^{32} - 1)]$$

$$[0 + 0, (0 + (2^{32} - 1)) + (0 + (2^{32} - 1))] = [0, 2^{33} - 2]$$

Example 2: Add two signed (two's complement [86]) 4-bit values with the first three bits set to 1 and unknown LSBs (111X).

$$[-2, -1] + [-2, -1] = [-4, -2]$$

This construction is patently designed for use in computational frameworks that heavily leverage early stopping. Further, we now have the tools to succinctly rephrase the definition of stopping for scan-based filter operations: Given some data value x, a predicate value p, a logical operation $\bigstar$, and a boolean predicate applied as $x \bigstar p$, stopping can only occur when p is not contained by the partial-value interval $[x]$ of x.

While this generalization only uses a small portion of the previously discussed information theory background material, it is enough for our uses below.

3.7.3 Early Stopping DFE

While we have defined partial-value intervals for natural integer encodings, we have not shown how they may be computed for alternative encodings.

To provide a starting point, we return to the distribution of the TPC-H/SSB "Quantity" columns, as given by the default data generator: A uniform random distribution of integers in the inclusive range $[1, 50]$. In this distribution, the DFE upper field has the following values:

- "1" for $[1]$
- "2" for $[2, 3]$
- "3" for $[4, 7]$
- "4" for $[8, 15]$
- "5" for $[16, 31]$
- "6" for $[32, 50]$

Note that we have not assigned a bit width for the DFE value, and thus the upper field is of unknown bit-length; however, the upper field still has a known set of symbols that will occupy it. Similarly, we have no need to identify that an upper field of 0 maps to 0; the value is not in the distribution, and thus the symbol never appears.

For now, we define our early stopping action as the following: The upper field is read in its entirety before the lower field is read one bit at a time. Using one of the TPC-H Q6 default query parameters, namely 24, we encounter the following early stopping behavior:

First, values that have not been stopped after the upper field, by definition, have an upper field of 5. Thus, we can evaluate the odds of stopping in two discrete parts: upper and lower.

First, one out of six upper fields has a matching value (5). Of the 50 values in the distribution, 16 values share an upper field of 5.

Next, each bit processed from the lower field reduces the unknown bit count by one, reducing the partial-value interval by half. The upper field of "5" is associated with $16 = 2^4$ values, resulting in a maximum evaluation depth of 4. As a reminder, this 4 matches the previously discussed salient bit count when accounting for the hidden bit $(5 - 1 = 4)$.

Thus, of the 50 values, $1 - \frac{16}{50} = 68\%$ stop after comparing the upper field, then we see the number of remaining values cut in half after each step. Thus, for our defined early stopping action, the cumulative stopping probabilities for the predicate value 24, applied to the uniform random distribution $[1, 50]$, using DFE is (per symbol): $\{68\%, 84\%, 92\%, 96\%, 100\%\}$

From this initial construction, we can apply the previously discussed forms of calculating stopping probabilities for various tasks. For example, assuming the upper field is represented using 3 bits, we can create a bit-to-bit stopping model fairly simply.

We compare the value $5 = \{1, 0, 1\}$ bit-by-bit to the other upper field symbols: the values 4, 5, and 6 have their MSB set; the values 1, 4, and 5 have a zero in the middle bit position, and the values 1, 3, and 5 have their LSB set. In this case, while the first bit applies a stop for $\frac{3}{6}$ values, the second bit only stops upper $= 6$ in addition to the already stopped values, while the final bit stops upper $= 4$. Thus, processing the MSB stops upper fields with values of $\{1, 2, 3\}$; processing the first two bits stops upper fields with values $\{1, 2, 3, 6\}$, and the first three bits stop all but 5. Thus, using the previously identified distribution of upper fields, we result in cumulative

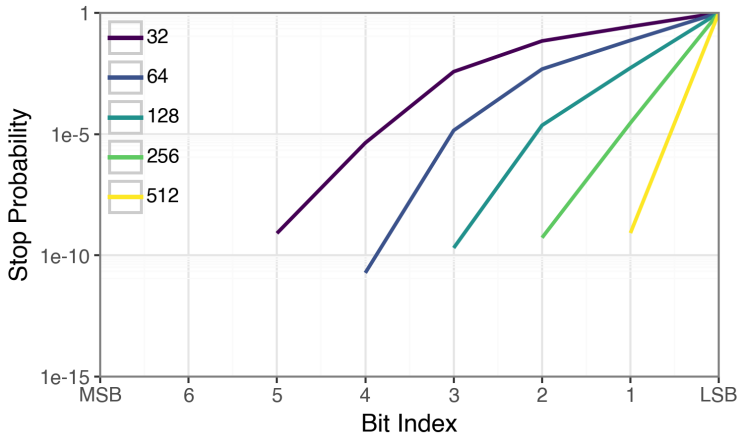

Figure 3.11: Stopping probabilities when performing a predicate-based scan on the TPC-H/SSB "Quantity" columns, using a value of 24, at multiple group sizes (g), when all values are represented in DFE. A (non-early) stop occurs when all bits of a value have been processed.

stopping odds of (per symbol/bit): $\{14\%, 52\%, 68\%, 84\%, 92\%, 96\%, 100\%\}$. As these are cumulative odds, we can directly apply parallelism to these values (all bits in a group must have stopped); for example, $g = 32$ raises each value to the 32nd power, resulting in about a 27% chance for compute to stop before the last bit is processed. These results are depicted in Figure 3.11.

4 ROW-SKIPPING METADATA

4.1 Introduction

Over the last decade, there has been considerable interest in tabular data storage formats like Parquet [6] and ORC [5] to aid interoperability between data platforms [94]. The disaggregation of storage and compute in the cloud has made these data formats even more important. Data is now stored in low-cost cloud storage in these open formats and accessed by data platforms that run in the compute layer. Cloud storage has become the primary data storage layer for such data platforms, separating the storage and compute layers at an architectural level.

If we inspect the storage layer more closely, we find a myriad of compute-adjacent tasks. Most prominently is compression-focused compute, which is fundamental for modern, high-performance data tasks [43]. We also find index structures such as small materialized aggregates (SMA, also known as zone maps) and bloom filters stored directly adjacent to data [10; 57]. These index structures form the *search acceleration layer* (SAL). This layer is leveraged to perform filtering via predicate pushdown, improving performance by reducing file I/O bandwidth usage.

Most file formats horizontally partition columns into nearly constant-length "chunks" or "blocks," that are independently compressed and annotated with search-acceleration metadata. Thus, search-acceleration and compression must operate on the same horizontal partition size. But depending on the underlying dataset, search-acceleration may only be useful when blocks are small; unfortunately, small blocks reduce compression efficiency. This trade-off forces chunk-based storage formats to either optimize for compressed file size or row-skipping during predicate pushdown or attempt to find a "one size fits most" configuration. Parquet and ORC have decided on their own partition-size configuration, either via specification or default parameters. Newer formats such as BtrBlocks [46] and Lance [48] challenge these existing parameters, proposing a set of new optimal configurations.

Although the contributions of each iteration of file format are both true and realized, the continuous revision of file formats is patently undesirable. To better

inform future tabular data storage formats, we attempt to find a horizontal partition size that performs well in the general case. However, we find that the partition sizes optimal for search acceleration and compressibility oppose each other (Section 4.3). Worse still, even between metadata structures that accelerate searching, we find that there is no "one size fits most" block size; while it is obvious that the behavior of summary structures is dependent on the underlying data, a block size that is optimal for one data distribution may degrade scan-based filter performance in another.

Without a singular, generally good horizontal partition size, we must approach this problem from an alternate direction. Returning to the fundamental notion of physical data independence [14], we propose the separation of the search-acceleration layer from the storage layer. The same data may be used in different ways by different applications, and the index layer may need to evolve dynamically. We argue there needs to be two standards that evolve independently and focus on their key strengths. First, a data storage layer that focuses on efficient data storage aspects like compression, to reduce the costs of data storage and transmission. Second, a loosely coupled but independent layer that enables efficient "querying" of data in the storage layer — the search-acceleration layer.

Separating the SAL from the storage layer enables each to evolve independently. As application needs change, search-acceleration structures can be added or removed. Similarly, both general improvements to compression techniques and inclusions of data-specific compression techniques are much more straightforward to implement. Further, these refinements can act as the data itself changes, allowing for a larger breadth of runtime-based optimizations for both storage efficiency and search-acceleration.

In this paper, we first demonstrate in Section 4.3 that the current status quo of requiring search-acceleration and storage partition sizes to be the same leads to unsatisfactory compromises between compressibility and search-acceleration. Further, we show that unlinking storage and search-acceleration structures enables optimizing partition sizes for each component, facilitating improved per-component performance. From these initial results, we propose a minimum set of goals for

separating the search-acceleration and storage layers based on principals rooted in physical data independence while still looking towards modern interoperability goals (Section 4.4). From this foundation, we then explore in Section 4.5 the impact of separated search-acceleration and storage layers on a real-world dataset.

4.2 Background

In this section, we discuss the history and role of row-skipping metadata in tabular data storage formats.

4.2.1 Blocked Arrays

Horizontal partitioning of large datasets is a well-studied technique that is utilized for a variety of reasons. From a myriad of use cases, there are many notable implementations, some of which we highlight [5; 6; 46; 48; 53]. Many of these techniques are inspired by the PAX data layout [4]. Although each format uses its own terminology, each usually horizontally partitions tabular data into "row groups." Within each horizontal partition, values are vertically partitioned, resulting in "blocked" or "chunked" columns.

This work focuses on the compression use case of these storage formats, which prioritizes minimizing the compressed size of stored data. Some forms of horizontal partitioning before compression evolved out of necessity. In one such method, a file is iteratively compressed as independent blocks, which are compressed and decompressed in isolation due to memory constraints. However, compressed partitions commonly must be entirely decompressed to access individual elements. Thus, storing some metadata describing the data held within the partition becomes beneficial to enable predicate pushdown without decompressing the underlying block.

4.2.2 Partition-skipping Metadata

The two most common forms of partition-skipping metadata are the *small materialized aggregate* (SMA, also known as a "zone map") and the *bloom filter* [7; 10; 57]. Before discussing their implementations, we first discuss shared properties of partition-skipping metadata.

First, partition-skipping metadata are defined over a partition size. For each individual partition, a single SMA or bloom filter *unit* is present. The number of horizontal partitions is the same as the number of partition-skipping metadata units. Thus, increasing the partition size decreases the number of metadata units. The importance of the partition size parameter is rooted in its origin: compression efficiency. Generally, a single partition size parameter is selected for all horizontal partitions, optimizing for compressibility, where the final partition is partially filled. While the exact compressibility of a dataset is highly data dependent, general heuristics of "at least a million" rows are common.

Second, partition-skipping metadata may emit false positives. Bloom filters store a hash-based manifest of values within their respective partition, which may result in false positives due to hash collisions. SMA may emit false positives depending on their usage. For example, a min/max SMA does not convey which exact values are contained by their range. In contrast, a null count or distinct count SMA precisely identifies the cardinality or the presence of null values in a horizontal partition.

Finally, partition-skipping metadata may scale in efficiency based on allocated resources. For example, the size of a bloom filter can be increased to reduce false positive rates [7]. This process is generally performed via either an optimization algorithm or a set of heuristics [56]. In contrast, SMA are $O(1)$; while SMA may or may not include a null count or distinct count, the practical impact of this decision is growing from two values per SMA unit to four values. Further, the efficiency of null counts and distinct counts is both dataset- and application-dependent; thus, the decision to include null counts and distinct counts is generally made per column rather than per partition.

However, while SMA is generally of constant size per unit, the number of row-

skipping metadata units depends on the column's horizontal partition size. Thus, all row-skipping metadata have a space complexity of at least $O(n)$, where n is the number of horizontal partitions. The *search acceleration overhead* of row-skipping metadata for a column is proportional to the number of horizontal partitions. In practice, this overhead is small due to horizontal partition sizes being dictated by compression-driven needs. When we unlink search acceleration from storage, we find that many row-skipping metadata configurations incur higher overhead costs. We demonstrate these costs in the following section.

4.3 Observations

In this section, we evaluate the impact the horizontal partition size of a blocked array has on both compressibility (Section 4.3.1) and performance of search-acceleration metadata (Section 4.3.2).

4.3.1 Optimizing Block Sizes for Compressability

Existing storage formats apply row-skipping metadata at the same granularity as their compression blocks. This is an intuitive approach because many compression techniques do not allow for the retrieval of individual values within a compressed block. Although it is well understood that compression algorithms benefit from larger block sizes, there are only soft guidelines for minimum effective block sizes [33]. However, because existing implementations of row-skipping metadata require small block sizes, we investigate the relationship between small block sizes and compression ratios.

For our analysis below, we use Zstandard (zstd) [54] to compress horizontal partitions of columns. We use zstd for two reasons: First, it is already proven to perform at least comparably to other state-of-the-art compression algorithms [43]. Second, it supports creating a sampling-based dictionary to assist in compressing files even when block sizes are small. Each dictionary is trained using randomly sampled rows from its respective synthetic dataset. We measure the compression

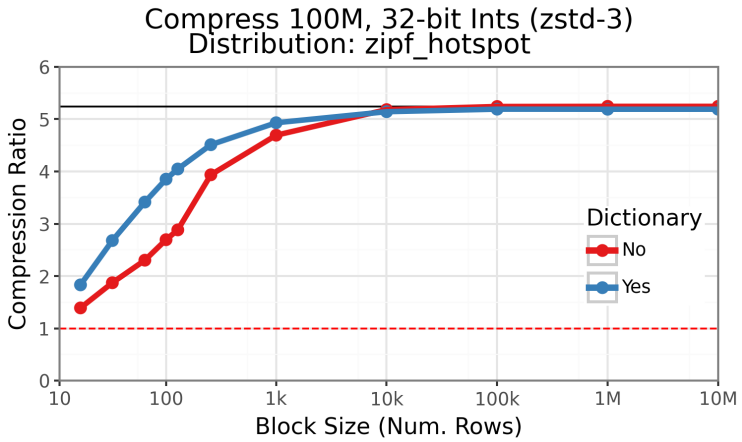

Figure 4.1: The compression ratio of the "hotspot" column, by block size and if a global dictionary was used.

ratio as the ratio of the size of the uncompressed data to the compressed data (including zstd dictionaries, if present). We operate zstd using its default compression level parameter ($\mathrm{CL}evel = 3$).

First, we compress a "Hotspot" Pseudo-Zipfian distributed column, over the ordered values [1, 100M] ($\mathrm{p} = 1.75$) [95]. This distribution resembles a dictionary-encoded column with a significantly skewed underlying dataset, where 1 is the most frequent value, 2 is the second most frequent, and so on. Our results are depicted in Figure 4.1. Compressing each block in isolation, we find that zstd requires a block size of at least 10k rows to achieve a compression ratio greater than 5. Beneath 10k, we see a rapid drop in compression ratios. We also depict the compression results when zstd first trains on 100 sampled blocks (same data, randomized alignment) before compression to generate an auto-optimized dictionary used by all blocks during compression. Although the dictionary does improve compression ratios at the low end, the block size must still be reasonably large to achieve near-optimal compression ratios.

Next, we also compress a "Gentle" Pseudo-Zipfean distributed column ($\mathrm{p} = 0.5$) using the same underlying values as the Hotspot column, shown in Figure 4.2. In this case, while the column is closely related to a dictionary-encoded column with

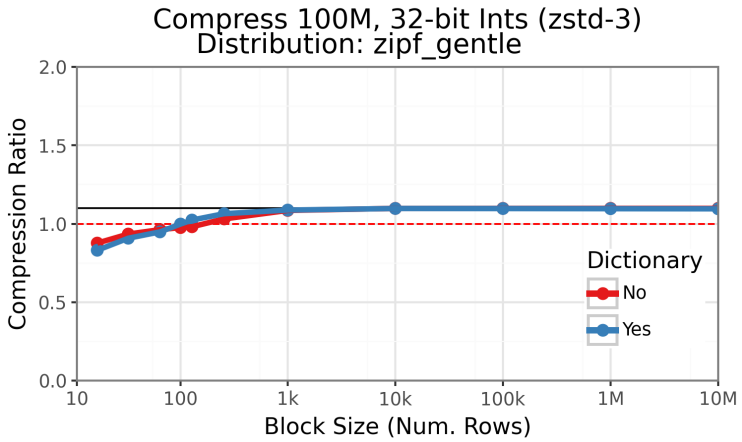

Figure 4.2: The compression ratio of the "gentle" column, by block size and if a global dictionary was used.

a skewed underlying dataset, the lower p parameter results in a longer distribution tail. While this distribution compresses comparatively poorly compared to the hotspot distribution, we see a similar convergence to near-maximum compression ratios once blocks are at least 1k rows.

Overall, these results match common wisdom for sizing blocks. However, this common wisdom directly impacts search-acceleration metadata: block-skipping Metadata attached to compression blocks must function at least 1k+ row blocks, preferably 10k rows. The following sections demonstrate that 1k-10k+ rows per metadata unit results in suboptimal search-acceleration performance.

4.3.2 Optimizing Block Sizes for Row-Skipping

Having investigated effective horizontal partition sizes for compression, we now focus on the same for row-skipping metadata. In this experiment, we use row-skipping metadata to accelerate a scan-based filter operation over an uncompressed column. We use the same hotspot and gentle distributions as before, as well as the same block sizes. We filter for "data EQ predicate" where the predicate value is randomly generated within the range $[0, 100\text{k})$. As the distributions are skewed towards zero, few records will satisfy this predicate: on average, 1 per hotspot

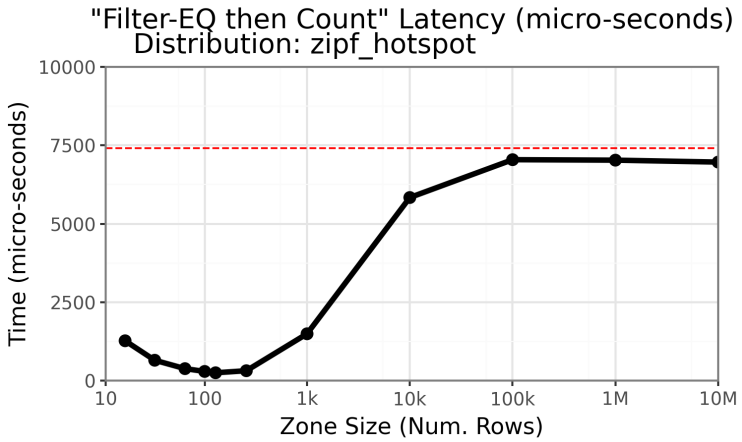

Figure 4.3: The time to perform a scan-based filter on the "hotspot" column, by block size.

distribution and 20 per gentle distribution. We generate one SMA and one bloom filter per block. The SMA is checked before the bloom filter to test for possible value membership. Our SMA records the minimum and maximum value per block, and our bloom filter is a single block Split Block Bloom Filter (SBBF) based on Parquet's implementation.[1]

Our experimental machine has an 8-core, 16-thread CPU capable of modern SIMD instructions, which we leverage where possible, and 64GB of DDR5 memory (4800 MT/s).

First, we find that block-skipping metadata is quite impactful for the hotspot distribution at small block sizes (Figure 4.3). The best configuration in this case is a block size of 128, resulting in a final scan time of about 252 µs. To contextualize this number, the 1k, 10k, and baseline (no row-skipping metadata) scans took ~1495 µs, ~5835 µs and ~7409 µs, respectively. Thus, while the 1k block size is 4.96x faster than the baseline scan, it is ~5.925x slower than the optimal configuration.

In contrast, the gentle distribution gains no significant performance improvement from block sizes of 1k (~7143 µs) and 10k (~7048 µs) over the baseline configuration (~7306 µs), as shown in Figure 4.4. While the smallest block sizes show

[1]We do not apply xxHash64 before inserting values into the SBBF, as we only use a single block SBBF and thus do not need to scramble the upper 32-bits.

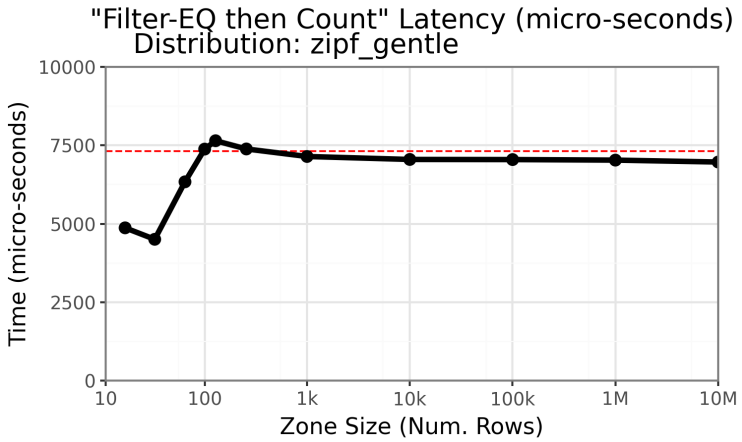

Figure 4.4: The time to perform a scan-based filter on the "gentle" column, by block size.

a modest improvement over the baseline (32: (~4504 µs)), block sizes between 100 to 256 slightly degrade scan performance. In both the sub-100 and 100 to 256 block size ranges, a significant amount of metadata overhead is present — 40B per partition. However, while the sub-100 block sizes are accompanied by fine-grained metadata units, the metadata units in the 100-256 size range are neither fine-grained enough to accelerate the query significantly nor lightweight enough to not incur a performance loss from overhead.

Thus, we reach a crossroads: prioritize the ability to skip blocks or prioritize compression ratios. Block formats today implicitly bundle these two considerations into one parameter: the horizontal partition size. This makes it nearly impossible to optimize both while also unnecessarily increasing the complexity of the block format specification.

4.4 Search-Acceleration Layer

As demonstrated, the goals of search-acceleration and storage layers do not always align. In this section, we articulate the goals for a SAL that has been separated from an underlying storage layer.

We first introduce a new term to describe a more general usage of search-acceleration metadata: *coverage*. Coverage defines the number of records that a single unit of search-acceleration metadata describes. For existing chunk-based storage formats, coverage and partition size are equivalent. Coverage size is distinct from horizontal partition size for two reasons: First, coverages are not implied to be uniform across all columns in a dataset. This allows for a SAL to tune coverage sizes per column. Second, coverages are not implied to be uniform across different search-acceleration metadata for the same column. Within the same column, SMA units may cover groups of 1k rows, while bloom filters may cover groups of 10k rows. The ability to overlap search-acceleration metadata allows for a more nuanced approach for application-specific SALs.

A search-acceleration layer is a collection of layers of search-acceleration metadata of the same kind with shared coverage parameters. For example, consider a column with 100M rows that we horizontally partition into 100 individual, 1M row partitions. We create a two-layer SAL to accelerate searches on this column: the first layer is an SMA layer with coverages of 10k rows, and the second layer is a bloom filter layer with coverages of 1k rows. We apply an example query: count the number of rows where the predicate equals x. To search each of the 100-column partitions, we first query each SMA unit; if the SMA unit contains x in its range, we then iterate over the bloom filter units (10 in total) covered by this outer SMA unit. If both the SMA unit and bloom filter unit contain the predicate value, we search the portion of the underlying data covered by the queried SMA and bloom filter units via a scan-based filter. Ideally, we have cached the covered portion of the underlying partition, though we may have to decompress the larger horizontal partition to expose this smaller covered segment of column data.

Now, we define our goals for search-acceleration layers.

First and foremost, search-acceleration metadata should be stored separately from data. This separation is paramount to achieving maximal compression ratios, which is necessary for a storage format to remain competitive. Unless a storage layer can leverage included metadata for storage-related goals, their size reduces compression ratios without clear benefit. Complex search-acceleration metadata

like bloom filters can rapidly incur significant amounts of overhead, and thus should generally not be included in storage formats without a clear storage-related benefit. On the other hand, SMA units are generally lightweight when covering large horizontal partitions and thus can be included at near-zero cost.

Second, the SAL must be optimized for both the underlying data and the accelerated application. Independent of the underlying data, some applications can tolerate larger space overheads to maximize performance, while other applications must minimize overhead while maintaining minimum latency requirements. An application's overhead and latency needs influence the choice of search-acceleration metadata and each metadata's coverage parameter. Further, both applications and their data access patterns change over time; a given block may be "hot" for some time, only to become "cold" later. If block-skipping metadata is baked into the storage format and optimized for the hot case, that cost is now a permanent debt on the storage cost. By unlinking search-acceleration metadata coverage from the underlying horizontal partition sizes for compression, the SAL is now free to cache data at application-specific granularities. Similarly, a SAL is well positioned to leverage state-of-the-art storage layers that support partial decompression, querying compressed data, and other future advancements.

We note that modern data formats often already implement SALs to varying degrees. Our definition of a SAL allows us to explore these existing implementations using a shared framework.

First, in-memory data formats usually begin with a standard data storage format and then build additional search structures on top. While this is partly motivated by the optionality of search-acceleration metadata in existing storage formats, many of these data formats use their own SAL to guarantee quality of service. Thus, a shared mechanism to represent SALs would reduce repeated implementation costs, especially if shared libraries provide baseline SAL implementations for common block formats.

Second, columns with dictionary-encoded data can already result in what effectively amounts to a multiple-layer SAL. If we encode an entire column with a single dictionary, if a value is absent from the dictionary, then the value is absent

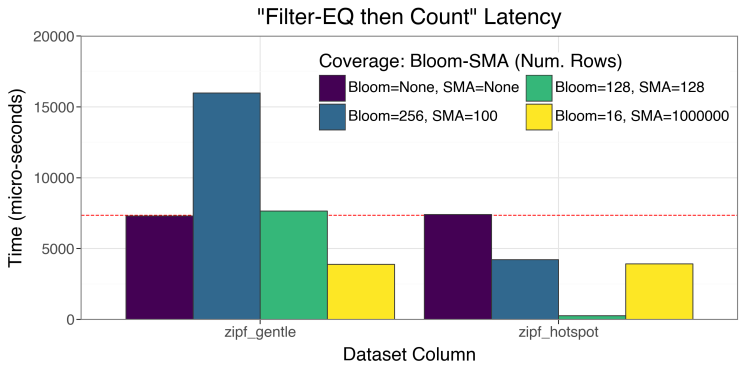

Figure 4.5: The performance of four different SAL configurations for the synthetic pseudo-zipfian distributions. The four configurations were selected from the best and worst configurations of each distribution.

from the column data. In this case, we can use the dictionary values to perform predicate pushdown before we query any other SAL metadata. This technique can be extended to other storage structures that indicate value presence in the overall column or an individual horizontal partition. A feature-complete SAL can leverage the underlying storage layer to accelerate searching; while the dictionary was included for compression, it still benefits search-acceleration.

4.5 Experimental results

In this section, we explore the feasibility of arbitrarily sized row-skipping metadata, unlinked from underlying horizontal partitions.

4.5.1 Synthetic Distribution Evaluation

We repeat the experiment from Section 4.3.2, though now we implement the search-acceleration metadata as a two-layer SAL: SMA then bloom filter. We evaluate every combination of the existing horizontal partition sizes as coverage parameters for each layer ("SMA=16 Bloom=16," "SMA=16 Bloom=32," etc.). We depict our findings in Figure 4.5, where we showcase the best and worst SAL coverage

configurations for both the "zipf_gentle" and "zipf_hotspot" distributions. The configuration that forgoes search-acceleration metadata is the worst-performing configuration we found for the zipf_hotspot distribution. In contrast, the worst performing configuration for the zipf_gentle distribution is when the bloom filter coverage is set to 256, and the SMA coverage is set to 100.

While the "Bloom=16, SMA=1000000" configuration does improve both scan-based filter queries over the baseline configuration of no metadata for both datasets, the "Bloom=128, SMA=128" configuration is about 15x faster for zipf_hotspot (~252 µs). This result mirrors our previous results, which show zipf_hotspot benefiting significantly from fine-grained metadata (Section 4.3.2). However, this same configuration is about 5% slower than the no metadata baseline for zipf_gentle ~7644 µs). Of the four configurations, only "Bloom=16, SMA=1000000" benefits the zipf_gentle column. Broadly, the performance differences between these two synthetic datasets reflect their sensitivity to metadata type and granularity: "one size fits most" leads to unsatisfactory compromises.

4.5.2 Real-world Dataset Evaluation

While we have demonstrated the usefulness of separating the search-acceleration and storage layers in synthetic workloads, we have not shown their feasibility for a real-world workload. We introduce the New York City (NYC) Yellow Cab dataset, provided by the NYC Taxi and Limousine Commission for 2023 [61]. Within this dataset, we highlight four rows: "Drop Off Location," "Rate Code," "Tip Amount," and "Total Fare."

We clean the dataset by filtering out all rows that record a taxi fare below the 2023 NYC initial taxi fare: $3.00. We represent all columns of the Taxi dataset using 32-bit integers. For decimal values (tips and total taxi cost), we store the column as a cents-unit integer. For enumerated columns, we store the enumerated values as-is. Then, we sample each real-data column with replacement to generate 100M-sized columns, the same size as the previously explored synthetic distribution columns.

First, we compress our four columns using the same set of compression param-

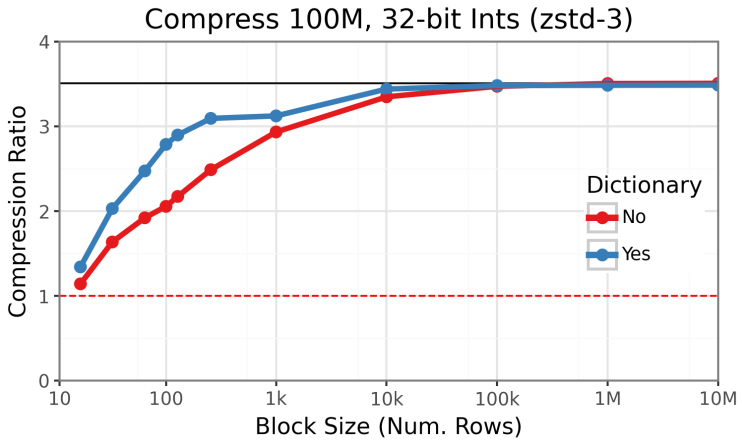

Figure 4.6: The compression ratio of the evaluated taxi dataset columns, by block size and if a global dictionary was used.

eters previously explored with the synthetic distributions (Section 4.3.1). These results are depicted in Figure 4.6. Once again, we find that a horizontal partition size of 10k rows or larger is necessary to achieve near-maximum compression ratios.

Then, we apply scan-based filter queries to each taxi column using the following predicate values.

1. How many rides were paid for using $20, using the remaining change as a tip? (0.2%)
2. How many rides were tipped exactly? $10 (0.7%)
3. How many ride fares were negotiated before the ride began? (0.5%)
4. How many rides ended at Newark Airport? (0.3%)
5. How many rides ended at LaGuardia Airport? (1.3%)

Each query has a baseline scan latency of about 7.25 ms. Similar to the synthetic dataset exploration, we showcase a limited set of results, though in this case, we only show the best-performing configurations. These results are depicted in Figure 4.7. If a configuration is best for a data set, we include this configuration's performance for

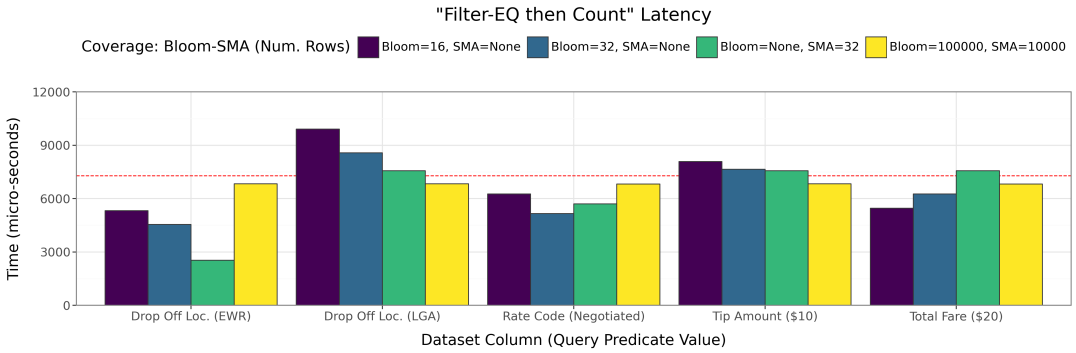

Figure 4.7: The performance of four different SAL configurations for the explored taxi dataset columns. The four configurations were selected from the best configurations for each distribution.

all the data sets. For example, "Bloom=32, SMA=None" is the best configuration for the Rate Code column.

The "Total Fare ($20)" query has a nearly opposite performance characteristic compared to the "Drop Off Location (LGA)" configuration. This result reinforces previous observations that the benefit of search-acceleration metadata is dataset-dependent. Of course, it is not unusual that columns containing different data have different optimal configurations for search-acceleration metadata. In contrast, we are surprised by the significant differences between the "Drop Off Location (EWR)" and "Drop Off Location (LGA)" configurations. The benefit of search-acceleration metadata can be wildly different depending on the predicate value for the scan-based filter performed. In this case, the best-performing SAL configuration for the LaGuardia (LGA) drop-off query performs poorly for the Newark (EWR) filter. The best-performing Newark configuration (No bloom filter and SMA of coverage 32) is uniquely performant. This uniqueness is due to the exact enumerated value for these two airports: 1 for Newark and 138 for LaGuardia. While these numbers make contextual sense, as LaGuardia Airport is within NYC and Newark is in New Jersey, Newark Airport, having the lowest Drop Off Location code, greatly benefits SMA.

Overall, we have reinforced that no "one size fits most" configuration exists

for search-acceleration metadata. Not only does the distribution of a column's underlying data impact the effectiveness of search-acceleration, but the applied predicate as well.

4.6 Conclusions

Currently, search-acceleration metadata like bloom filters and small materialized aggregates serve as a tool to avoid decompressing horizontal partitions of a compressed column. While advancements in columnar-like file formats have demonstrated that these structures can be used for search-acceleration, this benefit is always shown in the context of avoiding block decompression costs. We find this not only narrow in scope but also self-limiting. Under this construction, the partition sizes for search-acceleration metadata and compression are directly linked; as search-acceleration metadata benefits from small partitions while compression benefits from large partitions, we are forced to choose between one or the other. Modern storage layers always choose the latter option, as storage efficiency is their chief concern.

In this work, we have showcased an alternative path: splitting the storage layer into a dedicated storage component and a separate search-acceleration layer (SAL). By unlinking the two, each can prioritize its own needs. Further, we find that even within the SAL, underlying data distribution can drastically change the optimal configurations of row-skipping metadata.

5 CONCLUSION AND FUTURE WORK

In this chapter, I discuss avenues for future work and the challenges these research questions face, before discussing my closing thoughts.

5.1 Forward Encodings for General Numeric Representations

In the forward encoding paper, we converted the monetary columns in the taxi dataset to cent-unit integers. Cents integers are common practice when monetary applications only interact with small, consumer-facing applications. However, when a monetary task requires dividing a quantity of money into nearly equal portions, we require an entirely different numeric system. For example, dividing $100 (USD) by three may result in either two parts of $33.33 and one part $33.34, or the parts $33.33 and one withheld part of $0.01. Thus, without defining a set of best practices for preserving exact amounts, even elementary arithmetic may be ambiguous. For this reason, rounding rules and minimum decimal accuracy are defined in many accounting rules [22]. As there is no singular standard (GAAP for the US, IFRS for the EU, amongst others), data systems involving money must be flexible enough to meet monetary application needs [85]. In the totality of this context, it is not uncommon to see purpose-built numeric representations; for example: The IEEE 754-2008 Decimal Floating-Point Arithmetic Standard [1] and its implementation in the Intel Decimal Floating-Point Math Library [40]. The latter explicitly positions itself for applications where legal requirements require the use of decimal arithmetic instead of binary arithmetic.

Binary-coded decimals have long existed within computerized applications. The IBM System/360 [39] and the Motorola 68000 [58] both contained mechanisms for interacting with binary-coded decimals. Given this long history, we aim to introduce a forward encoding specifically for binary-coded decimals in future work. However, due to the wide variety of decimal representations and application needs,

the problem space quickly explodes. Thus, significant preliminary work must be undertaken to precisely define what real-world applications may benefit from an alternative encoding scheme. Further, many applications may place themselves in a position similar to that of SQLite, where increases in performance may not be worthwhile due to sacrifices in external areas such as format stability (Section 2.10).

5.2 Forward Encodings and Arithmetic

Arithmetic using forward-encoded numeric representations is not explored in published work. This has been left for future work due to two major obstacles.

First, while our previously explored forward encodings are integer representations, they share properties with floating-point values. In particular, arithmetic on two native DFE values requires a multi-step process where the upper field and lower fields are manipulated independently. In this case, we see the upper field resemble the exponent field of common floating-point representations even more closely. However, because decoding is only a few instructions, in many cases, it is faster to decode from DFE, perform the arithmetic, and then encode back to DFE.

Second, even if the theoretical underpinnings are sound, low-level algorithms are always beholden to the reality of modern hardware design. These real-world considerations had a substantial effect on the paper, which we discuss here.

In particular, we call "license-based downclocking" to attention [18]. While it is age-old wisdom that floating-point math is slower than integer math, the exact mechanisms behind this slowdown are due to the increased algorithmic complexity of multiplying floats, which in turn usually results in additional hardware [83]. This additional hardware has a significant impact on the overall chip, including power consumption and thermal considerations [21]. License-based downclocking partially avoids this issue by only downclocking when certain floating-point instructions are present. Colloquially, this process is known as "waking up the floating-point unit (FPU)," though fused multiply-add (FMA) units may also influence instruction licenses.

Unfortunately, in certain Intel™ hardware, the SIMD instructions to count leading zeros (VPLZCNTD, VPLZCNTQ) may cause a downclock due to their license. [1] This is troublesome for forward encodings, which leverage CLZ operations to create the upper field. Thus, while scalar encode/decode operations perform as expected, parallel encode/decode now may have surprisingly slow performance. Overall, arithmetic using forward-encoded values exposes many subtle performance considerations.

5.3 In-Situ Processing

Throughout this work, I have discussed techniques that, for a given data analysis operation, accomplish an equivalent operation with significantly reduced memory-to-CPU data transfer. Of course, this computing model assumes that data must be moved out of storage and into the CPU for processing. *In-situ* processing challenges this notion by attaching compute-capable hardware directly to storage.

There are many different kinds of in-situ processing technologies. One such technology is processing-in-memory (PIM) [16; 27; 49; 77; 90; 91; 93]. While PIM may or may not be the most suitable location for in-situ processing, it showcases many of the considerations that must be accounted for when co-designing hardware and software for a hardware-accelerated compute task.

First, as a memory row buffer is many (at least 16) cache lines wide, cache-line-based data parallelism now rapidly increases in size. This is an especially difficult problem for early-stopping-based compute to bypass, as large parallelism sizes can adversely impact performance (Section 3.2 and Section 3.7). In the remainder of this section, we explore the impact of these large row sizes when performing compute using multiple different PIM technologies.

Using the BLIMP simulator [16], we simulate the time required to filter rows of a 600M, 32-bit integer column using the predicate "data EQ constant" for any

[1] In Skylake Server architectures, "heavy" (FP/FMA unit) AVX2 instructions incur "soft" L1 transitions. In context, "light" AVX512 instructions incur "hard" L1 transitions while heavy AVX512 instructions incur soft L2 transitions. In Icelake client architectures, the soft and hard distinction seems to be removed, as the L1 license only applies to AVX512 instructions [18].

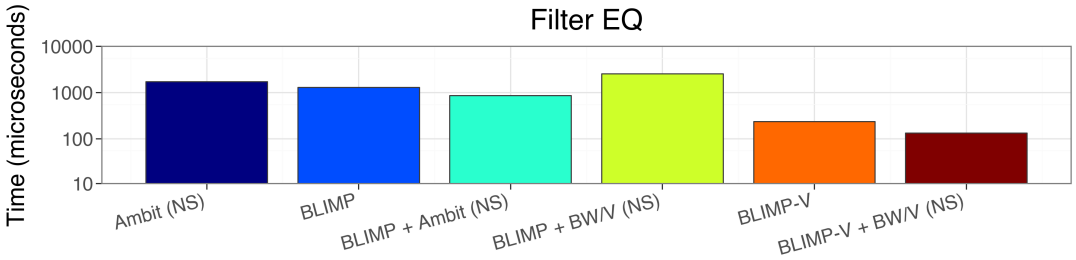

Figure 5.1: The filter-based scan latency of different PIM techniques. For techniques that can early stop, we disable stopping ("no stopping: "NS).

given 32-bit constant, outputting the result as a 600M long bitvector to the CPU that represents membership. First, Figure 5.1 showcases the performance of a number of PIM techniques that either cannot leverage early stopping or have their early stopping capabilities disabled.

First, increasing the area allotted to PIM technology increases its capability set, which generally increases performance. For comparison, BLIMP-V has about an 18% overhead while BLIMP has about a 4% area overhead. Adding the vector-compute unit to BLIMP significantly increases its performance. This is an expected result, reflecting existing in-situ processing trends.

However, departing from CPU baselines we explored earlier (Chapter 3), we do not see the same performance improvements by using bit-parallel compute instead of existing scalar compute. While BLIMP-V + BitWeaving/V (BW/V) incurs a moderate speedup of 1.77x (238 µs vs. 134 µs), BLIMP + BitWeaving/V (2554 µs) is significantly worse than its non-BW/V equivalent (1293 µs). This result demonstrates that, while the PIM techniques can be compared via area overhead, their compute capabilities are equally important. Because BLIMP lacks the vector processing units that BLIMP-V has, interacting with data in the vertical BitWeaving/V layout results in significant computational overhead, removing the effectiveness of the layout entirely. This is a starkly different result than prior CPU-driven work, which showcased benefits from vertical bit-parallel layouts.

To evaluate the impact of early stopping, we must define a data distribution and query predicate, which impact the overall query selectivity. We use two con-

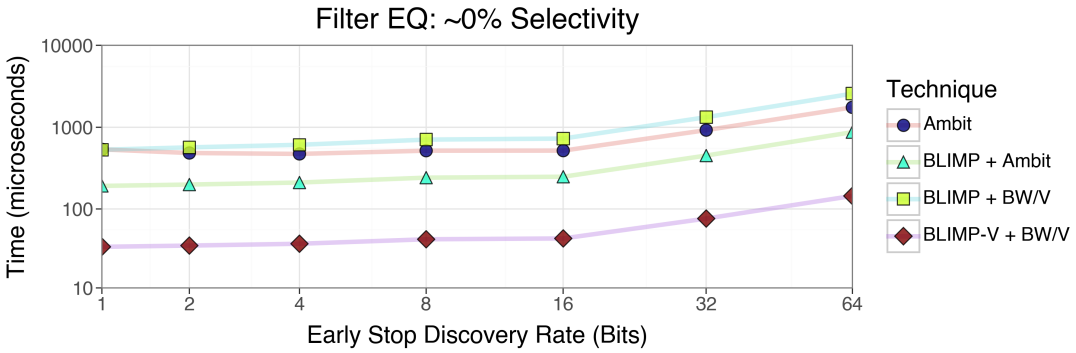

Figure 5.2: The filter-based scan latency of different PIM techniques, where the query has near zero selectivity.

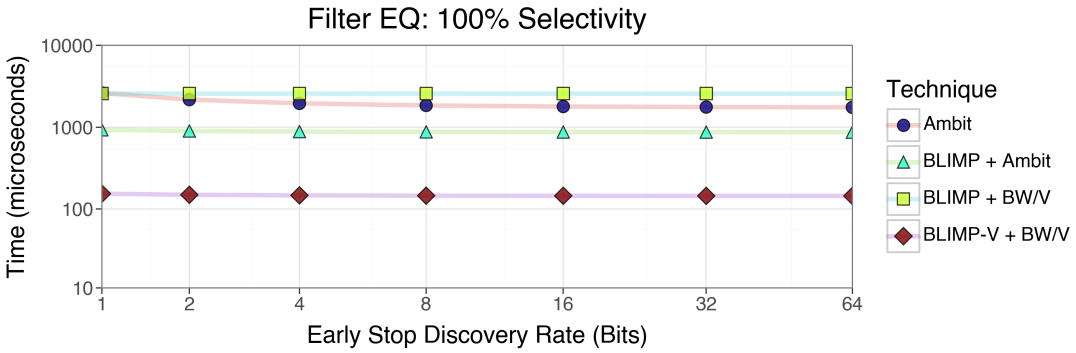

Figure 5.3: The filter-based scan latency of different PIM techniques, where the query has 100% selectivity.

figurations: First, we use a uniform-random data distribution of 64-bit integers in the half-open range $[0, 2^{64})$ and a query predicate of 1 (Figure 5.2). Next, we use a constant data distribution of 1, represented as a 64-bit integer, and query for values equal to 1 (Figure 5.3). Thus, our first configuration has a selectivity of near 0%, while our second configuration selects all values (100%). Then, for each technique, we vary the interval at which early stopping may be discovered from every 1 bit to every 64 bits, the latter causing an "early" stop right before the compute would naturally finish.

These results mirror previously discussed results (Section 3.4), where the optimal results minimize both the stop discovery rate and the average number of bits

processed. However, we find two important distinctions between these PIM results and our previous CPU results.

First, Ambit alone incurs CPU orchestration costs when performing a stop, as its compute unit is comparatively simple and requires direction from the CPU. In the case of BLIMP + Ambit, this coordination no longer must cross from memory to CPU, significantly increasing performance.

Second, due to the large size of the datasets, some rows do contain values that stop after one or two bits have been processed. Thus, while very frequent stop discovery steps lead to increased computational overhead, reducing the number of row activations significantly outweighs the performance costs of PIM-compute.

Together, these two points are a significant departure from CPU norms. Thus, for a proper PIM solution, we must design a bit-parallel implementation from the ground up with PIM in mind.

5.4 Conclusion

The concept of performing compute using values only "half loaded" from memory seems both unintuitive and impractical at face value. Yet, in this work, we have discussed use cases for this idea. Similarly, we demonstrated through multiple techniques that collections of data need not be processed in their entirety to achieve correct results. Broadly, we challenge existing columnar-processing techniques; instead of processing entire columns, we identify what exact portions of the underlying data are necessary to perform compute. In doing so, we opened up three doors to minimize the amount of data loaded during data analytic tasks.

First, we established a baseline: Columnar scans should be performed over data stored in columnar formats. While this seems like a trivial observation, we also showcased why this is not possible in all cases, and what steps may be taken to address such impossibilities by constructing an in-memory columnar storage layer for SQLite.

Next, we pushed columnar formats to their limits by exploring bit-parallel storage formats, which expose internal bit-columns of data. However, we show-

cased that many constructions of early-stopping rely upon the assumption that the underlying bits of data within a column are uniform-randomly distributed. As demonstrated by our real-world datasets, this is not always the case. By altering the representation of integer values, we were able to improve the performance of bit-parallel techniques, improving their usage in more general data cases.

Finally, we avoided processing horizontal partitions of columnar data through the use of metadata. We demonstrated that row-skipping metadata can be leveraged in complex ways, resulting in incredible performance gains when application needs, data distributions, and search-acceleration metadata arrangements all align.

Throughout this work, I have demonstrated that, sometimes, we can avoid the memory wall altogether by performing data analytics without processing data. In future work, I plan to extend these discussed techniques into new and related areas, exploring even more cases where "compute without (the entirety of the) data" is possible.

DISCARD THIS PAGE

COLOPHON

This document is based on a modified version of the "Wisconsin Dissertation Template" by William Benton, currently available via GitHub at the following link: `https://github.com/willb/wi-thesis-template`

BIBLIOGRAPHY

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[3] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, Association for Computing Machinery, p. 671–682.

[4] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving relations for cache performance. In *VLDB* (2001), vol. 1, pp. 169–180.

[5] Apache ORC Project. Apache orc. `https://orc.apache.org`, 2024. Accessed: 2024-06-12.

[6] Apache Parquet. Apache parquet documentation, Mar 2022.

[7] Apache Parquet. Apache parquet documentation: Bloom filter, 2024. Accessed: June 12, 2024.

[8] Apache Software Foundation. Apache arrow. `https://arrow.apache.org`, 2019.

[9] Barber, R., Bendel, P., Czech, M., Draese, O., Ho, F., Hrle, N., Idreos, S., Kim, M.-S., Koeth, O., Lee, J.-G., Li, T. T., Lohman, G., Morfonios, K., Müller, R., Murthy, K., Pandis, I., Qiao, L., Raman, V., Sidle, R., Stolze, K., and Szabo, S. Business analytics in (a) blink. *IEEE Data Engineering Bulletin 35*, 1 (2012), 9–14.

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

[11] Boncz, P. A., Kersten, M. L., and Manegold, S. Breaking the memory wall in monetdb. *Commun. ACM 51*, 12 (dec 2008), 77–85.

[12] Cam, L. L. The central limit theorem around 1935. *Statistical Science 1*, 1 (1986), 78–91.

[13] Chasseur, C., and Patel, J. M. Design and evaluation of storage organizations for read-optimized main memory databases. *Proc. VLDB Endow. 6*, 13 (aug 2013), 1474–1485.

[14] Codd, E. F. Is your dbms really relational?, oct 1985. (Note: The original document has been informally preserved by Dave Voorhis, 2019).

[15] Codd, E. F. *The Relational Model for Database Management: Version 2*. Addison-Wesley, 1990.

[16] Devic, A., Rai, S. B., Sivasubramaniam, A., Akel, A., Eilert, S., and Eno, J. To pim or not for emerging general purpose processing in ddr memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 231–244.

[17] Downs, T. Performance matters: Performance speed limits. `https://travisdowns.github.io/blog/2019/06/11/speed-limits.html`, 2019.

[18] Downs, T. Performance matters: Ice lake avx-512 downclocking. `https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html`, 2020.

[19] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of cloudlab. In *Proceedings of the 2019 USENIX Annual Technical Conference* (USA, 2019), USENIX ATC '19, USENIX Association, p. 1–14.

[20] Elias, P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21*, 2 (1975), 194–203.

[21] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011), pp. 365–376.

[22] European Commission: DG II. The introduction of the euro and the rounding of currency amounts. `https://ec.europa.eu/economy_finance/publications/pages/publication1224_en.pdf`, 1998. Accessed: 2024-06-12. See: `https://www.europeansources.info/record/the-introduction-of-the-euro-and-the-rounding-of-currency-amounts/`.

[23] Federal Election Commission. Individual contributions, 2022.

[24] Feng, Z., Lo, E., Kao, B., and Xu, W. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, p. 31–46.

[25] Fittl, L. libpg_query. `https://github.com/pganalyze/libpg_query`, 2017.

[26] Flores, I. Reflected number systems. *IRE Transactions on Electronic Computers EC-5*, 2 (June 1956), 79–82.

[27] Friesel, B., Lütke Dreimann, M., and Spinczyk, O. A full-system perspective on upmem performance. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (New York, NY, USA, 2023), DIMES '23, Association for Computing Machinery, p. 1–7.

[28] Färber, F., May, N., Lehner, W., Grosse, P., Müller, I., Rauhe, H., and Dees, J. The sap hana database - an architecture overview. *IEEE Data Eng. Bull. 35* (03 2012), 28–33.

[29] Gaffney, K. P., Prammer, M., Brasfield, L., Hipp, D. R., Kennedy, D., and Patel, J. M. Sqlite: past, present, and future. *Proc. VLDB Endow. 15*, 12 (aug 2022), 3535–3547.

[30] Galindo-Legaria, C. A., Grabs, T., Gukal, S., Herbert, S., Surna, A., Wang, S., Yu, W., Zabback, P., and Zhang, S. Optimizing star join queries for data warehousing in microsoft sql server. *2008 IEEE 24th International Conference on Data Engineering* (2008), 1190–1199.

[31] Golomb, S. Run-length encodings (corresp.). *IEEE Transactions on Information Theory 12*, 3 (1966), 399–401.

[32] Hankins, R. A., and Patel, J. M. Data morphing: an adaptive, cache-conscious storage technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (2003), VLDB '03, VLDB Endowment, p. 417–428.

[33] He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., and Xu, Z. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering* (2011), pp. 1199–1208.

[34] Hickey, T., Ju, Q., and Van Emden, M. H. Interval arithmetic: From principles to implementation. *J. ACM 48*, 5 (sep 2001), 1038–1068.

[35] Hipp, R. D., et al. SQLite, 2021.

[36] Hipp, R. D., et al. The Lemon LALR(1) Parser Generator, 2021.

[37] Hulsebos, M., Hu, K., Bakker, M., Zgraggen, E., Satyanarayan, A., Kraska, T., Demiralp, c., and Hidalgo, C. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (New York, NY, USA, 2019), KDD '19, Association for Computing Machinery, p. 1500–1508.

[38] Hustle Development Team. Hustle. `https://github.com/UWHustle/hustle/`, 2021.

[39] IBM. Ibm system/360 principles of operation. `http://bitsavers.trailing-edge.com/pdf/ibm/360/princOps/A22-6821-0_360PrincOps.pdf`, 1964.

[40] Intel. Intel decimal floating-point math library. `https://www.intel.com/content/www/us/en/developer/articles/tool/intel-decimal-floating-point-math-library.html`, 2018.

[41] Jiang, H., Liu, C., Paparrizos, J., Chien, A. A., Ma, J., and Elmore, A. J. Good to the last bit: Data-driven encoding with codecdb. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 843–856.

[42] Johnson, R., Raman, V., Sidle, R., and Swart, G. Row-wise parallel predicate evaluation. *Proc. VLDB Endow. 1*, 1 (aug 2008), 622–634.

[43] Karandikar, S., Udipi, A. N., Choi, J., Whangbo, J., Zhao, J., Kanev, S., Lim, E., Alakuijala, J., Madduri, V., Shao, Y. S., Nikolic, B., Asanovic, K., and Ranganathan, P. Cdpu: Co-designing compression and decompression processing units for hyperscale systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2023), ISCA '23, Association for Computing Machinery.

[44] Knuth, D. E. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997, ch. 4.1: Positional Number Systems.

[45] Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., and Zeier, A. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow. 5*, 1 (sep 2011), 61–72.

[46] Kuschewski, M., Sauerwein, D., Alhomssi, A., and Leis, V. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM Manag. Data 1*, 2 (jun 2023).

[47] Lamport, L. Multiple byte processing with full-word instructions. *Commun. ACM 18*, 8 (aug 1975), 471–475.

[48] LanceDB Development Team. Lancedb: A high-performance database system. `https://www.lancedb.org`, 2024. Accessed: 2024-06-12.

[49] Lenjani, M., Gonzalez, P., Sadredini, E., Li, S., Xie, Y., Akel, A., Eilert, S., Stan, M. R., and Skadron, K. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 556–569.

[50] Li, Y., Chasseur, C., and Patel, J. M. A padded encoding scheme to accelerate scans by leveraging skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, p. 1509–1524.

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

[52] Li, Y., and Patel, J. M. Widetable: An accelerator for analytical data processing. *Proc. VLDB Endow. 7*, 10 (jun 2014), 907–918.

[53] Liao, G., Liu, Y., Chen, J., and Abadi, D. J. Bullion: A column store for machine learning, 2024.

[54] Meta Platforms Inc. Zstandard, 2024.

[55] Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. Fp8 formats for deep learning, 2022.

[56] Mitzenmacher, M. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems* (2018), S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc.

[57] Moerkotte, G. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA* (1998), A. Gupta, O. Shmueli, and J. Widom, Eds., Morgan Kaufmann, pp. 476–487.

[58] Motorola INC. Motorola m68000 family programmer's reference manual. `https://www.nxp.com/docs/en/reference-manual/M68000PRM.pdf`, 1992.

[59] Neuvonen, S., Wolski, A., Manner, M., and Raatikka, V. Telecommunication application transaction processing (tatp) benchmark. `http://tatpbenchmark.sourceforge.net/`, 2011.

[60] New York City Metropolitan Transportation Authority. Car toll rates, 2021.

[61] New York City Taxi and Limousine Commission. Tlc trip record data - yellow taxi trip records, 2022, 2022.

[62] Nuzman, D., and Henderson, R. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)* (2006), pp. 11 pp.–294.

[63] O'Neil, P., O'Neil, E., Chen, X., and Revilak, S. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking* (Berlin, Heidelberg, 2009), R. Nambiar and M. Poess, Eds., Springer Berlin Heidelberg, pp. 237–252.

[64] O'Neil, P., and Quass, D. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1997), SIGMOD '97, Association for Computing Machinery, p. 38–49.

[65] Özcan, F., Tian, Y., and Tözün, P. Hybrid transactional/analytical processing. In *Proceedings of the 2017 ACM International Conference on Management of Data* (May 2017), ACM.

[66] Patel, J. M., Deshmukh, H., Zhu, J., Potti, N., Zhang, Z., Spehlmann, M., Memisoglu, H., and Saurabh, S. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow. 11*, 6 (oct 2018), 663–676.

[67] Polychroniou, O., Raghavan, A., and Ross, K. A. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, Association for Computing Machinery, p. 1493–1508.

[68] Pomeranz, J. B. The dice problem—then and now. *The Two-Year College Mathematics Journal 15*, 3 (1984), 229–237.

[69] Power, J., Li, Y., Hill, M. D., Patel, J. M., and Wood, D. A. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (May 2015), ACM.

[70] Prammer, M., and Patel, J. M. Rethinking the encoding of integers for scans on skewed data. *Proc. ACM Manag. Data 1*, 4 (dec 2023).

[71] Prammer, M., Rajesh, S. S., Chen, J., and Patel, J. M. Introducing a query acceleration path for analytics in sqlite3. In *Conference on Innovative Data Systems Research (CIDR)* (2022), CIDR '22.

[72] Raasveldt, M., and Mühleisen, H. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1981–1984.

[73] Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., and Sidle, R. Constant-time query processing. In *2008 IEEE 24th International Conference on Data Engineering* (2008), pp. 60–69.

[74] Rice, R., and Plaunt, J. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology 19*, 6 (1971), 889–897.

[75] Richardson, N., Cook, I., Crane, N., Dunnington, D., François, R., Keane, J., Moldovan-Grünfeld, D., Ooms, J., and Apache Arrow. arrow: Integration to 'apache' 'arrow', 2022. https://github.com/apache/arrow/, https://arrow.apache.org/docs/r/.

[76] Rinfret, D., O'Neil, P., and O'Neil, E. Bit-sliced index arithmetic. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2001), SIGMOD '01, Association for Computing Machinery, p. 47–57.

[77] Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, Association for Computing Machinery, p. 273–287.

[78] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal 27*, 3 (1948), 379–423.

[79] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal 3*, 5 (Oct. 2016), 637–646.

[80] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005), VLDB '05, VLDB Endowment, p. 553–564.

[81] Taylor, P. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025, Nov. 2023.

[82] The Transaction Processing Council. Tpc-h benchmark (version 3.0.1), 2022.

[83] Tomasulo, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development 11*, 1 (1967), 25–33.

[84] UK Power Networks. Smartmeter energy consumption data in london households, 2014.

[85] Van der Meulen, S., Gaeremynck, A., and Willekens, M. Attribute differences between u.s. gaap and ifrs earnings: An exploratory study. *The International Journal of Accounting 42*, 2 (2007), 123–142.

[86] von Neumann, J. First draft of a report on the edvac. *IEEE Annals of the History of Computing 15*, 4 (1993), 27–75. (Note: This is a reprint of the original 1945 document, digitized in 1992 by Michael D. Godfrey and published in 1993.).

[87] Wang, S., and Kanwar, P. Bfloat16: The secret to high performance on cloud tpus, Aug 2019.

[88] Willhalm, T., Oukid, I., Müller, I., and Faerber, F. Vectorizing database column scans with complex predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2013), Riva del Garda, Trento, I, August 26, 2013.* (2013), p. 1–12.

[89] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow. 2*, 1 (aug 2009), 385–394.

[90] Wu, L., Sharifi, R., Lenjani, M., Skadron, K., and Venkat, A. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), pp. 251–264.

[91] Wu, L., Sharifi, R., Venkat, A., and Skadron, K. Dram-cam: General-purpose bit-serial exact pattern matching. *IEEE Computer Architecture Letters 21*, 2 (2022), 89–92.

[92] Wulf, W. A., and McKee, S. A. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News 23*, 1 (mar 1995), 20–24.

[93] Xi, S. L., Augusta, A., Athanassoulis, M., and Idreos, S. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2015), DaMoN'15, Association for Computing Machinery.

[94] Zaharia, M., Ghodsi, A., Xin, R., and Armbrust, M. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings* (2021), www.cidrdb.org.

[95] Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., and Zhang, H. An empirical evaluation of columnar storage formats. *Proc. VLDB Endow. 17*, 2 (oct 2023), 148–161. (Note: An extended version of the paper is available on arXiv: https://arxiv.org/abs/2304.05028).

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

[97] Zhu, J., Potti, N., Saurabh, S., and Patel, J. M. Looking ahead makes query plans robust. *Proceedings of the VLDB Endowment 10*, 8 (Apr. 2017), 889–900.

[98] Zukowski, M., van de Wiel, M., and Boncz, P. Vectorwise: A vectorized analytical dbms. In *2012 IEEE 28th International Conference on Data Engineering* (2012), pp. 1349–1350.