Towards Functional Decomposition of Storage Formats

Martin Prammer*
Carnegie Mellon University
mprammer@cs.cmu.edu

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Ruijun Meng
Tsinghua University
mrj21@mails.tsinghua.edu.cn

Wes McKinney
Posit PBC
wes@posit.co

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Jignesh M. Patel
Carnegie Mellon University
jigneshp@cs.cmu.edu

ABSTRACT

The rise of data lakes containing mostly semi-structured and unstructured data has changed how traditional data platforms interact with collections of stand-alone files. Horizontally partitioned arrays are a fundamental construction in these columnar-like file formats, such as those partitioned into a column-chunk, row-group hierarchy (e.g., Parquet, ORC). Compressing each horizontal partition results in storage savings. Simultaneously, row-skipping metadata is a popular, lightweight indexing technique for accelerating columnar scans. Thus, existing storage-layer partitions are also used for general-purpose search acceleration. However, no single horizontal partition size optimizes both compressibility and row-skipping-driven scan performance.

Instead of settling for a suboptimal configuration, we return to the age-old wisdom of physical data independence: data should be kept separate from indexing structures. We propose splitting the current status quo into a "storage layer" and a "search acceleration layer" (SAL). By splitting these layers, row-skipping metadata is no longer stuck using the same partition sizes as compression blocks, allowing for fine-grained tuning of the SAL. In this paper, we explore the impact of such a split; not only do we find that search acceleration metadata is regularly optimal at small partition sizes (10-100), but also that optimal sizing depends on the metadata type, underlying data, and applied query. By separating the storage layer and SAL, we enable each to evolve independently, allowing for greater flexibility as datasets and application needs evolve.

1 INTRODUCTION

Over the last decade, there has been considerable interest in tabular data storage formats like Parquet [27] and ORC [30] to aid interoperability between data platforms [11, 13, 25, 32, 36]. The disaggregation of storage and compute in the cloud has made these data formats even more important [2, 10, 18, 31, 35]. 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 [14, 15, 33]. We also find index structures such as small materialized aggregates (SMA, also known as zone maps) and bloom filters stored directly adjacent to data [3, 9, 18, 24, 27, 30, 34]. 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.

Many of these file formats horizontally partition columns into nearly constant-length "chunks" or "blocks," each of which is independently compressed and annotated with search acceleration metadata [13, 27, 30, 34]. 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 [16] and Lance [34] 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 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 depends on the underlying data, an optimal block size 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 [5], 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

*Work performed while at the University of Wisconsin-Madison.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2025. 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

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 data 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 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). From this foundation, we then explore in Section 5 the impact of separated search acceleration and storage layers on a real-world dataset.

2 BACKGROUND

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

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 [16, 17, 27, 30, 34]. Many of these techniques are inspired by the PAX data layout [1]. 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 [13, 16]. 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, in many cases, compressed partitions must be entirely decompressed to access individual elements [13, 27, 34], such as when each partition is compressed with commodity compression libraries such as ZStandard [21], zlib [19], or Snappy [8]. Thus, storing metadata describing the data held within the partition becomes beneficial in enabling predicate pushdown without decompressing the underlying block.

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* [3, 6, 7, 9, 22, 24, 28]. 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 [21, 27, 34].

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 [20, 28]. This process is generally performed via either an optimization algorithm or a set of heuristics [23]. 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.

3 OBSERVATIONS

In this section, we evaluate the impact the horizontal partition size of a blocked array has on both compressibility (Section 3.1) and performance of search acceleration metadata (Section 3.2).

3.1 Optimizing Block Sizes for Compressibility

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

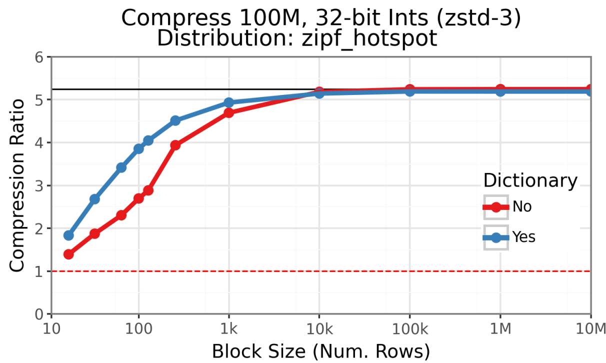

Figure 1: The compression ratio of the "hotspot" column, by block size and if a global dictionary was used.

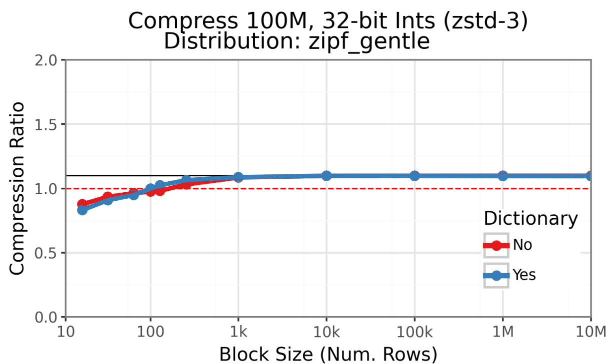

Figure 2: The compression ratio of the "gentle" column, by block size and if a global dictionary was used.

larger block sizes, there are only soft guidelines for minimum effective block sizes [12]. 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) [21] 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 [15]. 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 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 ($CLevel = 3$).

First, we compress a "Hotspot" ($p = 1.75$) Pseudo-Zipfian distributed column, consisting of unsigned 32-bit integers over the ordered values [1, 100M] [37]. 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 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

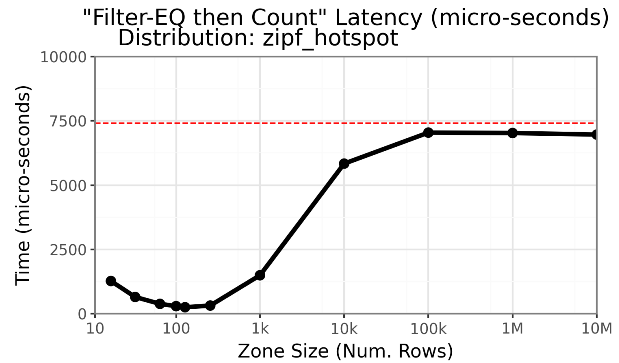

Figure 3: The time to perform a scan-based filter on the "hotspot" column, by block size.

(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 ($p = 0.5$) using the same underlying values as the Hotspot column, shown in Figure 2. In this case, while the column is closely related to a dictionary-encoded column with 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.

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, 100k). As the distributions are skewed towards zero, few records will satisfy this predicate: on average, 1 per hotspot 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] Thus, for each column of 32-bit integers, each block is augmented with 40B of metadata: 8B per SMA, 32B per bloom filter.

[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.

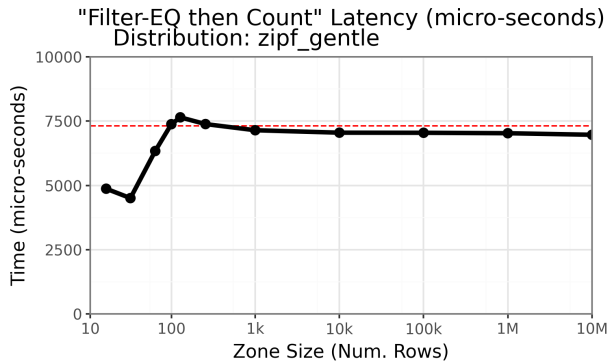

Figure 4: The time to perform a scan-based filter on the "gentle" column, by block size.

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 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. While the smallest block sizes show 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 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.

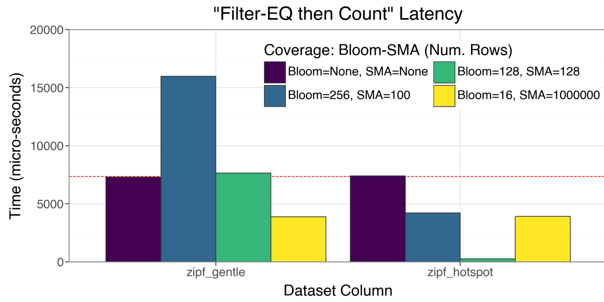

Figure 5: The performance of four different SAL configurations for the synthetic pseudo-zipfian distributions. The four configurations were selected from the best and worst configurations of each distribution.

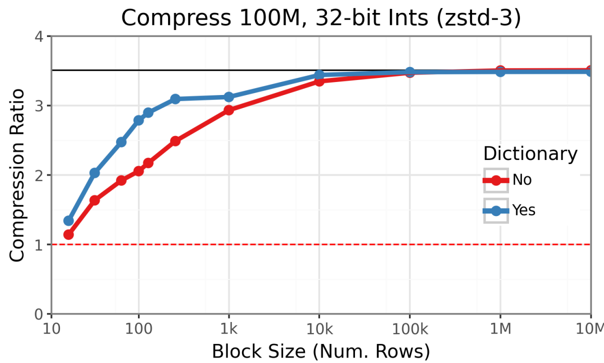

Figure 6: The compression ratio of the evaluated taxi dataset columns, by block size and if a global dictionary was used.

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 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.

5 EXPERIMENTAL RESULTS

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

5.1 Synthetic Distribution Evaluation

We repeat the experiment from Section 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 ("Bloom=16, SMA=16," "Bloom=16, SMA=32," etc.). We depict our findings in Figure 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 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.

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 [26]. 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 parameters previously explored with the synthetic distributions (Section 3.1). These results are depicted in Figure 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%)

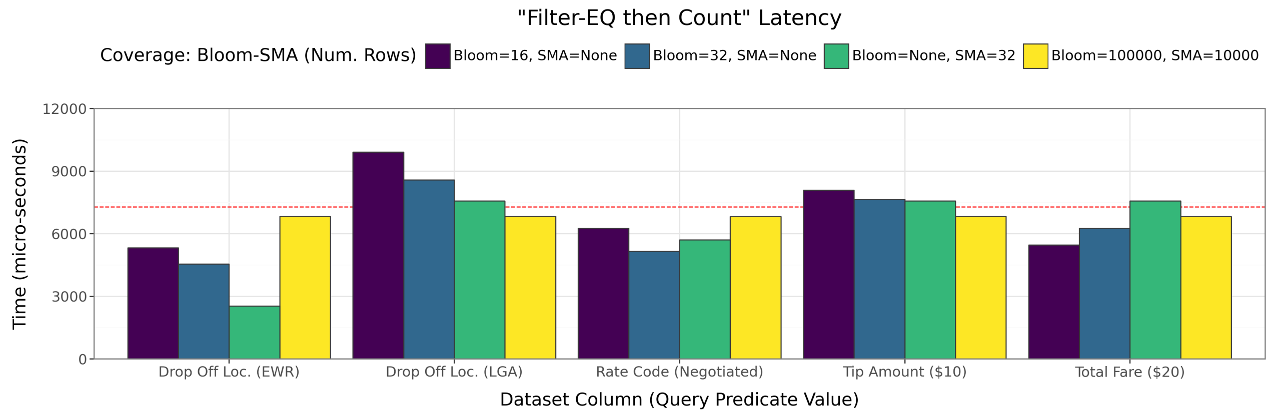

Figure 7: The performance of four different SAL configurations for the explored taxi dataset columns. The four configurations were selected from the best configurations for each distribution.

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 7. If a configuration is best for a data set, we include this configuration's performance for all the data sets. For example, "Bloom=32, SMA=None" is the best configuration for the scan applied to 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. The effectiveness of the SAL is impacted not only by the column's data distribution but also by the applied predicate.

6 CALL TO ACTION

With no "one size fits most" option, these results seem somewhat bleak. Yet, these results also illustrate a clear path forward.

First, the scan latency reduction of search acceleration metadata like bloom filters and SMA depends on the data distribution of the underlying columnar data. This benefit comes at the cost of the size of the SAL, which must be stored and processed to facilitate row-skipping. Thus, incurring this additional processing cost to leverage summary structures must be a conscious decision. In some cases, SAL-based row-skipping does not compensate for the additional cost of processing significant amounts of search acceleration metadata, resulting in the SAL increasing scan latency.

Further, each search acceleration metadata requires a coverage size that is small enough to capture variance within a larger dataset. If the data closely mirrors low-skew distributions, finding meaningful differences between horizontal partitions can only be achieved at small coverage sizes. While not the focus of this paper, the central limit theorem from probability theory influences our results; under the right circumstances, a large enough coverage is analogous to a set of samples from an underlying distribution, resulting in the summary statistics themselves trending towards a normal distribution [4, 29]. This concept elucidates the push-pull relationship between compressibility and search acceleration efficacy; while more uniquely identifiable covered data may reduce scan latency, increasing the distinction between partitions impedes partition-shared compression dictionaries, which instead benefit from uniformity.

Together, these considerations incentivize the specialization of storage formats and search acceleration layers. Only after decoupling search acceleration metadata coverage sizes from compression-optimized partition sizes can search acceleration layers be more flexibly optimized. While we only explored varying coverage between bloom filters and SMA, one could conceivably use heterogeneously sized search acceleration metadata structures within a single column for further opportunities to reduce scan latency.

7 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. 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.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation (NSF) under grant CCF #2407690.

REFERENCES

[1] Anastassia Ailamaki et al. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[2] Michael Armbrust et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[3] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[4] L. Le Cam. 1986. The Central Limit Theorem Around 1935. *Statist. Sci.* 1, 1 (1986), 78–91. http://www.jstor.org/stable/2245503

[5] Edgar F. Codd. 1985. Is your DBMS really relational? https://reldb.org/c/index.php/twelve-rules/ (Note: The original document has been informally preserved by Dave Voorhis, 2019).

[6] Benoit Dageville et al. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[7] Databricks. 2024. Bloom filter indexes. https://docs.databricks.com/en/optimizations/bloom-filters.html

[8] Google. [n. d.]. Snappy. https://google.github.io/snappy/ Accessed: November 12, 2024.

[9] Goetz Graefe. 2009. Fast loads and fast queries. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 111–124.

[10] Anurag Gupta et al. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[11] Rihan Hai, Christos Koutras, Christoph Quix, and Matthias Jarke. 2023. Data Lakes: A Survey of Functions and Systems. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12571–12590. https://doi.org/10.1109/TKDE.2023.3270101

[12] Yongqiang He et al. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*. 1199–1208. https://doi.org/10.1109/ICDE.2011.5767933

[13] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[14] Svilen Kanev et al. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[15] Sagar Karandikar et al. 2023. CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 39, 17 pages. https://doi.org/10.1145/3579371.3589074

[16] Maximilian Kuschewski et al. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2, Article 118 (jun 2023), 26 pages. https://doi.org/10.1145/3589263

[17] Gang Liao et al. 2024. Bullion: A Column Store for Machine Learning. arXiv:2404.08901 [cs.DB] https://arxiv.org/abs/2404.08901

[18] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (July 2023), 3044–3056. https://doi.org/10.14778/3611479.3611507

[19] Jean loup Gailly, Mark Adler, and Greg Roelofs. 2024. zlib. https://zlib.net/

[20] Gabriel Mersy, Zhuo Wang, Stavros Sintos, and Sanjay Krishnan. 2024. Optimizing Collections of Bloom Filters within a Space Budget. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3551–3564. https://doi.org/10.14778/3681954.3682020

[21] Meta Platforms Inc. [n. d.]. Zstandard. https://facebook.github.io/zstd/ Accessed: June 12, 2024.

[22] Michael Mitzenmacher. 2001. Compressed bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, USA) *(PODC '01)*. Association for Computing Machinery, New York, NY, USA, 144–150. https://doi.org/10.1145/383962.384004

[23] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf

[24] Guido Moerkotte. 1998. 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*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[25] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data lake management: challenges and opportunities. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[26] New York City Taxi and Limousine Commission. 2023. TLC Trip Record Data - Yellow Taxi Trip Records, 2023. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[27] Apache Parquet. 2024. Apache Parquet. https://parquet.apache.org Accessed: June 12, 2024.

[28] Apache Parquet. 2024. Apache Parquet Documentation: Bloom Filter. https://parquet.apache.org/docs/file-format/bloomfilter/ Accessed: June 12, 2024.

[29] Janet Bellcourt Pomeranz. 1984. The Dice Problem—Then and Now. *The Two-Year College Mathematics Journal* 15, 3 (1984), 229–237. https://www.tandfonline.com/doi/abs/10.1080/00494925.1984.11972775

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

[31] Raghu Ramakrishnan et al. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 51–63. https://doi.org/10.1145/3035918.3056100

[32] Franck Ravat and Yan Zhao. 2019. Data lakes: Trends and perspectives. In *Database and Expert Systems Applications: 30th International Conference, DEXA 2019, Linz, Austria, August 26–29, 2019, Proceedings, Part I 30*. Springer, 304–313.

[33] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 733–750. https://doi.org/10.1145/3373376.3378450

[34] LanceDB Development Team. 2024. LanceDB: A High-Performance Database System. https://www.lancedb.org. Accessed: 2024-06-12.

[35] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. 2010. Cloud Storage as the Infrastructure of Cloud Computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*. 380–383. https://doi.org/10.1109/ICICCI.2010.119

[36] Matei Zaharia et al. 2021. 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*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[37] Xinyu Zeng et al. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (oct 2023), 148–161. https://doi.org/10.14778/3626292.3626298 (Note: An extended version of the paper is available on arXiv: https://arxiv.org/abs/2304.05028).