GPU Acceleration of SQL Analytics on Compressed Data

Zezhou Huang
Microsoft
zacharyhuang@microsoft.com

Krystian Sakowski
Microsoft
krsakows@microsoft.com

Hans Lehnert
Microsoft
hans.lehnert@microsoft.com

Wei Cui
Microsoft
weicu@microsoft.com

Carlo Curino
Microsoft
carlo.curino@microsoft.com

Matteo Interlandi
Microsoft
mainterl@microsoft.com

Marius Dumitru
Microsoft
mariusd@microsoft.com

Rathijit Sen
Microsoft
rathijit.sen@microsoft.com

ABSTRACT

GPUs are uniquely suited to accelerate (SQL) analytics workloads when datasets fit in the GPU High Bandwidth Memory (HBM). Unfortunately, GPU HBMs remain typically small when compared with lower-bandwidth CPU main memory. Current solutions to accelerate queries on large datasets include multi-GPU execution, processing smaller data batches, and hybrid execution with a connected device (e.g., CPUs). Unfortunately, these approaches are exposed to the limitations of lower main memory and host-to-device interconnect bandwidths, introduce additional I/O overheads, or incur higher costs. This is a substantial problem when trying to scale adoption of GPUs on larger datasets. Data compression can alleviate this bottleneck, but to avoid paying for costly decompression/decoding, an ideal solution must include computation primitives to operate directly on data in compressed form.

This is the focus of our paper: a set of new methods for running queries directly on light-weight compressed data using schemes such as Run-Length Encoding (RLE), index encoding, bit-width reductions, and dictionary encoding. Our novelty includes operating on multiple RLE columns without decompression, handling heterogeneous column encodings, and leveraging PyTorch tensor operations for portability across devices. Experimental evaluations show speedups of an order of magnitude compared to state-of-the-art commercial CPU-only analytics systems, for real-world queries on a production dataset that would not fit into GPU memory uncompressed. This work paves the road for GPU adoption in a much broader set of use cases, and it is complementary to most other scale-out or fallback mechanisms.

PVLDB Reference Format:
Zezhou Huang, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. GPU Acceleration of SQL Analytics on Compressed Data. PVLDB, 19(3): 320 - 333, 2025.
doi:10.14778/3778092.3778095

1 INTRODUCTION

The massive compute parallelism and multi-TB/sec device memory bandwidths of modern GPUs make them attractive for accelerating SQL analytics queries [38, 42]. However, GPU high-bandwidth memory (HBM) capacity is much less, often by integer factors or an order of magnitude, compared to CPU main memory in high-end servers. This restricts dataset sizes processable on GPUs without requiring data movement over slow CPU-GPU interconnects, CPU-GPU hybrid processing [39], or using multiple GPUs [12, 30, 31, 40, 52]—which increases costs.

One approach to alleviate this bottleneck is to run queries on compressed data, which is the focus of this paper. We focus on light-weight encodings for compression, in particular, run-length encoding (RLE) for consecutively repeating values, index encoding for sparse data, dictionary encoding, and bit-width reductions. Our methods not only allow GPU query processing on datasets that would not fit uncompressed in the GPU HBM, but also achieve high performance by leveraging redundancy eliminated by compression.

Being able to operate directly on RLE data provides both space and time advantages. The representation uses constant space, *regardless of the length of the run*. In contrast, an uncompressed representation uses space that is directly proportional to the number of values, that is, the length of the run. This reduces GPU HBM capacity requirements. Operating on RLE data can also be faster. For example, a SUM operation over an RLE data column would need to multiply the run values with the run lengths, then add the results over all runs, whereas in a plain/uncompressed representation, every value in the column needs to be scanned. Additionally, the space savings with RLE reduces I/O overheads for data movement between the GPUs and other devices, e.g., CPU-GPU data transfers over the (relatively) low-bandwidth PCIe bus.

SQL query processing directly on RLE data, however, is challenging when operations involve multiple data columns. This is because of misalignment in runs between columns that can occur due to different data distributions as well as effect of predicate filters on different columns. For example, SUM (A+B) on two RLE columns A and B is not straightforward to implement since runs in A and B can start and end at different positions, making a direct step of adding the run values between the two columns impossible. While this can be handled with an iterative approach that loops through the different runs and keeps track of their start and end positions, such an approach is inefficient on parallel computational backends such as GPUs due to severe under-utilization of resources. Additional complexities arise with heterogeneous encoding schemes across columns, e.g., if A uses RLE, but B uses an index representation that is more suitable for non-repetitive sparse data.

While prior works [4, 28, 43] have recognized and exploited the space-savings advantages of RLE data, they usually decompress the data in the GPU memory hierarchy, shared memory, and registers before using them as inputs to relational operators. Some works study the potential for query processing over partially compressed

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778095

data, but focus mostly on specific operations such as table scan [15] or join [55] without providing a comprehensive framework for all relational operators. In contrast, we develop *novel parallel algorithms and a framework for SQL relational operators to: (1) directly operate on encoded data as much as possible without decompressing it, and (2) be efficiently implemented on parallel devices such as GPUs.* Our methods provide query run time speedups of an order of magnitude compared to state-of-the-art CPU-only engines for representative production queries on a production dataset that is larger than the HBM capacity of a single A100 GPU in uncompressed form. To make our implementations easily portable to multiple accelerators, we follow TQP [17, 18] and implement SQL queries using PyTorch tensor programs, but extend it to show how PyTorch functions can be used to implement relational operators for compressed data.

In summary, we extend the state of the art in the following ways.

- We show how SQL operations can be performed on data encoded for compression, particularly RLE data, in GPUs *without expanding as much as possible*, in contrast to traditional approaches.
- We demonstrate for the first time how this can be done in TQP with PyTorch tensor operations.
- We show query run time speedups of an order of magnitude compared to state-of-the-art CPU-only database and analytical systems on a real-world dataset that is too large to fit into the HBM of a high-end GPU without compression.

As far as we know this is the first work showcasing a comprehensive framework allowing to execute complex queries (e.g., including joins, group by, aggregation, projection and filter operators) end to end on lightweight-compressed data in GPU.

2 BACKGROUND

We first describe TQP, our query execution framework, then introduce the PyTorch primitives used in our compressed operators.

2.1 TQP

The Tensor Query Processor (TQP) [6, 18] converts SQL queries into tensor programs, then runs them using Tensor Computation Runtimes (TCRs), such as PyTorch [37], on hardware backends such as GPUs [8, 17, 18] and APUs [10]. This makes TQP easily portable to different devices while at the same time benefiting from the optimizations provided by the TCRs. In this work, we continue to use TQP's approach to get the same portability and performance advantages, but extend it to show how we can convert queries to tensor programs that can operate on lightweight-compressed data.

TQP supports input tabular data from files in Parquet [16] and CSV formats, and from in-memory formats such as NumPy [47] and Pandas DataFrames [32]. TQP converts each input column into PyTorch tensors. The conversion is straightforward for numeric and date columns. TQP does not currently support decimals, and instead represents them using floating-point numbers. ASCII string columns are value-encoded and dictionary-encoded. At the end of the conversion step, done offline before running queries, every column is represented using one or more numeric PyTorch tensors.

Given an input SQL query, TQP uses a query optimizer (currently, Apache Spark's [56] Catalyst optimizer) to obtain a physical query plan. It then converts it into an equivalent tensor program [18]. It loads tensors corresponding to the query input columns into the device memories—for GPUs, this is the HBM which is available as global memory on the GPUs. TQP loads and operates on entire columns, spanning all rows of the table, rather than splitting them up into smaller chunks. This is beneficial for good performance as it avoids overheads of repeated sets of kernel launches and improves GPU resource utilization, but requires the full column tensor (in addition to temporary results) to fit in the available HBM capacity. In this work, we continue with TQP's approach of loading and operating on full columns, but additionally allow compact representation for encoded data, thereby reducing the HBM requirements.

2.2 PyTorch Primitives

Tensors are the fundamental data structures in PyTorch. While tensors can be high-dimensional (for ML applications), in this work we use them as one-dimensional arrays that directly correspond to database columns. Our algorithms rely on several PyTorch primitives that have been highly optimized for ML tasks but proves effective for database query processing.

Tensor Indexing []: The bracket operator supports three modes: (1) single indexing (e.g., $data[0]$ selects the element at position 0), (2) multi-indexing with index tensors (e.g., $data[[0, 2]]$ selects elements at positions 0 and 2), and (3) filtering with boolean masks (e.g., $data[mask]$ returns elements where mask is True). For example, given $data = [10, 20, 30, 40]$: $data[[0, 2]]$ returns $[10, 30]$; $data[[True, False, True, False]]$ also returns $[10, 30]$.

bucketize(input, boundaries, right): Performs binary searches for bucket indices. With $right = True$, element j goes to bucket i if $boundaries[i-1] \leq input[j] < boundaries[i]$; with $right = False$, if $boundaries[i-1] < input[j] \leq boundaries[i]$. For example, $bucketize([1, 5, 3], [2, 4, 5])$ returns $[0, 3, 1]$ with $right = True$ ($1<2$, $5\geq5$, $2\leq3<4$) and $[0, 2, 1]$ with $right = False$ ($1\leq2$, $4<5\leq5$, $2<3\leq4$).

arange(end): Generates a tensor with values from 0 to $end - 1$. For example, $arange(5)$ returns $[0, 1, 2, 3, 4]$.

repeat_interleave(input, repeats): Repeats each element of *input* consecutively by the corresponding count in *repeats*. For example, $repeat\_interleave([10, 20], [2, 1])$ returns $[10, 10, 20]$.

unique(input): Returns unique elements and inverse indices mapping original elements to their positions. For example, $unique([3, 1, 3, 2])$ returns unique $[1, 2, 3]$ and inverse indices $[2, 0, 2, 1]$.

scatter(src, index, reduce): Accumulates values from *src* at positions specified by *index* using the *reduce* operation. Multiple values targeting the same index are combined according to the *reduce*. For example, $scatter([10, 20, 30], [0, 1, 0], reduce = sum)$ results in $[40, 20]$ (10+30 at index 0, 20 at index 1).

3 ENCODINGS FOR COMPRESSION AND THEIR TENSOR REPRESENTATIONS

In order to enable query processing on compressed data, we expand the encodings supported in TQP. We add support for two additional light-weight encoding techniques—Run-Length Encoding (RLE) and indexing. RLE is useful for compactly representing sequences of consecutively repeating values. We use indexing in novel ways, including to separate outliers and enable compression through bit-width reduction, and for efficient representation of sparse data. Currently we do not support operations on data in heavy-weight compression formats, e.g., Snappy, zstd, LZ4, gzip, etc.

3.1 Basic Encodings

We will discuss multiple columns, each with multiple fields for their tensor representations. For notation clarity: we use subscripts to distinguish different columns (e.g., c_1, c_2), and superscripts for field access within each column's encoding (e.g., c_1^{start}, c_2^{pos}).

First, we list the types of basic encodings for columns that we support and their tensor representations. These are applied on top of any dictionary encodings of values, e.g., for string columns.

- **Plain**: This is the existing tensor representation used by TQP as we discussed in Section 2.1. For every column, there is a 1:1 mapping from a row position in the column to the corresponding position in the tensor representing the values in the column.
- **RLE**: We use three tensors to represent a set of RLE runs: (1) values (c^{val}), (2) start positions (c^{start}), and (3) end positions (c^{end}). Tensors c^{val}, c^{start}, and c^{end} are of equal length, and the i^{th} entry represents a run with value $c^{val}[i]$ spanning row numbers $c^{start}[i] \rightarrow c^{end}[i]$. The positions are zero-based and unique. Tensors c^{val}, c^{start}, c^{end} are sorted by c^{start} (equivalently, by c^{end}). Note that, we could have used a tensor for run lengths (c^{len}) instead of for end positions (c^{end}). We can choose either lengths or end positions based on convenience, and calculate the other at run time using the equation: $c^{len} = c^{end} - c^{start} + 1$. Runs are non-overlapping by position.
- **Index**: We use two tensors to represent index encoding: (1) values (c^{val}), and (2) positions (c^{pos}). Tensors c^{val} and c^{pos} are of equal length, sorted by c^{pos}, and the i^{th} entry represents value $c^{val}[i]$ at row number $c^{pos}[i]$. The positions are zero-based and unique.

While Plain encoding has an implicit 1:1 correspondence between row numbers and positions in the tensor representation, RLE and Index encoding explicitly track row numbers. This also allows for efficient representation when there are gaps in the underlying data, e.g., when some portions of the column are deselected after application of filter predicates. Thus, for RLE, for two consecutive entries at tensor positions i and $i+1$, we have $c_{i+1}^{start} \geq c_i^{end} + 1$. Similarly, for Index encoding, we have $c_{i+1}^{pos} \geq c_i^{pos} + 1$. In contrast, Plain encoding does not allow for gaps in the tensor representation.

3.2 Composite Encodings

In addition to basic encodings, we also introduce the following novel composite encodings that combine Index with Plain and RLE to enable further compression.

- **Plain + Index**: PyTorch requires a uniform data type for all elements within a single tensor. However, outliers can force the entire tensor to use a larger data type than necessary. To enable bit-width reduction with narrower data types, we can combine Plain with Index encoding. We store most values in one PyTorch tensor with a narrower type and represent the outliers separately using Index encoding. At positions corresponding to outliers, the Plain tensor contains uninitialized PyTorch tensor values (values happened to be in that memory location). These uninitialized values are never used during query execution.
- **RLE + Index**: RLE works best for continuous value segments. However, some columns may contain both continuous (pure) and non-continuous (impure) segments. While impure segments can be represented by a series of unit-length RLE runs, this can be inefficient since each run is represented by three elements (value, start, end). Instead, we can handle impure segments with Index encoding. The Index positions and the RLE intervals are disjoint in this composite encoding.

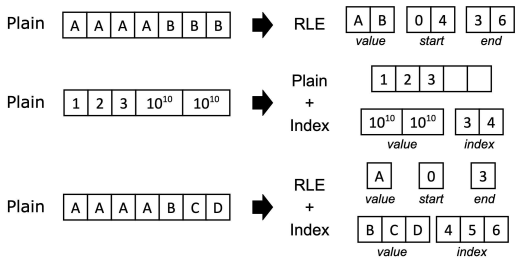

Figure 1: GPU-Optimized Tensor Data Representations.

Note that our bit-width reduction differs from traditional CPU compression techniques. While CPU approaches break columns into small groups to optimize bit width and combine with techniques like DELTA and FOR [1, 2, 57], the small groups suit poorly for GPU processing. Instead, we leverage PyTorch's preference for large tensors by using index-based outlier separation. After removing outliers, we apply centering to reduce bit width. Centering is similar to FOR, but instead of local reference values (typically minimum in each group) we use a global reference value of the mid-range for the entire column. This allows us to represent most values with narrower bit widths while handling outliers separately, achieving good compression without sacrificing GPU execution efficiency.

Example 1. *Consider the various optimized Tensor representations for Data Compression in Figure 1.*

Plain → RLE example: The sequence $[A, A, A, A, B, B, B]$ is compressed using run-length encoding. A repeats from positions 0 to 3, and B repeats from positions 4 to 6. We store this as $v = [A, B], s = [0, 4], e = [3, 6]$. This captures exactly where each value run begins and ends.

Plain → Plain + Index example: When handling a column with values $[1, 2, 3, 10^{10}, 10^{10}]$, the outliers at indices 3 and 4 would force the entire tensor to use a larger data type. We separate the data: regular values in Plain tensor $[1, 2, 3]$ and outliers in a separate value tensor $[10^{10}, 10^{10}]$ with their positions in index tensor $[3, 4]$. In a dataset with 1 billion elements, if a few outliers require int64 but most values fit in int8, Plain representation would force all values into int64, using 8 GiB of memory (8 bytes × 1 billion). With Plain + Index, we store most values in int8 (1 GiB) plus a small overhead for outliers and their indices, saving ~7 GiB of memory while preserving all data values.

Plain → RLE + Index example: The sequence $[A, A, A, A, B, C, D]$ has a consecutive run of A (positions 0-3) but B and C do not form runs. We use RLE for the A run ($v = [A], s = [0], e = [3]$) and Index encoding for the scattered values ($v = [B, C, D], p = [4, 5, 6]$).

3.3 Data and Mask column representations

Our encoding schemes are versatile and can be used to represent: (1) data columns containing actual values from database tables and intermediate results, (2) mask columns that encode boolean predicates for selections and filters, and (3) internal data structures required by query operators. This unified representation approach

allows our system to maintain a consistent execution model while adapting to different data characteristics and query requirements.

While encoding approaches for data and mask columns are similar, there are a few differences in their representations as follows.

- The domain of values for masks is restricted to $\{T, F\}$.
- Since there are no outliers in values, the composite Plain + Index encoding is not applicable for masks.
- In position-explicit encodings (RLE, Index) for masks, only positions corresponding to T values are maintained in the tensor representations. Plain encoding tracks both T and F values.
- In position-explicit encodings for masks, the value tensor (v) is not needed, since all tracked values are implicitly equal to T.

Depending on the operator type, the operands can be data columns, mask columns, scalar expressions, or literals. Result columns can be data or masks. We will use the terms DataColumn and MaskColumn where needed to distinguish between these two types.

4 PRIMITIVES

We develop a set of novel parallel algorithms to implement fundamental operations on encoded data types. These operations include conversion between encodings, detecting containment and overlap between position-explicit encodings, positional range intersection and union, compaction, etc. Table 1 lists the set of non-trivial operations. These operations can be used as building blocks to implement more complex operations. In Sections 5–8 we describe how we implement relational operators using these fundamental operations.

Table 1: Fundamental Operations for Encoded data.

Primitive	Description
`range_intersect`	Intersection of RLE runs
`idx_in_rle`	Intersection of Index and RLE data
`idx_in_idx`	Intersection of Index columns
`rle_contain_idx`	Index positions contained within an RLE run
`range_union`	Union of RLE runs
`merge_sorted_idx`	Merge sorted Index tensors
`compact_rle`	Remove gaps between runs of RLE data
`compact_rle+index`	Remove gaps in RLE + Index data
`complement_rle`	Complement of RLE intervals
`complement_index`	Complement of Index positions
`rle_to_index`	Convert RLE data to Index
`rle_to_plain`	Convert RLE data to Plain
`plain_to_rle`	Convert Plain data to RLE
`plain_to_rle+index`	Convert Plain data to Composite (RLE+Index)
`plain_to_plain+index`	Convert Plain data to Composite (Plain+Index)

Due to space limitation, in this section we include pseudo-code for only a few primitives, notably the intersection primitives that are heavily used for implementing relational operators. Functions in **bold font** are provided by PyTorch and have been optimized by PyTorch developers for different backends including GPUs. Note that there are no explicit for loops or conditional checks in the codes, which helps to maximize utilization of the GPU parallelism.

4.1 Range Intersection

This operation finds the intersection of two sorted lists of ranges. We use this operation frequently in computations involving multiple RLE columns. At the end of the intersect, all columns will have the same number of runs and same start and end positions (same s and e tensors for all columns) corresponding to position ranges that are common to all input columns. If a range is split, the value is duplicated for the new ranges. Our goal is to minimize the number of range splits (i.e., maximize run lengths) in the result.

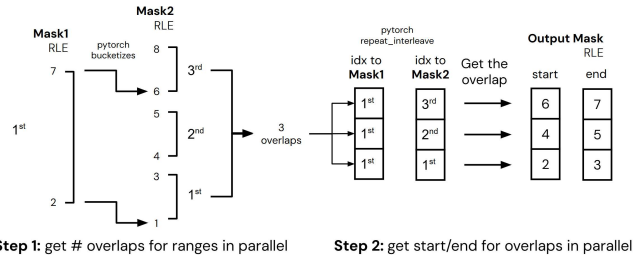

Figure 2: Illustration of range_intersect algorithm for AND between RLE masks: bucketization identifies overlaps, then parallel PyTorch operations compute precise intersections.

Our approach is inspired by bioinformatics techniques for chromosome range intersections [26], as detailed in Algorithm 1. The "range_intersect" algorithm efficiently computes the intersection of two RLE inputs, c_1 and c_2, by first bucketizing the start (c_1^{start}) and end (c_1^{end}) points of the first input relative to the second (Lines 1-2). It then counts intersections (Line 3) and generates index tensors idx_1 and idx_2 for each intersecting range (Lines 4-6), using the helper function `range_arange` (Algorithm 2) for idx_2. Unlike PyTorch's **arange**, which operates on single start/length values, `range_arange` accepts tensors of starts and lengths to generate multiple concatenated sequences. Finally, the algorithm determines the start and end points of the intersections (Line 7), producing the resulting tensors s and e. To optimize performance, the input with the fewer ranges should always be used as c_1.

Algorithm 1 range_intersect

Input: RLE columns c_1, c_2
Output: Intersection start and end positions s, e

1: $bin_s \leftarrow$ **bucketize**(c_1^{start}, c_2^{end}, *right=False*)
2: $bin_e \leftarrow$ **bucketize**(c_1^{end}, c_2^{start}, *right=True*)
3: $cnt \leftarrow bin_e$ - bin_s
4: $arange \leftarrow$ **arange**(**len**(cnt))
5: $idx_1 \leftarrow$ **repeat_interleave**($arange$, cnt)
6: $idx_2 \leftarrow$ `range_arange`(bin_s, cnt)
7: $s, e \leftarrow$ **max**($c_1^{start}[idx_1]$, $c_2^{start}[idx_2]$), **min**($c_1^{end}[idx_1]$, $c_2^{end}[idx_2]$)
8: **return** s, e

Algorithm 2 range_arange (helper function)

Input: *start, length*
Output: Concatenated sequence *result*

1: $t \leftarrow$ **cumsum**(*length*)
2: $t \leftarrow$ **cat**(([0], t[:-1])) // prepends 0, includes all but last element
3: *total_size* $\leftarrow$ **sum**(*length*)
4: *result* $\leftarrow$ **repeat_interleave**(*start, length*)
5: +**arange**(*total_length*)
6: -**repeat_interleave**(t, *length*)
7: **return** *result*

Example 2. *Consider the range intersection between the positional ranges of two RLE masks, c_1 and c_2, as shown in Figure 2. Mask c_1 has a single range represented as $c_1^{start} = [2]$, $c_1^{end} = [7]$. Mask c_2 has three ranges represented as $c_2^{start} = [1, 4, 6]$, $c_2^{end} = [3, 5, 8]$. Recall that masks only represent ranges corresponding to a value of T.*

To compute their intersection using range_intersect (Algorithm 1):

- **Step 1: Bucketize start positions.** We bucketize the start position (c_1^{start}) of Mask c_1 relative to the end positions (c_2^{end}) of Mask c_2. bin_s = **bucketize**($[2], [3, 5, 8]$) = $[0]$. This means the start point 2 comes before the first end point 3.
- **Step 2: Bucketize end positions.** We bucketize the end positions (c_1^{end}) of Mask c_1 relative to the start positions (c_2^{start}) of Mask c_2. bin_e = **bucketize**($[7], [1, 4, 6], right = True$) = $[3]$. This means the end point 7 comes after all start positions in c_2.
- **Step 3: Count overlaps.** We count the overlaps by subtracting bin_s from bin_e. $cnt = bin_e - bin_s = [3] - [0] = [3]$. This indicates that the range in c_1 overlaps with all 3 ranges in c_2.
- **Step 4: Create index tensors.** We create index tensors that identify which ranges from each mask participate in each overlap. idx_1 = **repeat_interleave**($[0], [3]$) = $[0, 0, 0]$ (indices into c_1) and idx_2 = **range_arange**($[0], [3]$) = $[0, 1, 2]$ (indices into c_2).
- **Step 5: Compute intersection points.** We compute the actual intersection points by taking the maximum of start points and minimum of end points. s = **max**($c_1^{start}[idx_1], c_2^{start}[idx_2]$) = **max**($[2, 2, 2], [1, 4, 6]$) = $[2, 4, 6]$ and e = **min**($c_1^{end}[idx_1], c_2^{end}[idx_2]$) = **min**($[7, 7, 7], [3, 5, 8]$) = $[3, 5, 7]$.

The output is an RLE mask with $s = [2, 4, 6]$ and $e = [3, 5, 7]$, representing the segments where both c_1 and c_2 have True values.

4.2 Index Intersection

These operations handle intersections for Index encoded data (c^{pos}). A common task is finding which positions (c_{idx}^{pos}) from an Index list fall within any RLE ranges ($c_{rle}^{start}, c_{rle}^{end}$). We provide two algorithms for this: `idx_in_rle` (Algorithm 3) and `rle_contain_idx` (Algorithm 5). The main computational work involves the `bucketize` operation performed by each. `idx_in_rle` uses `bucketize`($c_{idx}^{pos}, c_{rle}^{start}$), while `rle_contain_idx` uses `bucketize`($c_{rle}^{start}, c_{idx}^{pos}$) and `bucketize`($c_{rle}^{end}, c_{idx}^{pos}$). The choice between them for optimal performance depends on the relative input sizes: `idx_in_rle` is generally preferred when $|c_{idx}^{pos}| \ll |c_{rle}^{start}|$. We also provide `idx_in_idx` (Algorithm 4) for intersecting two Index lists.

Example 3 (`idx_in_rle`). *Let us consider* `idx_in_rle` *with an Index column c_{idx} and an RLE column c_{rle}. Suppose $c_{idx}^{pos} = [2, 4, 7]$ and $c_{rle}^{start} = [0, 6]$, $c_{rle}^{end} = [2, 7]$. We want to find which elements in c_{idx}^{pos} fall within the RLE ranges [0-2] or [6-7] defined by c_{rle}.*

- **Step 1: Bucketize positions.** `bucketize`($c_{idx}^{pos}, c_{rle}^{start}$, *right=True*) gives $[1, 1, 2]$. Subtracting 1 yields $bin = [0, 0, 1]$. This tells us which RLE range each position p *might* belong to (position 2 maps to range 0, 4 maps to range 0, 7 maps to range 1).
- **Step 2: Verify containment.** We check two conditions for each position p in c_{idx}^{pos}: (1) $bin \geq 0$ ensures the position is not before the first start s in c_{rle}^{start}, and (2) $p \leq c_{rle}^{end}[bin]$ ensures the position is not past the end e of its assigned RLE range. For position 2: bin=0. $0 \geq 0$ (T) and $2 \leq c_{rle}^{end}[0] = 2$ (T). 2 is included. For position 4: bin=0. $0 \geq 0$ (T) and $4 \leq c_{rle}^{end}[0] = 2$ (F). 4 is excluded. For position 7: bin=1. $1 \geq 0$ (T) and $7 \leq c_{rle}^{end}[1] = 7$ (T). 7 is included.
- **Step 3: Apply mask.** The resulting mask is $[T, F, T]$. Applying this to c_{idx}^{pos} gives the final result $p_{out} = [2, 7]$.

Algorithm 3 idx_in_rle

Input: Index column c_{idx}, RLE c_{rle}
Output: Result positions p_{out}

1: $bin \leftarrow$ **bucketize**($c_{idx}^{pos}, c_{rle}^{start}$, *right=True*)
2: $bin \leftarrow bin$ - 1
3: $mask \leftarrow (bin \geq 0) \wedge (c_{idx}^{pos} \leq c_{rle}^{end}[bin])$
4: **return** $c_{idx}^{pos}[mask]$

Algorithm 4 idx_in_idx

Input: Index columns c_1, c_2
Output: Result positions p_{out}

1: $bin \leftarrow$ **bucketize**(c_1^{pos}, c_2^{pos}, *right=True*) - 1
2: $mask \leftarrow (bin \geq 0) \wedge (c_1^{pos} = c_2^{pos}[bin])$
3: **return** $c_1^{pos}[mask]$

Algorithm 5 rle_contain_idx

Input: Index column c_{idx}, RLE column c_{rle}
Output: Result positions p_{out}

1: $bin_s \leftarrow$ **bucketize**($c_{rle}^{start}, c_{idx}^{pos}$)
2: $bin_e \leftarrow$ **bucketize**($c_{rle}^{end}, c_{idx}^{pos}$, *right=True*) - 1
3: $mask \leftarrow (bin_s \leq bin_e)$
4: $bin_s, bin_e \leftarrow bin_s[mask], bin_e[mask]$
5: **return** c_{idx}^{pos}[**rle_to_index**(bin_s, bin_e)]

Example 4 (`rle_contain_idx`). *Let us illustrate* `rle_contain_idx` *using the same input data as the* `idx_in_rle` *example for direct comparison. Suppose $c_{idx}^{pos} = [2, 4, 7]$ and the RLE column c_{rle} has ranges defined by $c_{rle}^{start} = [0, 6]$ and $c_{rle}^{end} = [2, 7]$. The ranges are [0-2] and [6-7]. We want to find which positions in c_{idx}^{pos} are contained within any of these RLE ranges.*

- **Step 1: Bucketize RLE starts.** We bucketize each RLE start position (c_{rle}^{start}) against the sorted index positions (c_{idx}^{pos}). bin_s = **bucketize**($c_{rle}^{start}, c_{idx}^{pos}$) = **bucketize**($[0, 6], [2, 4, 7]$) = $[0, 2]$. (0 is before index 0 value '2'; 6 is before index 2 value '7').
- **Step 2: Bucketize RLE ends.** We bucketize each RLE end position (c_{rle}^{end}) against c_{idx}^{pos}, using $right = True$ and subtracting 1. bin_e = **bucketize**($c_{rle}^{end}, c_{idx}^{pos}$, right=True) - 1 = **bucketize**($[2, 7]$, $[2, 4, 7]$, right=True) - 1 = $[1, 3]$ - 1 = $[0, 2]$. (2 is $\leq$ index 0 value 2, giving index 1; 7 is $\leq$ index 2 value 7, giving index 3. Then subtract 1).
- **Step 3: Create mask.** We create a mask where $bin_s \leq bin_e$. $mask = (bin_s \leq bin_e) = ([0, 2] \leq [0, 2]) = [True, True]$.
- **Step 4: Apply mask.** We apply $mask$ to bin_s and bin_e. They remain unchanged: $bin_s = [0, 2]$, $bin_e = [0, 2]$.
- **Step 5: Convert RLE indices to position indices.** We use `rle_to_index` to generate the flat list of indices into c_{idx}^{pos} that correspond to the ranges defined by the masked bin_s and bin_e. For range 1 ($s = 0, e = 0$): indices $[0]$. For range 2 ($s = 2, e = 2$): indices $[2]$. Combined indices: $[0, 2]$. Let this result be idx_{flat}.
- **Step 6: Gather positions.** We select the positions from c_{idx}^{pos} using idx_{flat}: $p_{out} = c_{idx}^{pos}[idx_{flat}] = c_{idx}^{pos}[[0, 2]] = [2, 7]$.

The final output is $p_{out} = [2, 7]$, which are exactly the positions from c_{idx}^{pos} that fall within the RLE ranges [0-2] or [6-7]. This matches the output of the `idx_in_rle` example for the same input.

Finally, the `idx_in_idx` algorithm (Algorithm 4) similarly to `idx_in_rle` uses `bucketize` to find potential matches between two sorted position lists (c_1^{pos}, c_2^{pos}) and then verifying exact equality.

4.3 Primitive Microbenchmarks

We ran microbenchmarks comparing CPU and GPU implementations of our 4 fundamental primitives across input sizes of 1K–100M elements on an Azure NC24ads A100 v4 VM [34]. Both implementations use PyTorch, with CPU execution via `torch.device('cpu')` on an AMD EPYC 7V13 (24 cores, 2.445 GHz) and GPU execution on an NVIDIA A100. Figure 3 shows that our GPU primitives achieve 21–46× speedups over CPU implementations at large scales (≥ 1M elements). While CPU outperforms GPU by 1.5–2.3× at small scales (1K elements) due to kernel launch overhead, GPU becomes advantageous at crossover points around 10K–100K elements, validating our design choice to optimize for large tensor query execution.

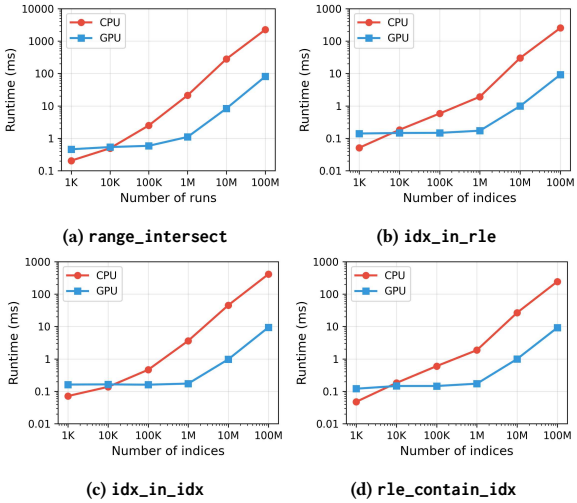

Figure 3: CPU vs GPU performance for RLE primitives. GPU achieves 21–46× speedup at scale (100M elements) with crossover points around 10K-100K elements.

5 LOGICAL OPERATIONS

Logical operators (AND, OR, NOT) operate on *MaskColumn*s as input operands and produce a *MaskColumn* as result.

5.1 AND

We summarize the design of this operator in Table 2.

- **Plain mask AND Plain mask**: Use PyTorch's logical AND operator (&). For example, $[T, F, T, F]$ AND $[T, T, F, F]$ yields $[T, F, F, F]$.
- **RLE mask AND RLE mask**: We implement this using the `range_intersect` operation (Algorithm 1).
- **RLE mask AND Plain mask**: Direct AND is inefficient due to searching each 'True' value in Plain mask. Instead, convert RLE mask to Index (`rle_to_index`) or Plain (`rle_to_plain`) then apply AND following Table 2. The choice depends on the selectivity threshold (total elements / selected elements). If this ratio exceeds the threshold, convert to an Index mask; otherwise, convert to a Plain mask. By default, we choose 20, which was determined through offline profiling on our GPU system. This threshold balances RLE compression benefits against conversion costs—providing good performance when RLE is beneficial while keeping conversion overhead acceptable.
- **RLE mask AND Index mask**: Check if Index values fall within RLE ranges using PyTorch's `bucketize`. Use `idx_in_rle` or `rle_contain_idx` depending on relative mask sizes.
- **Plain mask AND Index mask**: Select Index values where Plain positions are True using PyTorch's subscript operator ([]).
- **Index mask AND Index mask**: Check if values in one Index tensor appear in the other using `bucketize` on the larger tensor.

The output encoding for the AND operation appears in Table 3. Intuitively, the characteristics of the input layouts dictate the resulting output layout. For instance, if both inputs use RLE, their masks are typically continuous, so the output is also continuous and best encoded as RLE. However, if one input is RLE (ranges) and the other is Index (points), the resulting mask is likely discontinuous, making the Index representation more appropriate.

Alternative Design. For RLE mask AND Plain mask, our current design only converts RLE to Index/Plain representations, because it prioritizes simplicity and predictable performance. An alternative design could convert Plain to RLE instead. For instance, when a Plain mask undergoes multiple AND operations, it may become highly selective and potentially benefit from RLE compression. However, such conversion could introduce significant overhead.

To illustrate, we conducted a microbenchmark for AND between RLE mask and Plain mask, comparing RLE→Plain versus Plain→RLE conversion strategies. The setup uses 100M elements with a fixed highly-compressed RLE mask and Plain masks with varying compression ratios (1 to 10K). Figure 4 shows RLE→Plain is consistently 4.0× to 10.2× faster. Although the alternative design achieves faster AND operations with highly compressed data, conversion cost dominates: Plain→RLE conversion incurs 4.29ms overhead versus only 1.02ms for RLE→Plain conversion. This large conversion overhead negates the AND operation gains.

However, we acknowledge that, while for a single operator, converting Plain to RLE is too costly, the RLE representation may benefit downstream operations. To decide optimally though is challenging and potentially requires runtime statistics. We recognize there are interesting query optimization opportunities in this space, but leave these encoding decisions as future work.

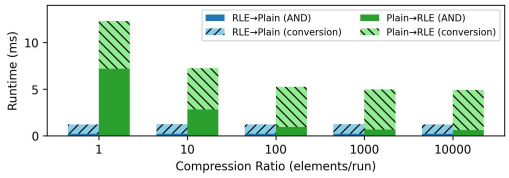

Figure 4: Performance comparison of alternative AND designs across Plain compression ratios. Alternative Design Plain → RLE has faster AND operation when Plain is highly compressible. However, the large overhead of Plain to RLE conversion negates the gain.

Table 2: Implementation Design of Physical Operator for AND between *Columns* (§ 5.1), Alignment (§ 6) and Apply Join Index (§ 8.2). Unsorted RLE and Unsorted Index are exclusively for Apply Join Index. For explicit-positional encoding (RLE or Index), the table's left section applies when m_1 and m_2 are sorted, with only the upper triangular matrix shown due to symmetry. Note: Outputs from `rle_to_index` (Index) and `rle_to_plain` (Plain) serve as input for subsequent lookups in this table.

	m_1: RLE	m_1: Plain	m_1: Index	m_1: Unsorted RLE	m_1: Unsorted Index
m_2: RLE	`range_intersect`(m_1, m_2) `range_intersect`(m_2, m_1)	`rle_to_index`(m_2) `rle_to_plain`(m_2)	`idx_in_rle`(m_1, m_2) `rle_contain_idx`(m_1, m_2)	`range_intersect`(m_1, m_2)	`idx_in_rle`(m_1, m_2)
m_2: Plain		m_1 & m_2	$m_1[m_2[m_1]]$	`rle_to_index`(m_1)	$m_1[m_2[m_1]]$
m_2: Index			`idx_in_idx`(m_1, m_2) `idx_in_idx`(m_2, m_1)	`rle_contain_idx`(m_2, m_1)	`idx_in_idx`(m_1, m_2)

Table 3: Output *MaskColumn* encoding for AND. The table is symmetric and we only show the upper triangular matrix.

	m_1: RLE	m_1: Plain	m_1: Index
m_2: RLE	RLE	Plain/Index	Index
m_2: Plain		Plain	Index
m_2: Index			Index

5.2 OR

We outline the design of this operator in Table 4, and show the encoding of the result in Table 5.

Table 4: Implementation design of Physical Operator for OR between *MaskColumns* m_1 and m_2. Since OR is commutative, the table is symmetric and we only show the upper triangular matrix for better readability.

	m_1: RLE	m_1: Plain	m_1: Index
m_2: RLE	`range_union`(m_1, m_2)	`rle_to_index`(m_2) `rle_to_plain`(m_2)	`idx_in_rle`(m_1, m_2) `rle_contain_idx`(m_1, m_2)
m_2: Plain		$m_1 \mid m_2$	$m_2[m_1] = T$
m_2: Index			`merge_sorted_idx`(m_1, m_2) `merge_sorted_idx`(m_2, m_1) `concat_sort`(m_1, m_2)

Table 5: Output *MaskColumn* encoding for OR Operation. The table is symmetric and we only show the upper triangular matrix for better readability.

	m_1: RLE	m_1: Plain	m_1: Index
m_2: RLE	RLE	Plain	RLE + Index
m_2: Plain		Plain	Plain
m_2: Index			Index

- **Plain mask OR Plain mask**: Use PyTorch's logical OR operator (|). For example, $[T, F, T, F]$ OR $[T, T, F, F]$ yields $[T, T, T, F]$.
- **RLE mask OR RLE mask**: Compute union of two sorted range lists by identifying start/end points of consecutive True values and determining non-overlapping segments.
- **RLE mask OR Plain mask**: Convert the RLE mask to Index (`rle_to_index`) or Plain (`rle_to_plain`) then apply OR operation. Similar to AND, the choice depends on the selectivity threshold with default value 20.
- **RLE Mask OR Index Mask**: Bucketize based on relative mask sizes using `idx_in_rle` or `rle_contain_idx`.
- **Plain Mask OR Index Mask**: Use PyTorch's subscript operator ([]).
- **Index Mask OR Index Mask**: Merge sorted Index tensors using `merge_sorted_idx`. Bucketize the larger tensor for efficiency, track element origins with flags, then merge using conditional selection. For small tensors, concatenate and sort.

5.3 NOT

The NOT operator takes a single *MaskColumn* as input and produces an output *MaskColumn*.

- **NOT Plain mask**: Use PyTorch's complement (~) operator for bitwise negation.
- **NOT RLE mask**: Use `complement_rle` primitive to compute gaps between consecutive runs. Each gap starts at previous run end + 1 and ends at next run start - 1. Requires metadata to track total column rows.
- **NOT Index mask**: Use `complement_index` primitive to compute gaps between indices, tracking column size. Output is in RLE format (not Index) because Index masks are sparse, making NOT results continuous and RLE-suited.

We provide further details and examples in the extended paper [22].

5.4 Operating on Composite *MaskColumns*

The previous sections discussed logical operations for non-composite *MaskColumns*. We now extend these operations to Composite masks (Plain + Index, or RLE + Index). Rather than implementing specialized operators for each composite configuration, we leverage a key insight: Composite *MaskColumns* can be conceptualized as the disjunction (OR operation) of their constituent mask tensors. This observation enables us to decompose logical operations on Composite masks into sequences of operations on their individual components, thereby reusing our existing non-composite operators. We apply Boolean algebra identities—specifically De Morgan's Laws [13], along with Associative and Distributive properties—to systematically transform these operations. Let m^{rle} and m^{idx} denote the RLE and Index components of Composite mask m, respectively.

- **NOT**: $\neg(m_1^{rle} \vee m_1^{idx}) = (\neg m_1^{rle}) \wedge (\neg m_1^{idx})$
- **OR**: $(m_1^{rle} \vee m_1^{idx}) \vee (m_2^{rle} \vee m_2^{idx}) = (m_1^{rle} \vee m_2^{rle}) \vee (m_1^{idx} \vee m_2^{idx})$
- **AND**: $(m_1^{rle} \vee m_1^{idx}) \wedge (m_2^{rle} \vee m_2^{idx})$
  $= (m_1^{rle} \wedge m_2^{rle}) \vee (m_1^{rle} \wedge m_2^{idx}) \vee (m_1^{idx} \wedge m_2^{rle}) \vee (m_1^{idx} \wedge m_2^{idx})$

Note that the sub-operations can all be performed in parallel using multiple CUDA streams. The encoding of the result *MaskColumn* is RLE for NOT operator, and Composite for OR and AND operators.

6 ARITHMETIC, COMPARISON, AND SELECTION OPERATIONS

We focus on operators that perform point-wise/position-wise operations on values from two input columns, for positions that are common to both columns. This includes the logical AND operator (Section 5.1), binary arithmetic operators, comparison operators, and selection operators with filter predicates. Joins need additional

handling (Section 8). The logical OR operator (Section 5.2) does not fit this category in the general case since the result covers a union of positions from the input columns which can be larger than the set of positions for any one column in the presence of gaps.

The main challenge in performing these operations on compressed inputs is that, unlike Plain encoding, there is no alignment between positional representations of the inputs. Thus, point-wise operations are not directly possible. We need to first align the positional representations of the inputs, including the value tensors as needed, then perform the operations on the aligned segments. We call this *Alignment*.

Example 5. *Consider $c_1 + c_2$ on input columns c_1 and c_2 both in RLE format. $c_1^{val} = [4, 1, 3], c_1^{start} = [0, 10, 20], c_1^{end} = [9, 19, 39]$; $c_2^{val} = [6, 8], c_2^{start} = [0, 15], c_2^{end} = [14, 39]$. Thus, c_1 has 3 runs (lengths 10, 10, 20) and c_2 has 2 runs (lengths 15, 25). Similar examples apply for $-, *, /, <, >, =$ operators.*

Due to run misalignment, we cannot directly do $c_1^{val} + c_2^{val}$. Misalignment can happen even with the same number of runs if run lengths differ. We first align the runs of the input columns, similar to the AND operator (Section 5.1) with an additional step of reconstructing the value tensors. After alignment, we get two RLE columns r_1 and r_2 with $r_1^{start} = r_2^{start} = [0, 10, 15, 20]$, $r_1^{end} = r_2^{end} = [9, 14, 19, 39]$, $r_1^{val} = [4, 1, 1, 3]$, $r_2^{val} = [6, 6, 8, 8]$. With identical position tensors, we can do point-wise addition: $r_1^{val} + r_2^{val} = [10, 7, 9, 11]$. Operations with scalar operands (e.g., $c * 2, 1 - c, c \geq 3$) are simple—no alignment needed, just operate on value tensors for RLE and Index encodings.

For selection operations (e.g., *SELECT C FROM D WHERE P*), we compute *MaskColumn m* from predicate *P*, align *m* and *C*, then apply *m* to the tensor representations of *C* if needed. For RLE and Index encodings, alignment performs selection (no final application needed). For Plain encodings, final mask application is required.

7 GROUP-BY-AGGREGATION OPERATIONS

For aggregation queries, the process is decomposed into two phases:

(1) **Grouping**: Partitioning rows according to the group-by columns.
(2) **Aggregating**: Computing summary statistics (e.g., SUM, COUNT, AVG, MIN, MAX) for each identified group.

Both phases are conceptually straightforward in PyTorch—function `torch.unique` can handle grouping by building inverse indices based on unique values in the group-by columns, and function `torch.scatter` can handle aggregation based on these inverse indices. However, the challenge lies in managing heterogeneous compression schemes across *DataColumn*s. For example, consider a query that groups by two columns, one RLE-compressed and one Plain. The `torch.unique` function will not work because it requires aligned Plain *DataColumn*s, but the RLE column values are not aligned with the Plain column. This heterogeneity prevents direct application of `torch.unique` and `torch.scatter` due to misalignment. We solve this by applying our Alignment technique (Section 6) to both phases.

7.1 Grouping Phase

The grouping phase constructs an inverse index that maps each row to its corresponding group; rows in the same group share the same data values across the group-by columns. It takes a set of aligned *DataColumn*s for the group-by as input and returns an inverse index. The inverse index is a single tensor containing values from the range 0 to N-1, where N is the number of unique values across the group-by columns. The inverse index length is the same as the number of values in the group-by columns—for Plain data, it is the number of rows; for RLE and Index data, it is the number of RLE runs or index points.

7.2 Aggregating Phase

The aggregation operator takes a set of aligned *DataColumn*s (for aggregation), the inverse index (from the grouping phase), and an aggregation function as input, and returns a set of tensors (one for each column to be aggregated) containing the aggregated results. The length of these tensors is the same as the number of unique values across the group-by columns.

To implement aggregation, we apply the `torch.scatter` function to the data columns, scattering based on the inverse indices. This approach efficiently computes aggregates like SUM, COUNT, AVG, STD, VAR, MIN, and MAX for each group. Note that if the data is RLE-compressed, each value must be repeated based on its corresponding RLE range size. We need to account for this when scattering the data. This depends on the aggregation function. We provide a detailed walkthrough in the extended paper [22].

- **MIN, MAX**: These functions are unaffected by compression and we only need to consider the value tensor: $min(v)$, $max(v)$.
- **SUM, COUNT**: For RLE columns, instead of expanding the data and counting/summing individual values, we compute the run lengths: $l = e - s + 1$. This is sufficient for COUNT. For SUM, we multiply the run lengths by the values ($v * l$).
- **AVG, STD, VAR**: These functions can be computed as a post-processing step after SUM and COUNT. For example, to compute AVG, we divide SUM by COUNT. To compute STD and VAR, we first compute the sum of the squared values, then use SUM and COUNT to calculate STD and VAR.

8 JOIN OPERATIONS

We reuse TQP's GPU-based hash join [18], used to join two Plain columns (from two tables) by building a hash table on one column (usually the smaller) and probing it with the other, but extend it to handle compressed columns (RLE and Index-encoded) as well. The join involves the following two steps.

(1) **Get Join Index**: Two *Join Index* tensors are computed—one for each input column—that reference the rows in each column which match according to a join predicate. These *Join Index* tensors may be unsorted and contain duplicates (in case of one-to-many and many-to-many joins).
(2) **Apply Join Index**: The *Join Index* tensors are applied to the columns participating in the join, to create the join result.

Example 6. *Consider the join operation illustrated in Figure 5 for the query "SELECT S.C FROM R, S WHERE R.A = S.B". We have two tables: Table R with column A containing values [A, B, B] at positions 0, 1, and 2. Table S with column B containing values [B, B, A] and column C containing [D, E, F] at positions 0, 1, and 2.*

The join operation executes as follows:

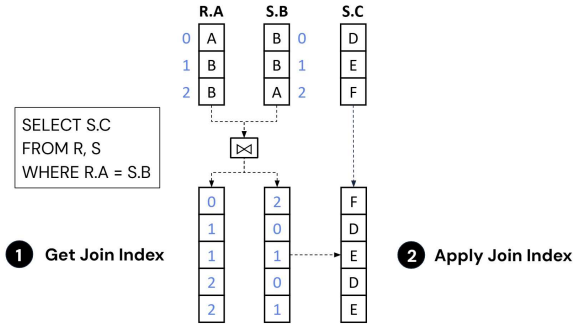

Figure 5: Illustration of join operation execution for the query "SELECT S.C FROM R, S WHERE R.A = S.B".

- **Step 1: Get Join Index.** We identify all pairs of positions where R.A equals S.B. R.A[0] = 'A' matches S.B[2] = 'A', creating the pair (0, 2). R.A[1] = 'B' matches both S.B[0] = 'B' and S.B[1] = 'B', creating pairs (1, 0) and (1, 1). Similarly, R.A[2] = 'B' also matches both S.B[0] and S.B[1], creating pairs (2, 0) and (2, 1). The resulting Join Index tensors are: Left index (for R): [0, 1, 1, 2, 2] (positions in R.A) and Right index (for S): [2, 0, 1, 0, 1] (positions in S.B).
- **Step 2: Apply Join Index.** We use the right index tensor to retrieve the corresponding values from S.C. This gives us S.C[2] = 'F' from the first match, S.C[0] = 'D' and S.C[1] = 'E' from the second and third matches (with R.A[1]), and S.C[0] = 'D' and S.C[1] = 'E' again from the fourth and fifth matches (with R.A[2]).

The final result of the join is the column [F, D, E, D, E], which contains all values from S.C that correspond to matching rows between R.A and S.B. Note that because joins can create one-to-many relationships, the same value may appear multiple times in the result, and the output size may be larger than either input table.

In the remainder of this section, we discuss how we extend the two steps to join columns encoded for compression, *without first decompressing them.*

8.1 Get Join Index

Given two *DataColumn*s, our goal is to implement a hash join on the two columns, such that the output is two *Join Index* tensors that reference the rows in each column which match according to a join predicate. Fortunately, the hashing and probing operations implementation for Plain columns can be reused directly for the value tensors of both RLE and Index *DataColumn*s. The adaptations for each compressed layout are as follows.

- **RLE Data**: We perform a hash join on the RLE column's value tensor, treating each run like a single row in the hash table. The resulting index references runs rather than individual rows. To map these run indices back to actual row indices, we re-expand by combining each run's start/end range with the index tensor.
- **Index Data**: We hash join on the column's value tensor, so the output indices initially reference the index entries themselves, not row positions. We then apply those indices to the original index tensor to recover the row positions for the final join.

The output of the hash join are two *Join Index* tensors, which are RLE or Index-encoded depending on the input *DataColumn* encodings. (Table 6).

Note that if either join column uses RLE, the matching index yields a one-to-many join and needs to be duplicated by the run length. If both join columns use RLE, then each pair of matching runs yields a many-to-many join, and the final run lengths are determined by the product of their lengths (by applying Algorithm 2 to duplicate the runs). We provide a detailed example of this join index generation process in the extended paper [22].

Table 6: *Join Index* encodings based on encodings of Input *DataColumns*. The *Join Index* tensors are unsorted and may contain duplicates. In each entry i, j of the table below, the top encoding is of the *Join Index* for the join column indicated by j and the bottom encoding is of the *Join Index* for the join column indicated by i.

	RLE Data	Plain Data	Index Data
RLE Data	RLE / RLE	Index / RLE	Index / RLE
Plain Data	RLE / Index	Index / Index	Index / Index
Index Data	RLE / Index	Index / Index	Index / Index

8.2 Apply Join Index

Once we have the *Join Indices*, we apply them to all participating *DataColumn*s to generate the joined result. This is straightforward for the join columns. For other columns, we build on the same approach we developed earlier for filtering/selecting data (see Section 6), but extend it to handle unsorted or duplicate entries.

Table 2 outlines the extension needed for Unsorted RLE and Unsorted Index *MaskColumn*s. For instance, suppose we want to apply an RLE *Join Index* to an RLE *DataColumn*. Previously, if both sides were sorted, we used Algorithm 1 to bucketize whichever side was smaller for performance. This was valid because `bucketize` requires sorted input. Now, if one side is unsorted, we must bucketize the sorted side so that each run or index from the unsorted side can be matched to the corresponding runs in the sorted data.

This step can be further optimized for one-to-many and one-to-one joins, since there are no duplicates in the *Join Index* for the many-side of the join for the former case, and no duplicates in either *Join Index* for the latter case.

9 EXPERIMENTAL SETUP AND RESULTS

We compare our tensor-based GPU query execution on compressed data against tensor-based GPU query execution on Plain data (with dictionary encoding, like TQP) [1], and CPU-only query execution with SQL Server and Analysis Services [36]. We run GPU experiments on an Azure NC24ads A100 v4 VM [34] (pay-as-you-go @$3.673/hour) having an NVIDIA A100 GPU with 80 GiB HBM, 24 vCPUs, and 220 GiB main memory. For CPU-only analytics systems, we use an Azure D64s v5 VM [33] (pay-as-you-go @$3.072/hour) with 64 vCPUs and 256 GiB main memory. We report warm query times, averaged over multiple runs, after the data is loaded into

[1]Execution plans on Plain data are an optimized version of the tensor programs generated by TQP. Plain performance are therefore equal or better than TQP. For more details on the query optimization rules applied, see the extended paper [22].

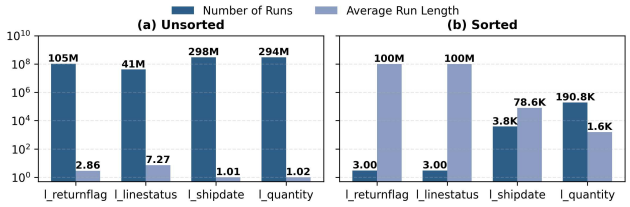

Figure 6: Total # of runs and average run lengths for fact table for TPC-H SF=50 Q1 before and after sort. Sorting greatly improves RLE compression, reducing runs from >105M to 3 while increasing average length from 2.86 to >100M.

memory (GPU HBM for GPU and main memory for CPU experiments) and caches have been warmed up. Our assumption is that data already resides in GPU memory during query execution. This is a key benefit of compression: data becomes smaller and can fit in GPU memory, avoiding expensive PCIe transfers. We monitor GPU memory usage with the nvidia-smi profiling tool.

We select input table column encodings using simple heuristics.

- Columns under 1M rows use Plain encoding.
- Else, if the RLE compression ratio > 20, then use RLE.
- Else, if many single-element runs exist but longer runs still yield an RLE compression ratio > 20, then use RLE+Index.
- Else, if removing top/bottom 5% of values allows a narrower type for the remaining data, then use Plain+Index.
- Else, use Plain encoding (possibly centered).

9.1 TPC-H Queries

We evaluate our approach using a subset of TPC-H queries (Q1, Q2, Q6, Q11, Q14, Q15, Q17, Q19) at scale factors (SF) 50, 100, and 300. The TPC-H benchmark utilizes synthetic data which, compared to real-world datasets (discussed in Section 9.2), exhibits limited sparsity and redundancy.

9.1.1 Query-Specific Data Ordering. To study the impact of data ordering on RLE compression effectiveness, we employ query-specific sort orders for the relevant tables based on the columns involved in each query's filters and joins. Each table is sorted using a global multi-column ordering (equivalent to SQL's ORDER BY clause). We show the specific column orderings used for each query in the extended paper [22]. Sorting significantly improves RLE effectiveness by increasing average run lengths and reducing the total number of runs. For instance, as illustrated for the l_returnflag column in Q1 at SF=50 (Figure 6), sorting reduces the number of runs from $> 105M$ to just 3, while increasing the average run length from 2.86 to $> 100M$. In contrast, general-purpose V-order [35] proved less effective for TPC-H even with skewness, due to high-cardinality columns, as we show in the extended paper [22].

Figure 7 presents both peak GPU memory usage and query run times, comparing query execution performance on uncompressed (Plain) data versus compressed (RLE, Index) data formats across SF 50, 100, and 300. Our evaluation reveals substantial benefits when processing compressed data directly on the GPU. Peak memory usage is significantly reduced; for example, at SF=300, Q19 requires 65.3 GiB with Plain data but only 17.5 GiB with compressed data (a 3.7× reduction). Compression enables processing larger scale factors like SF=300 for queries such as Q1 and Q17, which exceed the 80 GiB GPU memory limit with Plain data.

Query run times are also dramatically reduced, with speedups often exceeding 10×. For instance, at SF=300, Q6 runs 17.3× faster (68.6 ms vs 3.96 ms) and Q19 runs 23.8× faster (470.1 ms vs 19.8 ms) on compressed data. These results highlight the effectiveness of our techniques in improving both memory efficiency and performance for TPC-H queries on GPUs.

Using SQL Server query times for SF=50 as reference, the sum of times for the 8 queries running on GPU with compressed data was 12.8x lower for SF=50 and 2.6x lower for SF=300 (6x larger SF)! To compare with existing GPU database systems, we show HeavyDB [19] performance in Figure 8. HeavyDB outperforms uncompressed execution on Q1 and Q6, but underperforms on the rest. Overall, HeavyDB is 12.8× slower than our compressed execution (geometric mean across all queries).

9.1.2 Compression Quality Ablation Study. To study the impact of RLE compression quality on query performance, we experimented with Q17 and Q19 at SF=100. These queries demonstrate significant speedups with compression and their performance depends on the l_partkey column in LINEITEM, allowing systematic variation of compression effectiveness. We created datasets with varying compression ratios by controlling run formation. Starting with 600M LINEITEM rows naturally grouped into 20M partkeys (~ 30 rows each), we systematically break runs by further dividing each row group into 2–16 smaller runs. This creates datasets with compression ratios from 30× (20M runs) to 1.87× (320M runs).

Figure 9 shows that query performance degrades as the number of runs increases and compression effectiveness decreases. With optimal compression (30×, 20M runs), Q17 executes in 28.87ms but degrades to 189.95ms with poor compression (1.87×, 320M runs)—a 6.6× slowdown. Similarly, Q19 runtime increases from 4.11ms to 24.83ms—a 6× slowdown. These results underscore the importance of good RLE compression for high query performance.

9.2 Production queries

To study the properties of production workloads, we evaluate our methods using representative production queries on a first-party production dataset, selected by the product team as a challenging case where CPU baselines perform well. It has a star schema with a large fact table having 2.94 Billion rows and 13 small dimension tables. It also has 4 'bridge' tables that enable joins between the fact table and 4 of the dimension tables. Unlike in the TPC-H study (Section 9.1), we use the same input files, and thus the same row ordering for a given table, for all queries. We consider 2 ordering strategies: (1) On disk, each table has already been stored as V-ordered Parquet files [35], which is enabled by default in Microsoft Fabric. This is not query-specific. (2) We also experiment with a query-specific sorting strategy based on cardinality ordering, sorting all query columns by cardinality (smallest first). This corresponds to real-world scenarios where BI customers optimize specific queries using simple rules.

We use three queries for this study. Q1 uses 10 columns while Q2 and Q3 use 12 columns each. Together, the queries use 15 columns (we include detailed column usage in the extended paper [22]) which is a subset of the total columns in the fact table. Relational

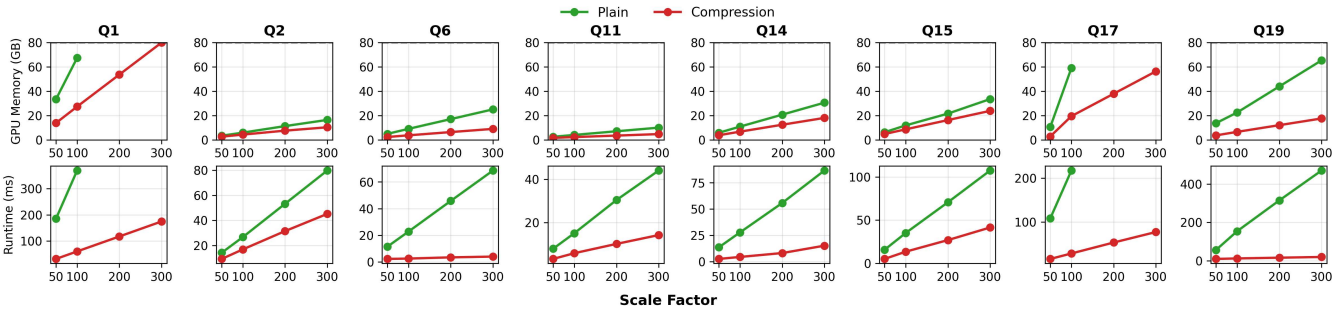

Figure 7: Peak GPU memory usage (top) and query run times (bottom) for TPC-H queries on Plain and Compressed input data across different scale factors. Compression achieves speedups up to 23.8× while reducing memory usage by up to 3.7×.

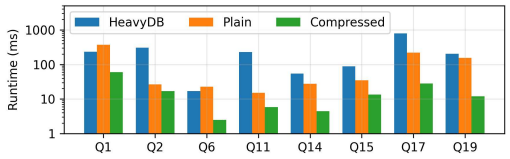

Figure 8: TPC-H query runtime between HeavyDB and compressed GPU execution at SF=100. Our compressed approach achieves 12.8× geometric mean speedup over HeavyDB.

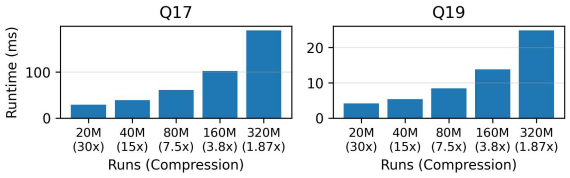

Figure 9: Query runtime degradation as compression ratio decreases for Q17 and Q19 at SF=100. Performance degrades 6×–6.6× as compression drops from 30× to 1.87×.

operations in the queries include predicate filters on dimension tables, joins and semi-joins, and group-by aggregation (SUM). Q1 includes 7 semi-joins and 2 primary-key foreign-key (PK-FK) joins, Q2 and Q3 each includes 10 semi-joins and 1 PK-FK join. Q2 and Q3 have a similar template but differ in a filter predicate.

9.2.1 Compression Effectiveness. To study compression benefits on real-world data, Figure 10 shows the sizes for the in-memory representation of the 15 columns of the fact table. The series 'Plain' refers to the default Plain encoding (with dictionary encoding) but with no other compression schemes applied.

The total size for the 15 columns is 120.36 GiB with Plain and does not fit into the 80 GiB GPU HBM. Considering only the subset of columns needed by the specific queries, Q1 needs 87.53 GiB and Q2 and Q3 need 82.06 GiB, which is still larger than the GPU HBM capacity. The bottom row of the x-axis shows the encoding scheme for Compressed—'C': Composite (Plain + Index), 'R': RLE, 'P': Plain with bit-width reduction based on the range of values. The total size for 15 columns with Compressed is 56.84 GiB. For Q1 columns it is 43.69 GiB and for Q2 and Q3 it is 30.86 GiB, which can fit in the GPU HBM and also leave space for intermediate results. For reference, the on-disk V-ordered file size for the projected fact table for the 15 columns is 29.87 GiB.

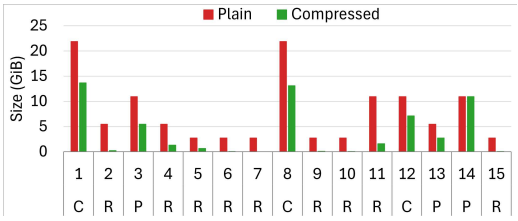

Figure 10: Input column sizes for the 15 columns in the fact table for Plain and Compressed representations. The top row in the x-axis labels shows the column numbers while the bottom shows the encoding for each column in compressed representation—R: RLE, C: Composite (Plain + Index), P: Plain with bit-width reduction.

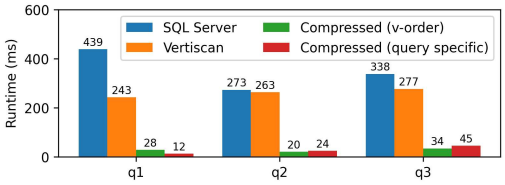

Figure 11: Query run times with Microsoft SQL Server on CPUs, Analysis Services on CPUs, and with our techniques for compressed input data on an A100 GPU.

This production dataset has substantial opportunities for RLE compression. 7 of the 15 fact table columns were compressed using RLE, achieving significant compression ratios. For example, column 7 has a single run of 2.94B rows, while column 11 achieves 6.9× compression (1.59 GiB vs 10.94 GiB for Plain representation). We provide detailed RLE statistics and compression analysis in the extended paper [22]. We also measured data transfer times from CPU to GPU using `tensor.to('cuda')` on our A100 80GB PCIe system (PCIe 4.0 x16 configuration). For compressed data, Q1 columns (45.85 GB) transfer in 2.16 seconds, while Q2 columns (26.18 GB) transfer in 1.29 seconds. These transfer times achieve 21.2 GB/s and 20.3 GB/s bandwidth respectively, representing ~64% of the PCIe 4.0 x16 theoretical maximum (~31.5 GB/s).

9.2.2 Query Performance Analysis. To study performance gains on production queries, Figure 11 shows query run times on state-of-the-art CPU-based commercial RDBMS, Microsoft SQL Server, and analytical data engine, Analysis Services. Our techniques lead to substantial speedups—15.75 (Q1), 13.66 (Q2), 9.82 (Q3), 12.76 (total)

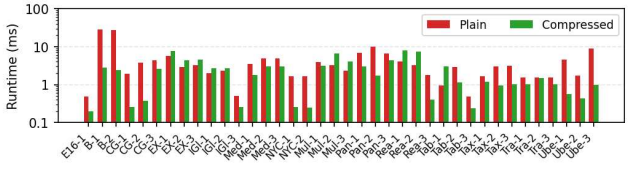

Figure 12: Query runtime for public BI datasets, comparing Plain vs Compressed data execution on GPU. Compression benefits 73.7% of queries with 2.02× geometric mean speedup and maximum speedup of 11.27×.

over SQL Server and 8.72 (Q1), 13.16 (Q2), 8.05 (Q3), 9.52 (total) over Analysis Services. The cardinality-based sorting strategy achieved run times of 12.26 ms (Q1), 24.27 ms (Q2), and 45.19 ms (Q3), comparable to V-ordering. V-order performed better for Q2 and Q3 despite being query-agnostic, while cardinality-based sorting creates highly skewed column size distributions. This demonstrates V-order's effectiveness without query-specific knowledge, while cardinality-based sorting provides a simple competitive alternative. In the extended paper [22], we present a scalability analysis showing that compressed execution enables processing datasets up to 222% (6.52B rows) of the original size within GPU memory, while Plain execution fails at 50% capacity. Compressed queries also scale better: Q2 and Q3 GPU times on Plain data at just 20% dataset size already exceed CPU times for the full 100% dataset.

9.3 Public BI Datasets

To further validate our approach on diverse real-world workloads, we evaluate our techniques using publicly available BI datasets [48]. These are collected from Tableau Public, a platform hosting over 60K user-generated BI visualizations spanning government, healthcare, transportation, e-commerce, and other domains. These datasets reflect realistic production scenarios with diverse characteristics.

Of the 34 datasets, 20 (59%) contain columns meeting the compression ratio threshold (> 20) for RLE compression. Among these 20 datasets with compression opportunities, 14 have adequate size ($\geq$ 1M rows) for meaningful GPU evaluation. We evaluate the first 3 qualified queries per dataset that utilize compressible columns, resulting in 38 total queries. Unlike our TPC-H experiments which used query-specific ordering, all datasets are sorted using V-order [35] across all columns.

Figure 12 shows compression provides significant benefits in 28 of 38 queries (73.7%) with maximum speedup of 11.27× and geometric mean speedup of 2.02×. There are also 10 queries that show slowdowns up to 3.13×. This occurs when queries have few RLE columns, and they are directly operated (e.g., AND operations) with plain columns; this introduces additional overhead from RLE-to-plain decompression. Overall, real-world BI queries demonstrate high compressibility and our execution framework shows effective acceleration for the majority of workloads.

10 RELATED WORK

GPU acceleration for SQL analytics: There has been considerable research into using GPUs to accelerate SQL analytical queries [7–9, 12, 14, 18–21, 24, 29–31, 38–42, 44, 50–55]. The majority of these works, either consider input data to be already in plain format(e.g., [7, 12, 18, 19]), or decompress it before executing queries (e.g., [42]). Several of these works also consider using GPUs to accelerate only a subset of query operators (e.g., [30, 44, 51]). Compared to this previous research, we (1) support end-to-end GPU acceleration of queries; and (2) we are the first one providing a framework allowing to run queries in GPU and directly on compressed data.

Query Execution over Compressed Data: The domain of query execution over compressed data has seen considerable research. Numerous studies have optimized dictionary encoding [11, 23, 27], among which C-store [1, 2] stands out as especially relevant. C-store introduces optimization strategies, such as rewriting summations as products over RLE and performing selections directly on the dictionary. Despite these advancements, two significant limitations exist: (1) Constricted design space: Previous works do not consider more complex scenarios where two RLE representations might select masks from each other, or where RLE representations are employed as group-by columns. Furthermore, we introduce a novel index representation, whose interaction with RLE has not previously been considered. The challenge in these studies is the amplified complexity associated with operator implementation [2, 57]. For our case, this complexity is mitigated because we utilize PyTorch-based operators. Our work presents a comprehensive exploration of these scenarios. (2) CPU-centric designs: The majority of prior work focuses on CPU operations. In contrast, our designs and operators are fine-tuned specifically for GPUs, which fundamentally differ from CPUs. A GPU's architecture emphasizes parallelism, necessitating each thread to manage simple tasks. This distinction raises a plethora of intriguing design questions.

Compression on GPU: Several works apply GPUs for query execution [6, 8, 10, 17, 18, 42] but are limited to only basic compression of dictionary encoding. With the challenges posed by limited GPU memory and PCIe bandwidth, many researchers have turned their attention to harnessing GPUs for compression [4, 15, 28, 39, 43, 45]. However, this compression typically facilitates only data transfer, and once the data is onboard the GPU, it often undergoes decompression, albeit potentially in a GPU-optimized manner. Existing works largely concentrate on refining the compression and decompression stages without addressing the direct end-to-end execution of queries in the encoded space.

RLE compression in data formats: RLE compression has wide support in both storage and in-memory data formats [3, 5, 16, 25, 49]. Reordering table rows to improve RLE compression has been leveraged in Microsoft Fabric [35], DuckDB [46], and others. We present new methods for SQL query processing to exploit the compression better for higher performance and smaller memory footprints.

11 CONCLUSION

We presented new methods for leveraging light-weight compression schemes to execute SQL analytics queries on GPUs directly on compressed data, with substantial reductions in GPU memory requirements and query run times. Our framework includes primitives for operating on encoded data, and implementations of relational operators including selection, group-by, aggregate, and join, that avoid decompression as far as possible. Our techniques result in significant acceleration of representative production queries on a real-world dataset compared to state-of-the-art commercial CPU analytics systems, and GPU query execution on uncompressed data.

REFERENCES

[1] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems.* Now Publishers Inc., Hanover, MA, USA.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[3] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *PVLDB* 16, 9 (May 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[4] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3662010.3663450

[5] Apache Arrow. (last accessed) 2025. Arrow Columnar Format: Run-End Encoded Layout. [Online] Available from: https://arrow.apache.org/docs/format/Columnar.html#run-end-encoded-layout.

[6] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *PVLDB* (2022), 3598–3601.

[7] BlazingSQL. 2021. BlazingSQL. [Online] Available from: https://github.com/BlazingDB/blazingsql.

[8] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *PVLDB* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[9] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[10] Wei Cui, Qianxi Zhang, Spyros Blanas, Jesús Camacho-Rodríguez, Brandon Haynes, Yinan Li, Ravi Ramamurthy, Peng Cheng, Rathijit Sen, and Matteo Interlandi. 2023. Query Processing on Gaming Consoles. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 86–88. https://doi.org/10.1145/3592980.3595313

[11] Patrick Damme, Annett Ungeth"um, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *arXiv preprint arXiv:2004.09350* (2020).

[12] Voltron Data. (last accessed) 2025. Theseus The Enterprise SQL Engine. [Online] Available from: https://voltrondata.com/.

[13] Augustus De Morgan. 1847. *Formal Logic: Or, The Calculus of Inference, Necessary and Probable.* Taylor and Walton, London.

[14] Yangshen Deng, Shiwen Chen, Zhaoyang Hong, and Bo Tang. 2024. How Does Software Prefetching Work on GPU Query Processing?. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/3662010.3663445

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

[16] Apache Software Foundation. (last accessed) 2025. Apache Parquet. https://parquet.apache.org/

[17] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2023. The Tensor Data Platform: Towards an AI-centric Database System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, Amsterdam. https://www.cidrdb.org/cidr2023/papers/p68-gandhi.pdf

[18] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *PVLDB* (2022), 2811–2825.

[19] HeavyDB. (last accessed) 2025. HeavyDB. [Online] Available from: https://github.com/heavyai/heavydb.

[20] Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S Bhowmick, and Wook-Shin Han. 2025. Themis: A GPU-Accelerated Relational Query Execution Engine. *Proc. VLDB Endow.* 18, 2 (Feb. 2025), 426–438. https://doi.org/10.14778/3705829.3705856

[21] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1360–1374. https://doi.org/10.1145/3514221.3517869

[22] Zezhou Huang, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. 2025. GPU Acceleration of SQL Analytics on Compressed Data. arXiv:2506.10092 [cs.DB] https://arxiv.org/abs/2506.10092

[23] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 843–856. https://doi.org/10.1145/3448016.3457283

[24] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS@VLDB*. https://api.semanticscholar.org/CorpusID:5017248

[25] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. In *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data (SIGMOD '23)*. 2205–2217.

[26] Ryan M Layer, Kevin Skadron, Gabriel Robins, Ira M Hall, and Aaron R Quinlan. 2013. Binary Interval Search: a scalable algorithm for counting interval intersections. *Bioinformatics* 29, 1 (2013), 1–7.

[27] Jae-Gil Lee, Guy Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, et al. 2014. Joins on encoded and partitioned data. *Proceedings of the VLDB Endowment* (2014).

[28] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *PVLDB* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[29] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases Beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (July 2025), 4518–4531. https://doi.org/10.14778/3749646.3749710

[30] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[31] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1795–1809. https://doi.org/10.1145/3514221.3517842

[32] Wes McKinney. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. 51–56.

[33] Microsoft. (last accessed) 2025. Dv5 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dv5-series

[34] Microsoft. (last accessed) 2025. NC_A100_v4 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nca100v4-series

[35] Microsoft. (last accessed) 2025. Understand V-Order for Microsoft Fabric Warehouse. [Online] Available from: https://learn.microsoft.com/en-us/fabric/data-warehouse/v-order.

[36] Microsoft. (last accessed) 2025. What is Analysis Services? [Online] Available from: https://learn.microsoft.com/en-us/analysis-services/analysis-services-overview?view=asallproducts-allversions.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[38] Johns Paul, Shengliang Lu, and Bingsheng He. 2021. *Database Systems on GPUs.* Now Foundations and Trends.

[39] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

[40] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[41] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (Chicago, IL, USA) *(SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[42] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. 1617–1632.

[43] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132

[44] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[45] Evangelia Sitaridi. 2016. *GPU-acceleration of in-memory data analytics*. Columbia University.

[46] Jim Stam. 2022. *Low overhead self-optimizing storage for compression in DuckDB*. Master's thesis. Universiteit van Amsterdam–Vrije Universiteit Amsterdam.

[47] Oliphant Travis E. (last accessed) 2025. NumPy. [Online] Available from: http://www.numpy.org/.

[48] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[49] Vortex. (last accessed) 2025. Vortex. [Online] Available from: https://github.com/spiraldb/vortex.

[50] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. arXiv:2506.09226 [cs.DB] https://arxiv.org/abs/2506.09226

[51] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1, Article 39 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709689

[52] Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. 2025. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17, 13 (Feb. 2025), 4709–4722. https://doi.org/10.14778/3704965.3704977

[53] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (July 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[54] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2025. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. arXiv:2502.09541 [cs.DB] https://arxiv.org/abs/2502.09541

[55] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. In *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 6. 817–828.

[56] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

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