Scaling GPU-Accelerated Databases beyond GPU Memory Size

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 Johannes Gehrke

Microsoft [‡]Cornell University [§]The Ohio State University

ABSTRACT

There has been considerable interest in leveraging GPUs' computational power and high memory bandwidth for analytical database workloads. However, their limited memory capacity remains a fundamental limitation for databases whose sizes far exceed the GPU memory size. This challenge is exacerbated by the slow PCIe data transfer speed, that creates a bottleneck in overall system performance. In this work, we introduce a hybrid CPU-GPU query processing strategy that leverages the distinct strengths of CPU and GPU to alleviate the data transfer bottleneck. Our approach performs highly efficient data filtering on the CPU, which substantially reduces the volume of data transferred to the GPU via PCIe, and offloads compute-intensive operators such as joins to the GPU for further processing. Our evaluation on the TPC-H benchmark at scale factors up to 1000 (1TB), using a single A100 GPU with 80GB memory, demonstrates that our approach can effectively handle datasets significantly larger than the GPU memory size. Moreover, it substantially outperforms a state-of-the-art CPU-only database system in both performance and cost-effectiveness.

PVLDB Reference Format:
Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, Johannes Gehrke. Scaling GPU-Accelerated Databases beyond GPU Memory Size. PVLDB, 18(11): 4518 - 4531, 2025.
doi:10.14778/3749646.3749710

1 INTRODUCTION

Graphics Processing Units (GPUs) have evolved rapidly, especially in the last decade, largely driven by the demands of machine learning applications. Given their extraordinary computational power and high-bandwidth memory (HBM), there has been a surge of interest in leveraging GPUs for building analytical database engines by caching data directly in the fast GPU memory [4, 5, 9, 13, 18, 29, 30, 33, 45, 53]. However, the limited capacity of HBM on GPUs imposes a significant impediment to making GPU accelerated analytical database systems practical. For example, NVIDIA's data center GPUs, e.g., A100 and H100, have up to 80GB of HBM [7, 8]. Many analytical databases, however, have sizes well above the HBM capacity of GPUs, often reaching hundreds of GBs or several TBs, even in a single-node setting. When the database cannot be entirely cached in GPU memory, data must be transferred on demand from the CPU to the GPU at query execution time. Unfortunately, the interconnect between CPUs and GPUs is via a slow link, such as PCIe 4.0 at 24GB/s. Compared to the bandwidth of GPU memory, e.g., 2TB/s for the NVIDIA A100, the limited data transfer bandwidth has become a major bottleneck, severely restricting the performance benefits of accelerating analytical database workloads with GPUs.

There have been several lines of prior work that address the performance bottleneck arising from data transfer, including pipelined execution between CPU and GPU [21], and techniques to alleviate the slow interconnect with caching and prefetching [32, 62]. There have also been various efforts on improving the placement of query plans and operators on CPU and GPU with cost-based scheduling [29, 41, 61, 62]. However, these techniques are not designed to handle databases that significantly exceed GPU memory capacity, e.g., by 10×, and to our knowledge, no prior work has demonstrated effective performance at such scales. Lastly, there has been work on scaling GPU database systems using multiple GPUs [5, 21, 44, 48, 56, 61], though such solutions can become very expensive for practical adoption.

In this paper, we propose techniques for cost-effective GPU acceleration of analytical database systems in a single-node setting, enabling a single GPU to efficiently process up to a few TBs of data. We focus on the single-node setting for two reasons. First, recent analysis [14] shows that over 95% of cloud data warehouse deployments manage databases smaller than 1TB, and the vast majority of real-world analytical workloads fall well within the capacity of a single node. Second, even in large-scale deployments that require distributed execution, the per-node query engine remains a fundamental building block. Our techniques can also benefit distributed systems by increasing the amount of data each individual node can process through GPU acceleration.

We address the PCIe bottleneck based on three key observations (Section 2): 1) in modern column-store database engines, scan operators on the CPU, despite processing highly compressed data, can achieve throughput close to main memory bandwidth, e.g., 80GB/s on an A100 GPU VM, which far exceeds the PCIe bandwidth; 2) in contrast, join operators on the CPU typically run significantly slower than the PCIe speed, making it beneficial to offload their execution to the GPU, despite the cost of data movement; 3) scans serve as data-reduction operators, often filtering out a large fraction of rows, thereby reducing data processed by downstream operators.

Based on these observations, we propose performing aggressive data filtering on the CPU and transferring only the filtered data to the GPU for compute-intensive operators such as joins (Section 3). CPU-side data filtering can operate at or near memory speed, which is faster than transferring the unfiltered data directly to the GPU. With access to larger main memory, the CPU is well suited for scanning large input data. The resulting filtered data is typically much

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749710

smaller in size, allowing the GPU to process only a subset of rows in downstream operators, making effective use of its high-throughput but limited-capacity memory. This design strategically leverages the distinct architectural strengths of both the CPU and the GPU to overcome the PCIe bottleneck and to reduce GPU memory usage. However, realizing efficient data filtering on the CPU presents several technical challenges, as discussed below.

First, a key distinction in our approach lies in the design of the scan operator, which deviates from the standard scan by producing filtered output in a *compressed* format. This design choice is critical for conserving PCIe bandwidth, as it enables the transfer of compressed data via PCIe. While prior work has explored how to evaluate predicates directly over compressed data [10, 35, 40, 46, 57, 58], our use case introduces an additional challenge: after filtering, the projected columns must be compacted by retaining only values from matching rows, while preserving their compressed representation. To address this, we build on recent work [39] and develop a highly efficient operator that can compact compressed values directly, without having to decompress and re-compress (Section 4). Its efficiency stems from the ability to simultaneously operate on all compressed values that fit within a word by leveraging Bit Manipulation Instructions (BMI) [34] available in modern CPUs.

Second, sometimes the data reduction on large tables in a query is only introduced by joins, e.g., when rows from a fact table are discarded after failing to join with a filtered dimension table. While we offload the execution of joins to the GPU, we propose techniques that take advantage of data reduction introduced by joins to eliminate irrelevant rows during scans on base tables (Section 5), inspired by prior work on bitvector filtering [20, 23, 42, 59, 60, 65]. Since the overhead of building and applying bitvector filters can sometimes outweigh the savings in transfer time, we further propose algorithms to strike a balance between data reduction and the cost of using bitvector filters.

We have implemented our techniques in a custom build of Microsoft SQL Server with a GPU database engine connected via PCIe. We evaluate our hybrid approach on an Azure VM with 24 CPU cores and a single 80GB A100 GPU using the TPC-H benchmark at scales up to 1TB (Section 6). To the best of our knowledge, this represents the first evaluation of the TPC-H benchmark at the 1TB scale for GPU-accelerated database systems on a single server, where a single CPU connected to a single GPU over a PCIe bus. We demonstrate that our techniques can effectively handle datasets significantly larger than the GPU memory size, successfully accelerating all 22 queries at the 1TB scale. In contrast, existing GPU database systems can execute only a limited number of queries at this scale. Overall, our approach achieves a 3.5× performance improvement over SQL Server at the 1TB scale. Our cost-performance analysis shows that our approach is highly cost-efficient: it offers a 3.4× speedup over SQL Server on a similarly priced VM, or it is 2.9× cheaper than SQL Server running on a high-end VM while delivering a 1.4× performance gain.

In summary, this work makes the following contributions:

- We introduce a hybrid CPU-GPU coprocessing strategy that applies aggressive data filtering on the CPU to alleviate the PCIe transfer bottleneck and to reduce GPU memory usage.
- We propose a scan operator that outputs compressed, filtered data, with an efficient implementation based on recent work.
- We demonstrate the effectiveness of bitvector filtering and propose an algorithm to apply it selectively for efficient filtering.
- We empirically demonstrate that our approach efficiently handles datasets an order of magnitude larger than GPU memory.

2 MOTIVATION

2.1 The PCIe Bottleneck

In general-purpose architectures, the CPU and GPU communicate through the PCI Express (PCIe) interface. However, PCIe is significantly slower than both GPU and CPU memory, making it a performance bottleneck for analytical database workloads when data needs to move between the CPU and GPU.

Table 1 compares the bandwidths of CPU and GPU memory, and PCIe across two generations of data center GPU servers. On the NVIDIA A100 VM, which uses PCIe 4.0 with a unidirectional bandwidth of 24GB/s, PCIe is 80× slower than GPU memory. More notably, its bandwidth is also 3.3× lower than the CPU memory bandwidth of 80GB/s[1]. The NVIDIA H100 VM adopts PCIe 5.0, which doubles the PCIe bandwidth compared to PCIe 4.0. However, the memory bandwidths of both the GPU and the CPU also increase accordingly. As a result, the ratio between PCIe and memory bandwidths remains nearly unchanged across the two configurations.

Prior research has shown that, given their immense compute power and memory bandwidth, GPUs can substantially accelerate query processing when the entire dataset fits in GPU memory [13, 19, 29, 50]. However, it is well understood that once the dataset exceeds GPU memory capacity, performance is often limited by the high data movement cost between the CPU and GPU, diminishing the benefits of GPU acceleration [55, 63]. The PCIe bottleneck is arguably the primary reason why, despite being an active research area, GPU-accelerated DBMSs have so far seen limited adoption and deployment in real-world systems.

Azure VM SKU	CPU memory	PCIe	GPU memory
NC24ads A100 v4	80 GB/s (24 cores)	24 GB/s (PCIe 4.0)	2 TB/s (NVIDIA A100)
NC40ads H100 v5	180 GB/s (40 cores)	48 GB/s (PCIe 5.0)	3 TB/s (NVIDIA H100)

Table 1: Bandwidth comparison: CPU vs. PCIe vs. GPU

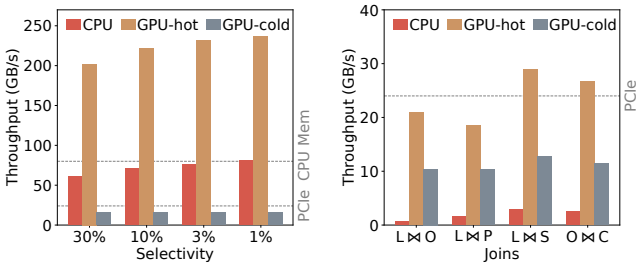
2.2 Key Observations

In this work, we address the PCIe bottleneck by leveraging the performance characteristics of key database operators and common patterns in analytical workloads. Our approach is guided by three key observations derived from a series of microbenchmark experiments conducted on an Azure NC24ads VM (see Table 1) using TPC-H at the 100GB scale. In our evaluation, we use Microsoft SQL Server [6] as the CPU engine, configured to use 24 threads, and TQP [31] as the GPU engine. While our evaluation focuses on these systems, the findings are broadly applicable to other modern analytical database engines.

Observation 1: *CPU scans outperform GPU scans when data must be transferred via PCIe.*

Figure 1a compares the throughput of scan operators under three scenarios: CPU scans, GPU scans with data pre-loaded in

[1]The server hosts four such VMs, with a total aggregate memory bandwidth of 320GB/s.



(a) Scan (b) Join

Figure 1: Operator throughput comparison: CPU vs. GPU

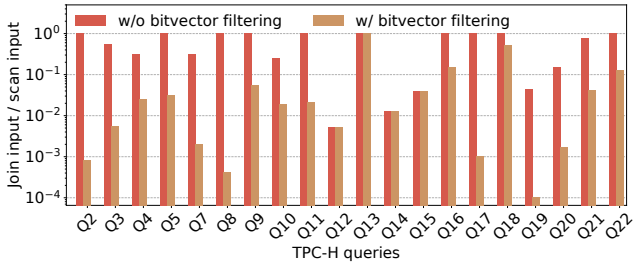


Figure 2: Operator input size comparison: scan vs. join

GPU memory (labeled as "GPU-hot"), and GPU scans where the accessed columns are transferred via PCIe (labeled as "GPU-cold"). The query applies a filter on the `l_shipdate` column and projects three additional columns: a foreign key column (`l_orderkey`), a low-cardinality column (`l_discount`), and a high-cardinality column (`l_extendedprice`). We vary the selectivity of the query from 30% to 1%. Throughput is computed based on the scan operator's execution time and the size of the compressed input columns.

Despite the input data being compressed, CPU scans can efficiently decompress and filter data at speeds approaching the CPU memory bandwidth, i.e., 80GB/s, especially when the filters are highly selective. This is largely due to the sequential access pattern, the effective use of SIMD vectorization, and the advanced techniques that enable filter evaluation on compressed or partially-compressed data [10, 35, 40, 46, 54, 57, 58]. Given the performance gap between PCIe and CPU memory bandwidths on this VM, CPU scans exceed the PCIe transfer rate by more than 3×. As a result, while GPU scans in hot runs (GPU-hot) significantly outperform CPU scans, their advantage disappears in cold runs (GPU-cold), where PCIe data transfer becomes the bottleneck and CPU scans outperform GPU scans.

Observation 2: *GPU joins are consistently faster than CPU joins, regardless of whether the data is transferred over PCIe or not.*

Figure 1b shows the throughput of four different joins on both the CPU and the GPU: `lineitem` ⋈ `orders` (L ⋈ O), `lineitem` ⋈ `part` (L ⋈ P), `lineitem` ⋈ `supplier` (L ⋈ S), and `orders` ⋈ `customer` (O ⋈ C). The throughput is calculated based on the compressed size of the larger input table. In contrast to scans, CPU joins are significantly slower than the PCIe transfer speed, primarily due to the inherently random memory access patterns of join operators. Consequently, not only do GPU joins outperform CPU joins when data is pre-loaded (hot runs), but they also remain substantially faster even when data must be transferred over PCIe (cold runs).

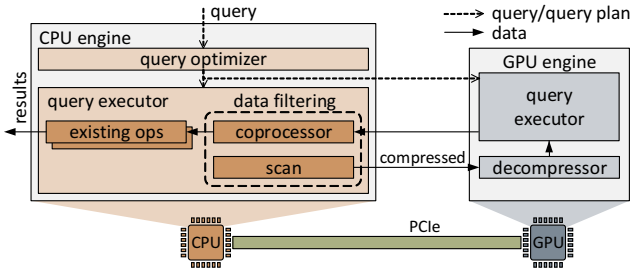


Figure 3: System architecture

Observation 3: *Joins often process far fewer rows than scans, sometimes by orders of magnitude, especially when using bitvector filtering.*

Figure 2 shows the ratio of rows processed by the scan and join operators on the largest table of each query in the TPC-H benchmark, executed on SQL Server. Queries Q1 and Q6 are excluded from this figure as they are scan-only queries. When bitvector filtering is disabled, joins process fewer than 10% of the scanned rows in only 4 out of the 20 queries. In contrast, when bitvector filtering is enabled and unmatched rows are eliminated prior to joins, the number of rows processed by joins dramatically decreases. Compared to scans, for approximately half of the queries, joins process less than 1% of rows, and only Q13 and Q18 process more than 15%. These results indicate that only a small fraction of the input data needs to be processed by downstream operators, including joins. Furthermore, a recent study of a large number of real-world workloads shows that real-world queries tend to be even more selective than those in TPC-H [66], further supporting this observation.

3 SYSTEM OVERVIEW

This section presents an overview of the hybrid CPU-GPU coprocessing solution. Building on the three observations described in Section 2.2, we propose a simple yet effective strategy to address the PCIe bottleneck in GPU-accelerated DBMSs. First, scans are executed on the CPU, which has been shown to be more efficient than transferring all data directly to the GPU (Observation 1). Since scans often reduce data volume by more than an order of magnitude (Observation 3), this approach significantly lowers the amount of data transferred over PCIe, thereby mitigating the PCIe bottleneck. The filtered data is then sent to the GPU, where compute-intensive operators such as joins and aggregates are executed, taking advantage of the GPU's superior performance for these operators (Observation 2). For analytical workloads, the final query results are typically small and can be transferred back to the CPU with minimal overhead. Overall, this strategy effectively leverages the complementary architectural strengths of the CPU and GPU to optimize overall performance.

3.1 System Architecture

Figure 3 shows the high-level system architecture of our hybrid approach. It integrates a host database engine on the CPU with an execution engine on the GPU that is connected via the PCIe bus. Data is stored in compressed columnar format in the host database. Upon the arrival of a query, the query optimizer generates a query plan that may include one or more subplans offloaded to the GPU engine. Each offloaded subplan is rooted at a *coprocessor operator*, which orchestrates data access and data movement between the CPU and GPU. Based on the subplan, the required columns are read

by scan operators and transferred to the GPU over PCIe. The GPU engine decompresses the received data, executes the subplan, and returns results back to the coprocessor operator. These results are directly emitted as the output of the coprocessor operator and are consumed by the downstream operators that were not offloaded to the GPU. The host database engine then executes these operators on the CPU and returns the final query result to the user.

Based on observations in Section 2.2, we perform aggressive data filtering in the host database engine on the CPU. Instead of transferring entire columns, our hybrid approach filters out irrelevant data and transfers only filtered data to the GPU execution engine. Importantly, the filtered data retains the same compressed format as the original columns, which makes filtering on the CPU transparent to the GPU: the GPU engine does not need to distinguish between original and filtered data.

3.2 Overview of Data Filtering

In this subsection, we dive into the data filtering process, highlighted in the dashed box in Figure 3, and present an overview of its workflow and the key techniques it employs. For the offloaded subplan, the coprocessor operator is connected to a set of scan operators, one for each table, responsible for performing all data filtering. These scan operators employ two key techniques: *predicate filtering* and *bitvector filtering*. We use an example query, Q1, based on the TPC-H schema, to illustrate this process. Figure 4 shows the operator tree of Q1's query plan on the CPU side, which is rooted at a coprocessor operator and includes two scan operators.

```
Q1:  SELECT SUM(l_extendedprice) FROM lineitem, part
     WHERE p_partkey = l_partkey AND p_size < 10.
```

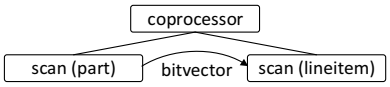


Figure 4: Operator tree for data filtering in Q1 (CPU-side)

Predicate filtering (Section 4). With predicate filtering, the scan operator evaluates predicates and discards rows that do not satisfy them. While this resembles the behavior of a standard scan operator in CPU-only execution, the key difference lies in the output format: the filtered data remains in compressed form, identical to the original columns. Transferring compressed data over PCIe is a well-known technique for reducing data movement in GPU database engines [12, 25, 51, 63]. Our approach retains the benefits of compression even after filtering, resulting in data that is both filtered and compressed before being transferred to the GPU, thereby further mitigating the PCIe bottleneck. For example, in Q1, applying the filter `p_size < 10` (20% selectivity) reduces the PCIe transfer for the `part` table by 5×, by retaining only the matching 20% of rows while preserving the compression format and ratio.

Bitvector filtering (Section 5). When a query lacks selective predicates on larger tables, predicate filtering alone is insufficient for effective data reduction. For instance, in Q1, the `lineitem` table is substantially larger than the `part` table and lacks direct filter predicates. As a result, predicate filtering fails to reduce its size, providing limited benefit to overall performance. To address this, we adapt bitvector filtering [20, 23, 42, 65], a well-known technique for improving join efficiency. The core idea is to propagate the filter from the smaller (build-side) table across the join to the larger (probe-side) table by constructing a compact data structure, typically a simple bitmap or Bloom filter [17], that encodes the join keys of relevant rows on the build side. This bitvector is then used to prefilter rows on the probe side during the scan, discarding those that cannot match any join key. As previously shown in Figure 2, bitvector filtering is highly effective in further reducing data volume for many queries. In the context of the hybrid approach, bitvector filters are both constructed and applied within the scan operator running on the CPU, before data is transferred over PCIe for join processing. This early filtering significantly reduces the number of rows produced by scans, thereby lowering PCIe traffic, especially when joins are selective. For example, in Q1, as shown in Figure 4, after performing predicate filtering on `part`, we build a bitvector over `p_partkey` and use it to filter `lineitem` on `l_partkey`, resulting in a 5× reduction in data transfer from `lineitem`. Bitvector filters are applied using the same predicate filtering technique described above, by treating them as additional filtering predicates.

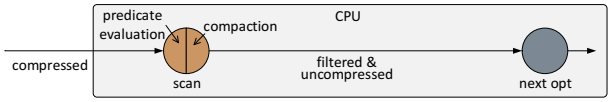
Key challenge. The hybrid approach is only effective if data filtering on the CPU is faster than the PCIe transfer rate; otherwise sending compressed but unfiltered data to the GPU is preferable. While Figure 1a shows that standard scans on the CPU can exceed PCIe throughput, our scan operator performs additional tasks such as bitvector filtering and preserving compression. The key challenge lies in ensuring that the scan operator remains faster than the PCIe transfer rate despite these extra operations. Sections 4 and 5 describe our techniques that address this challenge.

3.3 Streaming and Partitioning

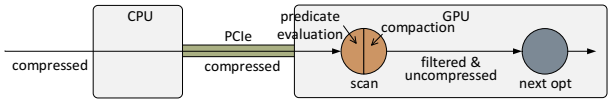
In addition to data filtering, we use two techniques, streaming and partitioning, to further reduce GPU memory usage and to enable processing of (filtered) tables that exceed GPU memory size. However, unlike data filtering described in Section 3.2, these techniques do not reduce PCIe transfer volume and therefore do not alleviate the PCIe bottleneck.

Streaming. Streaming enables columns to be transferred and processed in chunks rather than all at once. In streaming execution, an operator can proceed as long as its internal state fits within GPU memory. For example, in a hash join, the hash table built from the build table is kept in GPU memory, while the probe table is streamed in chunks for probing. This approach requires the GPU to store only the hash table and one chunk of the probe table at a time, thereby significantly reducing memory usage. This technique has been adopted in prior work to reduce GPU memory usage [4, 27, 52].

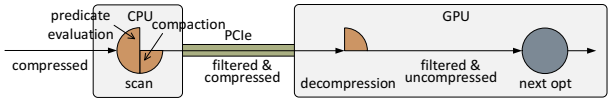
Partitioning. The partitioning technique described below is inspired by the classic repartitioning method [49] used in distributed database systems. With repartitioning, both sides of a join are partitioned using a common scheme, such as range or hash partitioning, and corresponding partition pairs are shuffled and joined in parallel. In contrast, our approach processes partition pairs sequentially, ensuring that each pair fits entirely in GPU memory. Partitioning is achieved by repeatedly scanning the input tables with partition-specific predicates to extract one partition per scan. For example, in Q1, we partition the input tables into two ranges using filters such as `p_partkey BETWEEN x AND y` and `l_partkey BETWEEN x AND y`, requiring two scans per table. This approach leverages the



(a) Scan for CPU-only execution



(b) Scan for GPU-only execution



(c) Scan for hybrid CPU-GPU execution

Figure 5: Comparison of scan operators

predicate filtering technique (Section 4) to perform fast scans and is particularly efficient when the number of partitions is small. Hash partitioning could also be used to mitigate data skew, although it would introduce additional overhead due to hash computations. Overall, this design allows us to process one partition pair at a time on the GPU, further reducing GPU memory usage.

We will demonstrate how these techniques, combined with data filtering, effectively reduce memory usage in Section 6.4.

4 PREDICATE FILTERING

This section presents the design of our scan operator for predicate filtering. We first review the standard scan operator (Section 4.1), then introduce our customized design for the hybrid approach (Section 4.2 and 4.3), followed by additional optimizations (Section 4.4).

4.1 Background on Scan

In modern column-store analytical database systems, a standard scan operator performs two functions, predicate evaluation and compaction, as shown in Figure 5a. Because the downstream operators often cannot directly process compressed data, a standard scan operator outputs filtered values in an *uncompressed* format. The typical workflow of a scan operator is the following:

- **Predicate evaluation.** During this step, the operator evaluates the predicates either directly on compressed values or on values after decompression, and produces a selection bitmap to indicate matching rows. This step has been extensively optimized in prior work through various techniques [10, 35, 39, 40, 46, 57, 58, 64].
- **Compaction.** Using the selection bitmap from the previous step, the next step compacts the projected columns, i.e., the subset of columns needed by subsequent operators, by discarding values from unselected rows. The projected columns are first decompressed and then compacted by removing unselected values, with both steps optimized using SIMD vectorization [46, 58].

Prior work has shown that GPUs can efficiently perform decompression, and thus advocates transferring compressed columns to GPUs to reduce PCIe overhead [12, 25, 51, 63]. In this design, a GPU scan mirrors its CPU counterpart by executing both predicate evaluation and compaction directly on the GPU, as shown in Figure 5b.

4.2 Scan for Hybrid Coprocessing

We propose a specialized scan operator for the hybrid approach, as illustrated in Figure 5c. While it performs predicate evaluation on the CPU similarly to a traditional CPU-only scan, it differs in how it compacts data. Instead of outputting uncompressed values, it produces filtered data in *compressed* form. This combination of early filtering and compression is essential to mitigating the PCIe bottleneck.

The key challenge is *how to design a compaction operator that can compact and output compressed values efficiently?*

In the standard scan operator, as described in Section 4.1, compressed data cannot be directly processed by the compaction step and must be partially or fully decompressed first. To produce compressed output, a naïve solution is to apply a separate compression step after compaction. However, this incurs significant CPU overhead and can substantially offset the PCIe transfer savings achieved through sending compressed data.

To address this challenge, we develop a compaction operator that directly compacts compressed values, avoiding both decompression and re-compression on the CPU. Unlike the naïve approach, which introduces additional overhead, this method compacts compressed values directly on the CPU and offloads decompression of projected columns to the GPU (as shown in Figure 5c), often resulting in better performance than the standard scan operator. The details of this technique are presented in the following subsection.

4.3 Direct Compaction on Compressed Values

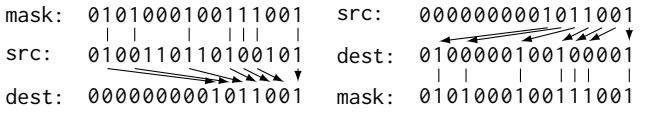
Our compaction operator takes a byte array containing n compressed values and an n-bit selection bitmap as input and outputs a byte array that includes only the values selected by the bitmap in their original *compressed* format. Below, we describe how to implement this operator for three encoding schemes: bit-packing, run-length encoding (RLE), and dictionary encoding.

4.3.1 Bit-packing. Bit-packing encoding uses only as many bits as needed to represent each value, often resulting in values that do not align with byte or word boundaries. This misalignment poses challenges for efficiently compacting these bit-packed values. The example below shows the input and expected output of the compaction operator (values are shown in binary representation; we alternate background colors to differentiate bits from different values). According to the selection bitmap, three values (v1, v5, and v6) are extracted from a sequence of eight 4-bit values and written to the output in their original compressed form.

```
input                    v7  v6  v5  v4  v3  v2  v1  v0
  bit-packed values:  10000001110100111001010111010010
   selection bitmap:     0   1   1   0   0   0   1   0
output                                      v6  v5  v1
                                        000111011101
```

Given that each value occupies only 4 bits, the goal is to process all values that fit within a 64-bit processor word simultaneously, rather than iterating over and processing each value individually. Achieving this with conventional CPU instructions is challenging, if not impossible, due to the lack of fine-grained bit-level manipulation capabilities [37]. Recent work [39] addresses this challenge using two special instructions, PEXT and PDEP [34], from the Bit

Manipulation Instructions (BMI) set, an x86 extension supported by both Intel and AMD processors[2]. These instructions have been available in server processors for several years and are now widely supported in today's CPUs. Originally, this BMI-based compaction operator was proposed to optimize the scan operator in CPU-only execution [39]. In this paper, we extend its use to a new context: the scan operator in hybrid execution, where compacted output in compressed form is required. This setting aligns naturally with the operator's functionality. Below, we briefly review the technique.



(a) PEXT (parallel bit extract) **(b) PDEP (parallel bit deposit)**

Figure 6: Examples of BMI instructions

PEXT (Parallel Bit Extract) and PDEP (Parallel Bit Deposit) are 64-bit instructions designed to perform bit-level gather and scatter operations, respectively. Figure 6 shows examples of the two instructions on 16-bit operands. PEXT selectively extracts bits from a source operand based on a selection mask and copies them into the continuous low-order bits of the destination operand. PDEP performs the opposite operation: it places the contiguous low-order bits from the source operand into positions in the destination operand specified by the selection mask.

Figure 7 demonstrates the use of PEXT and PDEP to compact the bit-packed values in the example above. The process has two steps. In the first step, the selection bitmap is transformed into an extended bitmap by replicating each bit four times to match the bit-width of the values. For a bit $b \in \{0, 1\}$, the replicated pattern bbbb can be computed as `b0000 - 0000b`. This operation is applied in parallel across all value lanes within a word, using two PDEP instructions to place each bit into the correct positions for generating `b0000` and `0000b`, followed by a subtraction to produce the extended bitmap. In the second step, with the extended bitmap, we can now simply use PEXT to extract all the bits of the selected values, thus producing the compacted bit-packed values.

The BMI-based compaction operator simultaneously processes all values that fit into a 64-bit word by using only five instructions, regardless of the bit width (e.g., 16 4-bit values, or $10\frac{2}{3}$ 6-bit values). This data parallelism enables highly efficient compaction that directly produces bit-packed values.

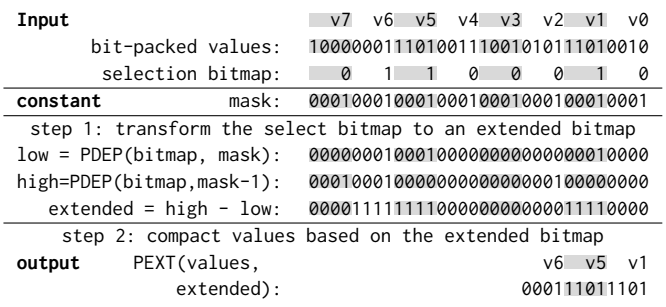


Figure 7: BMI-based compaction on eight 4-bit values

[2]Arm supports similar functionality via vectorized instructions BDEP and BEXT [2].

4.3.2 RLE. An RLE run is represented by a value v and its repetition count k. To compact an RLE run, we count the number of 1s within the next k bits of the selection bitmap, corresponding to the occurrences of v. This can be implemented using the POPCNT instruction, which is available in both x86 and Arm architectures. The compacted RLE is then represented by the value v and the new count, which reflects how many times v is selected according to the selection bitmap. In a column containing consecutive RLE runs, we process each run individually, compacting them one at a time.

4.3.3 Dictionary encoding. With dictionary encoding, all unique column values are stored in a dictionary, and the data column contains only their corresponding dictionary indexes. These indexes can be further compressed using either bit-packing or RLE. For GPU execution, both the index column and the dictionary must be transferred over PCIe. During compaction, the index column is compacted using the same methods described earlier for bit-packing or RLE. When the dictionary is significantly larger than the index column, we further reduce the transfer cost through dictionary compaction, which eliminates entries that are not referenced by any selected rows. This is achieved without modifying the index column: unused entries are replaced with empty values, preserving their original positions and dictionary indexes. This method effectively reduces the dictionary size—and consequently the PCIe transfer traffic—without incurring additional CPU overhead.

4.3.4 Putting them together. Columns often interleave bit-packing and RLE runs, so regions of repeated values are encoded with RLE while other regions use bit-packing. When compacting such columns, we apply the appropriate method to compact each individual run according to its specific encoding. Any empty runs resulting from this compaction are removed. In addition, we also merge consecutive runs using the same encoding after removing the empty runs. These optimizations can reduce the number of runs, potentially improving subsequent GPU decompression by lowering the overhead associated with managing multiple runs.

4.4 Skipping Filters

Despite the highly efficient scan operator, skipping certain filters on the CPU can sometimes be beneficial when the cost of evaluating them outweighs the potential gains. For example, filters that are either insufficiently selective or computationally expensive (e.g., string matching or UDFs) may slow CPU execution enough to negate the expected transfer savings. In such cases, it is more efficient to bypass CPU-side filtering and instead offload the filter to the GPU. If all the predicate filters are skipped, the system simply transfers the original, unfiltered compressed columns to the GPU. Section 5.2.2 describes our cost-based algorithm that jointly selects both predicate and bitvector filters.

5 BITVECTOR FILTERING

In this section, we present the bitvector filtering technique tailored for hybrid query processing. We begin by outlining the performance requirements of our use case in Section 5.1, then describe how bitvector filters are selected in Section 5.2, and finally discuss the design and implementation of bitvector filters in Section 5.3.

5.1 Bitvector Filtering for Hybrid Coprocessing

Bitvector filtering is a well-known technique for reducing the number of rows processed by join operators on the CPU. In this setting, since join operators are computationally expensive, bitvector filtering is designed to maximize data reduction, i.e., to eliminate as many non-matching rows as possible. This goal often leads to two design choices: 1) bitvector filters are applied aggressively, even when they yield only modest data reduction, and 2) the filter structure is optimized to minimize false positive rates.

However, in the hybrid approach, bitvector filtering is used not to reduce join input size, as in traditional use cases, but to reduce data transferred over PCIe. This technique is effective only when the CPU cost of constructing and probing the bitvector is outweighed by the savings in transfer cost. As previously shown in Figure 1b, PCIe bandwidth (the dashed line in Figure 1b) typically exceeds CPU join throughput by an order of magnitude. This performance gap imposes significantly stricter performance requirements on bitvector filtering in our setting, compared to conventional use cases. To meet these requirements, filters must be applied *selectively*—when the expected reduction in data transfer is marginal, the added cost may outweigh the benefit. In addition, bitvectors must be carefully tuned to strike a balance between false positive rate and probing performance. Our designs for these two aspects are presented in Sections 5.2 and 5.3, respectively.

5.2 Selecting Bitvector Filters

For a given query, we construct a bitvector filtering plan by first identifying all the candidate bitvector filters of a query plan and then selecting only those that offer substantial data reduction.

5.2.1 Deriving candidate bitvector filters. Given a query plan generated by the query optimizer, we derive the candidate bitvector filters by finding all the bitvector filters that can be constructed from the plan by adapting the existing approaches [20, 23, 42, 65]. Intuitively, bitvector filters can be constructed based on equi-join conditions. In addition, they are required to be created from and probed by columns from the base tables (i.e., no intermediate results) in hybrid coprocessing. We derive these candidate bitvector filters by tracing the lineage of the columns of the join conditions in the query plan. We recursively traverse the query plan from top to bottom, extract the columns from equi-join conditions, and trace the create-from and probe-by columns of a bitvector filter to the corresponding columns from the base tables. If a join condition involves equi-join of more than one column, we derive a candidate bitvector filter for each pair of join columns.

Figure 8a shows the join graph for TPC-H Q5, and the corresponding join plan with bitvector filtering is shown in Figure 8b. To derive the candidate bitvector filters, we start from the root join $Join_1$ with the join condition `s_nationkey=c_nationkey AND s_suppkey=l_suppkey`. Since there are two columns in the join condition, we derive one candidate bitvector filter per column, i.e., $\mathcal{F}_1 : s\_nationkey \rightarrow c\_nationkey$ and $\mathcal{F}_2 : s\_suppkey \rightarrow l\_suppkey$. Then we traverse to the left child of $Join_1$, i.e., $Join_2$ with the join condition `n_nationkey=s_nationkey`, and we derive a candidate bitvector filter $\mathcal{F}_3 : n\_nationkey \rightarrow s\_nationkey$. Similarly, we derive the candidate bitvector filter $\mathcal{F}_4 : r\_regionkey \rightarrow n\_regionkey$ with $Join_3$, and we finally traverse to the base table `region`. Here, we map the column `r_regionkey` that creates $\mathcal{F}_4$ to `region` table. Now we backtrack to `nation` table, where we map both column `n_regionkey` that probes $\mathcal{F}_4$ and column `n_nationkey` that creates $\mathcal{F}_3$ to `nation` table. After traversing the plan shown in Figure 8b, we derive two additional bitvector filters: $\mathcal{F}_5 : c\_custkey \rightarrow o\_custkey$ and $\mathcal{F}_6 : o\_orderkey \rightarrow l\_orderkey$. We end up with six candidate bitvector filters as shown in Figure 8c.

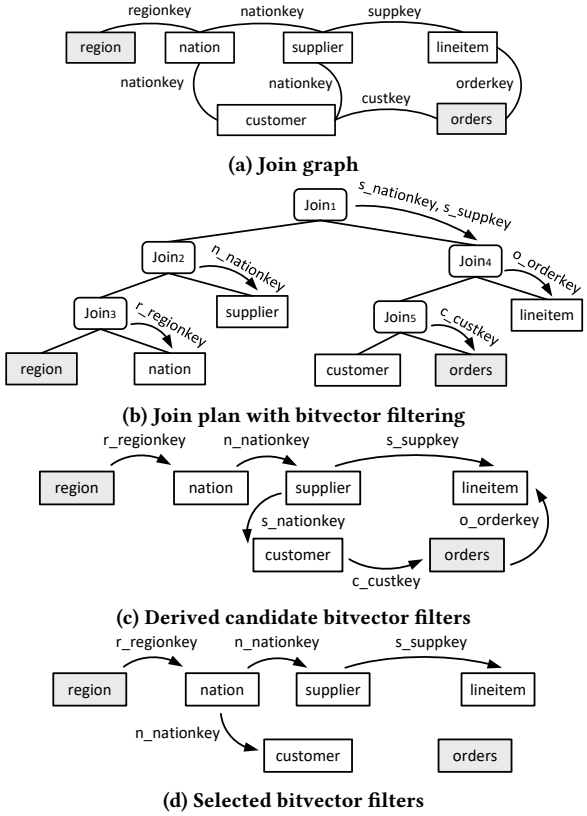


Figure 8: Example bitvector filtering plan derived from the join plan of TPC-H Q5. Arrows indicate where bitvector filters are created and probed, annotated with the column to create the filter. Tables with predicate filters are colored gray.

5.2.2 Selecting candidate filters. Given a set of candidate bitvector filters, it is not always beneficial to apply all the filters for data filtering in hybrid coprocessing. Instead, we select both predicate filters (as discussed in Section 4.4) and bitvector filters jointly, taking into account their combined effect on data reduction. When multiple filters apply to the same table, the marginal benefit of each additional filter may diminish as the joint selectivity decreases. Therefore, the filter selection process must be cost-based, balancing the overhead of applying filters on the CPU against the expected reduction in PCIe transfer.

We propose a greedy algorithm to holistically select a subset of predicate filters and bitvector filters in a cost-based manner as shown in Algorithm 1. We first remove equivalent or redundant bitvector filters based on their dependency and selectivity

Algorithm 1 Cost-based filter selection using a greedy algorithm

```
1:  function GetDataFilterPlan(plan, filters, threshold)
2:      filters ← RemoveRedundantBitvectorFilters(filters)
3:      tables ← GetTablesDfs(plan)
4:      result ← ∅
5:      for t ∈ tables do
6:          bfs ← GetBitvectorFiltersToProbe(t, filters)
7:          pfs ← GetPredicateFilters(t)
8:          tblFilters ← bfs ∪ pfs
9:          sortedFilters ← SortByEstimatedBenefitDesc(tblFilters)
10:         selectedFilters ← ∅
11:         for f ∈ sortedFilters do
12:             selectedFilters.Add(f)
13:             s ← GetSelectivity(tables, t, result ∪ selectedFilters)
14:             if s < threshold then
15:                 result ← result ∪ selectedFilters
16:                 break
17:     return DeriveBitvectorFilteringPlan(tables, result)
```

(line 2). For the example of TPC-H Q5 shown in Figure 8c, because there is no predicate filter on `supplier` table and all values in `n_nationkey` appear in `supplier` table, the bitvector filter $\mathcal{F}_1$ created from `s_nationkey` is equivalent to $\mathcal{F}_3$ created from `n_nationkey`. Thus, $\mathcal{F}_1$ can be replaced by $\mathcal{F}_3$ and removed. Next, we decide which filters to apply on each table in their traversal order (line 3), i.e., respecting the dependency of the bitvector filter creation and application. For each table, we sort the filters based on their estimated cost benefit in descending order (line 6-9) and then select a subset of the filters until the selectivity on the table drops below a threshold (line 10-16). For example, while the bitvector filter $\mathcal{F}_5 : c\_custkey \rightarrow o\_orderkey$ can be applied to the `orders` table, because the predicate filter on `orders` is already selective enough with 15% selectivity, $\mathcal{F}_5$ is not selected for the plan. Note that the estimated cost benefit of a filter considers both the overhead of using the filter and its reduction of data transfer cost (line 9). If the filter is a predicate filter, we estimate the cost of evaluating the predicate and reduction on data transfer cost; if the filter is a bitvector filter, we estimate the cost of creating and probing the bitvector filter and reduction on data transfer cost. For example, although the bitvector filter $\mathcal{F}_2 : s\_suppkey \rightarrow l\_suppkey$ is less selective than $\mathcal{F}_6 : o\_orderkey \rightarrow l\_orderkey$ on `lineitem` table, creating $\mathcal{F}_2$ from `supplier` table has much lower overhead and thus its estimated cost benefit is higher. The resulting bitvector filtering plan of TPC-H Q5 is shown in Figure 8d.

Note that the cost benefit estimation neglects the correlations between filters applied to the same table, i.e., assuming independence. Additionally, the greedy algorithm does not account for the cascading effect, i.e., how the selectivity of the current table impacts the effectiveness of bitvector filters created from it. However, we observe empirically that the resulting bitvector filtering plan is often good enough for hybrid coprocessing.

5.3 Bitvector Filter Design and Implementation

Various exact and approximate data structures [17, 23, 24] have been proposed to perform bitvector filtering, including simple bitmaps (i.e., one bit per value for values in the domain), hash-based bitvectors, and Bloom filters [17]. In the context of hybrid coprocessing, where performance requirements are stringent, throughput is often prioritized over false positive rate in bitvector filtering. For domains of relatively small cardinality, a simple bitmap is often preferred due to its lookup performance. In contrast, for larger domains where a simple bitmap exceeds CPU cache size, a hash-based bitvector with a lightweight hash function is used. The size of the bitvector is also carefully configured to favor lookup throughput, even at the expense of a higher false positive rate. In particular, smaller bitvectors may be chosen to ensure CPU cache efficiency, trading off accuracy for faster accesses.

In our implementation, the scan operator is responsible for both constructing and applying bitvector filters. In particular, probing a bitvector is treated as an additional predicate and evaluated in the scan operator described in Section 4. To improve performance, bitvector probing is further optimized using SIMD vectorization.

6 EVALUATION

We implemented a prototype system by integrating the proposed techniques into the query execution engine of a custom build of Microsoft SQL Server [6], coupled with the TQP GPU database engine [31]. We evaluate this prototype using the industry-standard TPC-H benchmark to answer the following questions:

- **Efficiency**: What is the performance of the hybrid approach compared to CPU-only and GPU-only approaches, and to what extent does it mitigate the PCIe bottleneck? (Sections 6.2 and 6.3)
- **Scalability**: How well does our approach scale compared to existing approaches in both performance and workload coverage when the dataset exceeds GPU memory capacity? (Section 6.4)
- **Cost-effectiveness**: How does our approach compare to the CPU-only approach in terms of perf/$? (Sections 6.5)

6.1 Setup

Our experiments are conducted on an Azure NC24ads A100 V4 VM, equipped with 24 physical cores, 220GB of main memory, and an 80GB NVIDIA A100 GPU connected through PCIe 4.0. The bandwidths for the main memory, GPU memory, and PCIe are listed in Table 1 (see Section 2). The physical server hosts four such VMs and has two AMD EPYC 7V13 (Milan) processors, 880GB of main memory, and four A100 GPUs.

We run all 22 queries from the TPC-H benchmark [1] at three scale factors: 100GB, 300GB, and 1TB. The 100GB dataset represents the setting where the working set of the workload fits entirely in GPU memory. In contrast, the 1TB dataset exceeds the GPU memory size by an order of magnitude, making data transfers over PCIe necessary during query execution (see Section 2.1).

Each query is executed with warm main memory caches, i.e., the data accessed by the query is loaded from CPU memory. We report the average execution time over 10 warm runs. The CPU database engine uses 24 threads, matching the number of cores in the VM. For the GPU-only baselines, we evaluate the queries in both hot and cold runs. In hot runs, data is preloaded into GPU memory, requiring that the entire dataset fits within GPU memory. In cold runs, all required data is transferred on demand from the CPU to the GPU via PCIe.

In our experiments, we always offload the full query plan to the GPU (while our hybrid approach, by design, still performs

data filtering on the CPU). When datasets exceed GPU memory capacity, queries may trigger out-of-memory (OOM) exceptions. Our prototype handles such cases by catching the exceptions and falling back to CPU execution. However, for evaluation purposes, we expose the OOM exceptions encountered by each approach.

We evaluate the following approaches in our experiments:

- **Hybrid**. Our prototype integrates the proposed techniques into Microsoft SQL Server as the CPU database engine and uses TQP as the GPU database engine. Our current implementation uses synchronous PCIe transfers, which prevents the overlap of CPU execution and PCIe transfers within a single thread but still allows inter-thread overlap.
- **Microsoft SQL Server (*SQL*)**. We use Microsoft SQL Server [6] as the CPU-only baseline, and create column-store indexes on all the tables, as is expected for an analytical database.
- **TQP**. We use the TQP GPU database engine [31] as a GPU-only baseline. TQP is integrated with SQL Server following the architecture shown in Figure 3. Data is either pre-loaded to the GPU or transferred on demand (but without data filtering).
- **HeavyDB**. HeavyDB [5] is a leading open-source database engine built for GPUs and serves as an additional GPU-only baseline. We performed best-effort tuning to enhance its performance. Additionally, since the evaluation focuses on query execution, we also tried to alleviate any performance inefficiencies that result from suboptimal plans by unnesting some queries (Q2, Q4, Q16-Q18, Q20, and Q22) with manual rewriting.

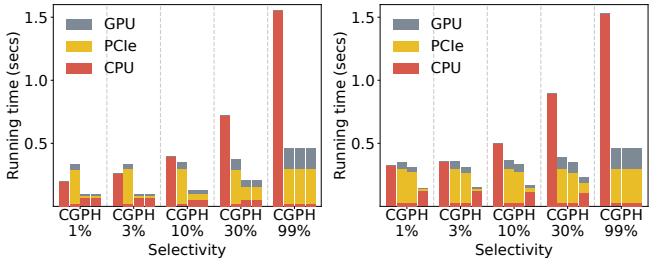
6.2 Microbenchmark

We first evaluate Hybrid using a microbenchmark, comparing it against the CPU-only (SQL Server) and GPU-only (TQP) approaches. The microbenchmark runs a join query on the filtered `Lineitem` and `Part` tables from the TPC-H 100GB dataset, where `Lineitem` is 30× larger than `Part`. We vary the filter selectivity on each table from 1% to 99% individually and measure cold-run performance. The benchmark query is shown below:

```
SELECT SUM(p_retailprice - l_extendedprice * (1 - l_discount)
FROM lineitem, part
WHERE l_partkey = p_partkey AND l_shipdate < X AND p_size < Y.
```

Figure 9a shows the execution time of all approaches as we vary the selectivity on the larger table (`lineitem`), while fixing the selectivity on the smaller table (`part`) at 99%. The execution time is broken down into CPU time, GPU time, and PCIe transfer time. For Hybrid, we report two variants: one using only predicate filtering (P) and the other using both predicate and bitvector filtering (H).

In the CPU-only approach, execution time increases substantially as as the scan become less selective, since more rows are produced by the scan and subsequently processed by the more expensive join operator. In the GPU-only approach, GPU execution time is substantially lower, nearly an order of magnitude less than the CPU execution time in the CPU-only approach, demonstrating the performance advantage of GPUs. However, total execution time is bottlenecked by PCIe transfer, which remains unchanged regardless of selectivity, since TQP always transfers full columns. When the filter is highly selective and the query is scan-heavy, the GPU-only approach becomes slower than CPU-only due to PCIe bandwidth limits, consistent with Observation 1 described in Section 2.2. As



(a) Varying selectivity on `lineitem` **(b) Varying selectivity on `part`**

Figure 9: Micro-benchmark comparison: C = CPU-only (SQL) vs. G = GPU-only (TQP-cold) vs. P = Hybrid w/ predicate filtering vs. H = Hybrid w/ both predicate and bitvector filtering

selectivity increases and the join dominates execution, GPU-only outperforms CPU-only, in line with Observation 2.

Unlike GPU-only, the Hybrid approaches consistently outperform CPU-only across all selectivities, with speedups ranging from 2.1× to 3.5×. Compared to GPU-only, Hybrid performs predicate filtering on the CPU to reduce PCIe transfer volume. Although this incurs additional CPU time, the added cost is more than offset by the reduction in PCIe transfer time, thanks to the efficient predicate filtering technique (see Section 4). This trade-off is especially effective when the filter is selective, where Hybrid achieves up to a 3.5× speedup. As selectivity increases and fewer rows are filtered out, the benefit diminishes. When nearly all rows are selected, Hybrid disables CPU-side filtering and reverts to GPU-only execution (Section 5.2). Since the filter on the smaller table is fixed at 99% and is not sufficiently selective, Hybrid does not apply bitvector filtering, resulting in identical performance for both P and H variants.

Figure 9b compares the approaches as we vary the selectivity on the smaller table, while keeping the larger table fixed at 99% selectivity. In this setting, predicate filtering alone offers limited benefit, as it does not substantially reduce total transfer volume. In contrast, bitvector filtering in Hybrid effectively propagates the selective filter from the smaller to the larger table, significantly reducing the data transfer volume of the larger table. Although this introduces additional CPU overhead for building and probing the bitvector, the savings in transfer time outweigh the overhead. Overall, Hybrid achieves up to a 2.4× speedup over the best of the CPU-only and GPU-only baselines in this scenario.

6.3 TPC-H 100GB

We next evaluate the efficiency of various approaches using the TPC-H benchmark at the 100GB scale. At this scale, the working set fits entirely in GPU memory, allowing all baselines to complete every query without triggering out-of-memory (OOM) exceptions[3]. While our hybrid approach is not specifically designed for this scenario, it provides a useful setting to compare the execution times of all approaches and to assess the overhead of cold runs.

Figure 10 shows the execution times for all the queries across the evaluated approaches, with total execution times summarized in Table 2. Among all approaches, TQP-hot is the fastest, which is

[3]As of the writing of this paper, HeavyDB does not support Q21 due to limitations of its query optimizer.

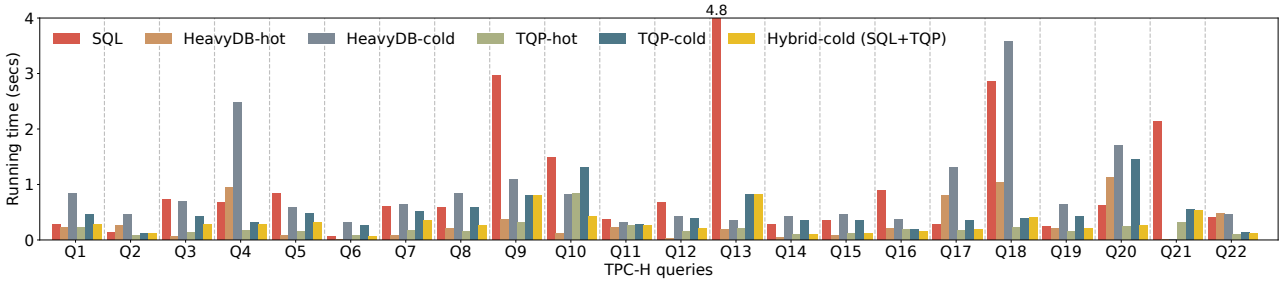


Figure 10: TPC-H 100GB comparison: CPU-only (SQL) vs. GPU-only (HeavyDB and TQP) vs. Hybrid (SQL + TQP)

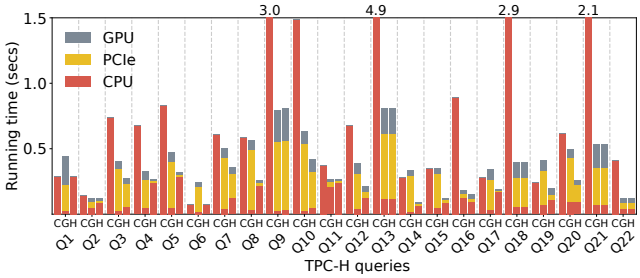


Figure 11: Time breakdown on 100GB TPC-H: C = CPU-only (SQL) vs. G = GPU-only (TQP-cold) vs. H = Hybrid

expected given that no PCIe transfer is required in hot runs. Compared to the other GPU-only approach, TQP outperforms HeavyDB by 1.5× in hot runs.

In cold runs, Hybrid achieves a 3.5× speedup over SQL Server, an improvement from the 2.4× speedup of TQP-cold. It also narrows the performance gap between hot and cold runs from 4.7 to 1.9 seconds, yielding a 2.3× reduction compared to TQP-cold. With this reduced overhead, Hybrid is only 45% slower than TQP-hot, despite data being transferred over the slow PCIe bus.

Figure 11 breaks down the execution time into three components: CPU, GPU, and PCIe time. Although TQP transfers compressed columns and decompresses them on the GPU, similar to prior work [12, 25, 51, 63], PCIe remains the primary bottleneck for nearly all queries. As a result, TQP shows mixed performance: it achieves substantial speedups on long-running queries but is slower than SQL Server on 6 out of 22 queries, with a maximum slowdown of 3.4× due to the PCIe overhead. In contrast, Hybrid matches or exceeds SQL Server on all queries.

By pushing data filtering to the CPU, Hybrid significantly reduces PCIe transfer time at the expense of increased CPU time. This trade-off is generally favorable, as the reduction in PCIe time outweighs the additional CPU cost. Moreover, CPU-side filtering also lowers GPU execution time by reducing the input size to the GPU engine. However, this benefit is less substantial, as GPU execution time is typically not the primary performance bottleneck.

There are a few exceptions. For 5 queries, Hybrid gains no benefit from data filtering compared to GPU-only: Q18 and Q21 lack selective predicates, while Q9, Q13, and Q22 include expensive filters that are better suited for GPU execution (see Section 4.4). In these cases, Hybrid sends full compressed columns to the GPU, identical to GPU-only. Conversely, Q1 and Q6 are scan-only queries that run faster on the CPU (Observation 1). As a result, Hybrid executes them fully on the CPU without sending any data to the GPU.

SQL	HeavyDB (excluding Q21)		TQP		Hybrid
	-hot	-cold	-hot	-cold	-cold
22.4 s	6.8 s	18.8 s	4.5 s	9.2 s	6.4 s

Table 2: Total execution time on 100GB TPC-H

6.4 TPC-H 1TB

In this set of experiments, we use the TPC-H 1TB benchmark to assess how the various solutions perform when the datasets significantly exceeds the GPU memory size. Since, at this scale,the working set of most queries cannot fit into GPU memory, we focus only on cold runs. We evaluate each approach in terms of performance and query coverage by reporting the execution time and the number of queries executable without OOM exceptions. These metrics directly expose the two main limitations of GPUs: the PCIe bottleneck and limited memory capacity.

6.4.1 Overall Comparison. Figure 12 shows the execution times for the CPU-only (*SQL*), GPU-only (*TQP* and *HeavyDB*), and hybrid (*Hybrid*) approaches, where "x" denotes queries that do not successfully run on the GPU due to OOM exceptions.

Given that the dataset substantially exceeds the GPU memory capacity, unsurprisingly, the GPU-only TQP approach executes merely 4 out of 22 queries without OOM exceptions. Three of the four runnable queries involve joins among smaller tables. The fourth one, Q22, accesses a single column of the second largest table (`orders`) in the database, which is small enough to fit into GPU memory. These results demonstrate that limited GPU memory capacity is a major challenge for scalable GPU database systems.

Similarly, the other GPU-only approach, HeavyDB, successfully executes 9 queries on the GPU without falling back to the CPU. However, despite our tuning efforts, 6 of these 9 queries are in fact significantly slower than the CPU-only approach, likely due to the PCIe bottleneck.

In contrast, our hybrid approach successfully executes all 22 queries at the 1TB scale, providing significant improvements over other approaches. The speedup over CPU-only is shown above each bar group in Figure 12. Overall, the speedups range from 0.8× to 9.1×, with an overall speedup of 3.5× across all queries. It outperforms CPU-only on 21 out of 22 queries.

6.4.2 Impact of Key Techniques. We next investigate the impact of each technique on execution time and the number of executable queries without OOM exceptions. We begin with the basic TQP approach, and incrementally enable additional techniques, one at a time, in the following order: streaming (Section 3.3), predicate

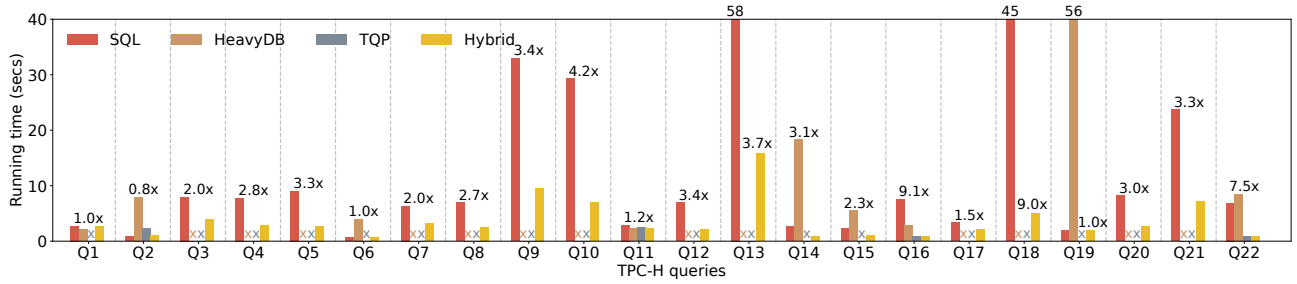


Figure 12: TPC-H 1TB comparison: CPU-only (SQL) vs. GPU-only (TQP and HeavyDB) vs. Hybrid. "x": queries causing OOMs.

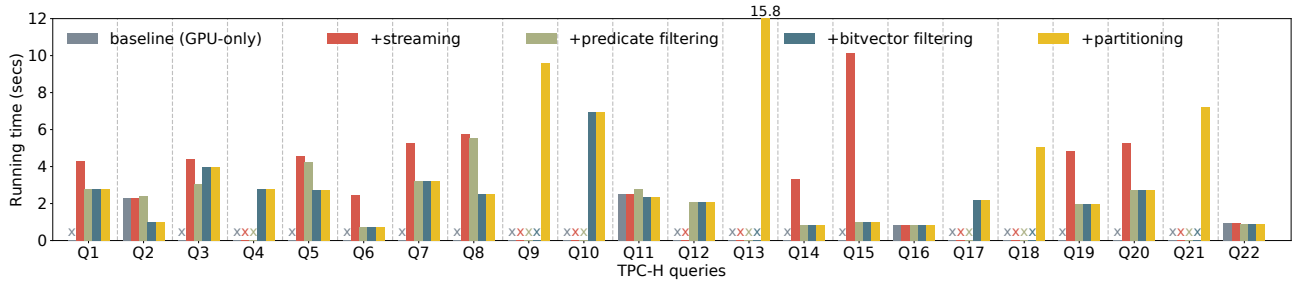


Figure 13: Impact of key techniques in Hybrid at TPC-H 1TB. "x" denotes queries causing OOM exceptions.

filtering (Section 4), bitvector filtering (Section 5), and partitioning (Section 3.3). The results are shown in Figure 13.

Streaming. Enabling streaming allows us to execute 10 additional queries without OOM exceptions. For these queries, the largest table is streamed to the GPU. Since, for the largest table, only the currently processed chunks need to be kept in GPU memory, the peak GPU memory usage reduces significantly. However, streaming alone does not bring significant performance benefits: 7 out of these 10 queries are either slower or only marginally faster than the CPU-only baseline. This is because streaming does not reduce the amount of data transferred via PCIe, which remains a bottleneck for many queries.

Predicate filtering. With predicate filtering enabled, we are able to run an additional query, Q12, compared to the streaming approach. Moreover, we find that predicate filtering eliminates the need for streaming of the largest tables in all but two queries, Q7 and Q20, as the reduction in table size from predicate filtering on the CPU enables these filtered tables to fit entirely in GPU memory. More importantly, predicate filtering brings significant performance gains by reducing the amount of data that needs to be transferred through PCIe. Compared to the streaming approach, we observe notable speedups in seven queries (Q1, Q6, Q7, Q14, Q15, Q19, and Q20). Most of these queries have highly selective filters on the largest table.

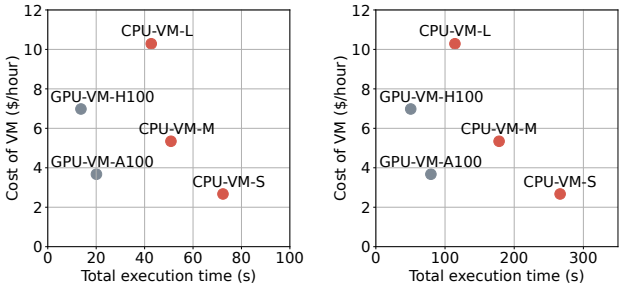
Bitvector filtering. Bitvector filtering enables three additional queries (Q4, Q10, and Q17) to run successfully and improves the performance of three other queries (Q2, Q5, and Q8). These six queries share a common pattern: the selective predicate filters are on the smaller tables, and the data filtering on the largest table can only be achieved through join conditions. In such cases, predicate filtering alone is insufficient. Bitvector filtering is essential for pushing filters across the join to the largest table, which reduces PCIe data transfer and lowers GPU memory consumption.

Partitioning. Enabling partitioning allows us to execute all four remaining queries (Q9, Q13, Q18, and Q21) without OOM exceptions. Although Q9 has a highly selective predicate filter, the filter is a string matching expression that is expensive to evaluate on the CPU and is therefore offloaded to the GPU (see Section 4.4). Similarly, the predicate filter in Q18, which applies to the result of group-by aggregates, is also offloaded to the GPU. As a result, these queries require transferring the full columns to the GPU. With partitioning, we add filters on the join keys on both sides of the largest join, allowing us to generate partitions with predicate filtering and to execute the query on each partition pair in sequence. This makes all four queries runnable without OOM exceptions and results in significant performance improvements over CPU-only execution: 3.4×, 3.7×, 9.0×, and 3.3× for Q9, Q13, Q18, and Q21, respectively. Note that while partitioning is necessary, it alone is not sufficient; the generated filters must still be applied to produce a filtered table via predicate filtering.

6.5 Cost-Effectiveness

In the next experiment, we evaluate the cost-effectiveness of the hybrid approach on two GPU VMs, comparing it against SQL Server running on CPU-only VMs. In addition to the NC24ads A100 v4 VM (*GPU-VM-A100*), we also evaluate the hybrid approach on the NC40ads H100 v5 VM with 40 cores and 320GB memory (*GPU-VM-H100*), which is equipped with a single NVIDIA H100 GPU [8], the successor to the A100, with the same 80GB of GPU memory. As shown in Table 1, the H100 VM offers 2.3× higher main memory bandwidth, 1.5× higher GPU memory bandwidth, and 2.0× higher PCIe bandwidth compared to the A100 VM.

For the CPU-only baselines, we run SQL Server on three VMs representing different price-performance points: an E32bds v5 VM with 16 CPU cores and 256GB memory (*CPU-VM-S*), an E64bds v5 VM with 32 cores and 512GB memory (*CPU-VM-M*), and an E112ibds v5 VM with 56 cores and 672GB memory (*CPU-VM-L*) [3]. Figure 14 plots the total execution time for all TPC-H queries versus VM cost at the 300GB and 1TB scales.



(a) TPC-H 300GB **(b) TPC-H 1TB**

Figure 14: Cost-effectiveness of CPU-only vs. Hybrid

We first compare the hybrid approach running on the GPU-VM-A100 against the CPU-only baselines. At the 300GB scale, the hybrid approach achieves a speedup of 2.1× over the most powerful VM (E112ibds) at only 35% of the cost. This translates to a 6.0× improvement in perf/$. When compared to the most economical VM (E32bds), the hybrid approach achieves a 3.6× speedup at 1.4× the cost, which results in 2.6× better cost-efficiency. At the 1TB scale, the hybrid approach achieves a 1.4× speedup over the most powerful VM (E112ibds) at just 35% of the cost, resulting in a 4.0× improvement in perf/$. Compared to the more economical VM (E32bds), the hybrid approach is 37% more costly but delivers a speedup of 3.4×. Overall, the hybrid approach on A100 achieves higher performance than the most powerful CPU-only VM, at a cost that is comparable to the most economical option.

We further evaluate the hybrid approach using the GPU-VM-H100, which delivers the best performance among all configurations tested. Compared to SQL Server running on the most powerful CPU VM (CPU-VM-L), it achieves a 3.7× speedup at the 300GB scale and a 2.3× speedup at the 1TB scale, while costing only 67% as much. This translates to 5.5× and 3.4× improvements in perf/$ at the 300GB and 1TB scales, respectively. Together, the hybrid approach running on the A100 and H100 VMs provides two cost-effective options at different price-performance points, both of which substantially outperform CPU-only baselines in terms of cost-efficiency.

7 RELATED WORK

Query processing within GPU memory. One of the first research prototypes of GPU database engines is the columnar CoGaDB [18]. Ocelot [33] is a hardware-oblivious extension of MonetDB that uses OpenCL to run on GPUs. Open-source solutions include BlazingSQL [4], HeavyDB [5], and RAPIDS for Spark [9]. Prior research has also explored using PyTorch as a data processing runtime to target multiple hardware accelerators [13, 31]. A rich body of literature studies efficient algorithms and implementations of relational operators for GPUs [30, 44, 48, 53]. Beyond operator-level optimizations, prior work has studied operator fusion through query compilation to minimize data movement from and to GPU [27, 43]. Other recent research work has focused on data processing and placement in a multi-GPU setting [5, 21, 44, 48, 56, 61]. Our work is complementary to these approaches: Hybrid execution can leverage these optimizations to improve on-GPU performance, while these systems can benefit from a hybrid CPU-GPU execution to handle workloads that exceed GPU memory capacity.

Query processing beyond GPU memory size. The streaming execution model has been adopted to reduce memory usage by transferring and processing data in smaller chunks [4, 27, 52]. The performance impact of limited PCIe bandwidth on GPU query processing has been extensively studied [63]. To mitigate this bottleneck, prior research has proposed transferring compressed data and performing decompression on GPU [12, 25, 51, 63]. However, as shown in our evaluation, these techniques are insufficient to handle databases that significantly exceed GPU memory capacity.

Hybrid CPU-GPU execution. He et al. [29] explore coprocessing between CPU and early GPU designs. Kaldewey et al. [36] demonstrate how a database system can leverage unified virtual addressing to process data on the GPU at PCIe bandwidth. Chrysogelos et al. [21] introduce the HetExchange operator, which selectively executes parts of the query plan on the GPU. Yogatama et al. [62] propose data placement strategies for heterogeneous execution. Prior work on hybrid execution has considered coupled architectures [22, 32], where the CPU and GPU share the same die, enabling much higher data transfer bandwidth. Moreover, Hybrid execution has been applied to specific relational operators such as joins and sort [52, 53]. Beyond relational operators, recent work explores hybrid coprocessing for index lookups [28, 38] by leveraging the complementary strengths of CPUs and GPUs. Commercial database systems, such as DB2 BLU [41], have also explored hybrid execution. A recent survey by Rosenfeld et al. [47] provides a comprehensive overview of this area. Our hybrid approach differs from prior work in its focus on scalability: it leverages CPUs to handle workloads that significantly exceed GPU memory capacity, leading to a distinct set of design choices.

Fast scans and bitvector filtering on CPUs. The scan operator has been extensively optimized through various techniques, such as SIMD vectorization [11, 46, 57, 58, 64], direct predicate evaluation on compressed data [10, 35, 40], and parallel processing of compressed values fitting within a processor word [26, 35, 39, 40]. Our approach benefits from these advancements, particularly in the predicate evaluation step. For the projection step, we have adapted the approach outlined in [39] to develop our customized scan operator that outputs compressed data directly. Bitvector filtering has been inspired by semi-join reduction [16, 60] and sideway information passing [15]. It has been widely used in database systems to improve query performance [23, 42, 65]. Prior work has explored how to incorporate bitvector filtering in query execution [20, 59] and query optimization [23].

8 CONCLUSIONS

In this work, we propose hybrid CPU-GPU processing for analytical database systems. We demonstrate that data filtering on the CPU, specifically through predicate and bitvector filtering, can be made highly efficient, and it effectively reduces data transfer over PCIe to improve query performance. When combined with two additional techniques, streaming and partitioning, data filtering further decreases the demand on GPU memory. This reduction extends the scope of GPU acceleration to databases that significantly exceed GPU memory size. Beyond performance improvements, our findings also indicate that the hybrid approach presents a compelling cost-effective alternative to traditional CPU-only systems.

REFERENCES

[1] [n.d.]. *TPC Benchmark H Standard Specification. Revision 2.17.1.* https://www.tpc.org

[2] 2025. Arm A64 Instruction Set Architecture: SVE Instructions. https://developer.arm.com/documentation/ddi0602/2025-03/SVE-Instructions. [Online; accessed July-2025].

[3] 2025. Azure Virtual Machines Pricing. https://https://azure.microsoft.com/en-us/pricing/details/virtual-machines/. [Online; accessed Feb-2025].

[4] 2025. BlazingSQL. https://github.com/BlazingDB/blazingsql. [Online; accessed Feb-2025].

[5] 2025. HeavyDB. https://www.heavy.ai/product/heavydb. [Online; accessed Feb-2025].

[6] 2025. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server. [Online; accessed Feb-2025].

[7] 2025. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/. [Online; accessed Feb-2025].

[8] 2025. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/. [Online; accessed Feb-2025].

[9] 2025. RAPIDS Accelerator For Apache Spark. https://github.com/NVIDIA/spark-rapids. [Online; accessed Feb-2025].

[10] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. ACM, 671–682. https://doi.org/10.1145/1142473.1142548

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

[12] Azim Afroozeh, Lotte Felius, and Peter A. Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 8:1–8:11. https://doi.org/10.1145/3662010.3663450

[13] 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. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3598–3601. https://doi.org/10.14778/3554821.3554853

[14] R. J. Atwal, Peter A. Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves Le Maout, Boaz Leskes, Yao Liu, Alex Monahan, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. 2024. MotherDuck: DuckDB in the cloud and in the client. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p46-atwal.pdf

[15] C. Beeri and R. Ramakrishnan. 1987. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '87)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/28659.28689

[16] Philip A. Bernstein and Nathan Goodman. 1981. Power of Natural Semijoins. *SIAM J. Comput.* 10, 4 (1981), 751–771. https://doi.org/10.1137/0210059 arXiv:https://doi.org/10.1137/0210059

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

[18] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (2014), 199–209.

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

[20] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1993. Applying Hash Filters to Improving the Execution of Bushy Trees. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 505–516.

[21] 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

[22] 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, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, Norman May and Nesime Tatbul (Eds.). ACM, 86–88. https://doi.org/10.1145/3592980.3595313

[23] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. 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, 2011–2026. https://doi.org/10.1145/3318464.3389769

[24] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[25] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1 (2010), 670–680. https://doi.org/10.14778/1920841.1920927

[26] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 31–46. https://doi.org/10.1145/2723372.2747642

[27] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1603–1618. https://doi.org/10.1145/3183713.3183734

[28] Tobias Groth, Sven Groppe, Thilo Pionteck, Franz Valdiek, and Martin Koppehel. 2022. Accelerated Parallel Hybrid GPU/CPU Hash Table Queries with String Keys. In *Database and Expert Systems Applications - 33rd International Conference, DEXA 2022, Vienna, Austria, August 22-24, 2022, Proceedings, Part II (Lecture Notes in Computer Science)*, Christine Strauss, Alfredo Cuzzocrea, Gabriele Kotsis, A Min Tjoa, and Ismail Khalil (Eds.), Vol. 13427. Springer, 191–203. https://doi.org/10.1007/978-3-031-12426-6_15

[29] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (Dec. 2009), 39 pages. https://doi.org/10.1145/1620585.1620588

[30] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[31] Dong He, Supun Chathuranga 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. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[32] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 329–340. https://doi.org/10.14778/2735496.2735497

[33] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[34] Intel Corporation. 2025. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference.* Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html,.

[35] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.* 1, 1 (2008), 622–634. https://doi.org/10.14778/1453856.1453925

[36] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[37] Donald E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams* (12th ed.). Addison-Wesley Professional.

[38] Martin Koppehel, Tobias Groth, Sven Groppe, and Thilo Pionteck. 2021. CuART - a CUDA-based, scalable Radix-Tree lookup and update engine. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 12:1–12:10. https://doi.org/10.1145/3472456.3472511

[39] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2 (2023), 178:1–178:26. https://doi.org/10.1145/3589323

[40] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 289–300. https://doi.org/10.1145/2463676.2465322

[41] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. 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,

1951–1960. https://doi.org/10.1145/2882903.2903735

[42] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New query optimization techniques in the Spark engine of Azure synapse. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 936–948. https://doi.org/10.14778/3503585.3503601

[43] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 202–214. https://doi.org/10.14778/3425879.3425890

[44] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[45] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1707–1718. https://doi.org/10.14778/3007328.3007336

[46] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 1493–1508. https://doi.org/10.1145/2723372.2747645

[47] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (Jan. 2022), 38 pages. https://doi.org/10.1145/3485126

[48] 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

[49] Donovan A. Schneider and David J. DeWitt. 1989. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 110–121. https://doi.org/10.1145/67544.66937

[50] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[51] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1390–1403. https://doi.org/10.1145/3514221.3526132

[52] 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

[53] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. 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, 417–432. https://doi.org/10.1145/3035918.3064043

[54] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. ACM, 553–564. http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf

[55] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around.. *SIGMOD Rec.* 53, 2 (2024), 21–37. https://doi.org/10.1145/3685980.3685984

[56] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM Manag. Data* 1, 1, Article 29 (May 2023), 26 pages. https://doi.org/10.1145/3588709

[57] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013*. 1–12. http://www.adms-conf.org/2013/muller_adms13.pdf

[58] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (2009), 385–394. https://doi.org/10.14778/1687627.1687671

[59] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf

[60] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[61] 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

[62] 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

[63] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[64] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 145–156. https://doi.org/10.1145/564691.564709

[65] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking ahead makes query plans robust: making the initial case with in-memory star schema data warehouse workloads. *Proc. VLDB Endow.* 10, 8 (April 2017), 889–900. https://doi.org/10.14778/3090163.3090167

[66] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working Smarter, Not Harder. *Proc. ACM Manag. Data* 3 (2025).