FaScalSQL: A Fast and Scalable GPU-Accelerated SQL Query Engine for Out-of-Memory Tables

Chaemin Lim*, Suhyun Lee*, Jinwoo Choi†, Kwanghyun Park*, Jinho Lee‡, Joonsung Kim§, Youngsok Kim*
*Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea
†Korea Institute of Science and Technology, Seoul, South Korea
‡Department of Electrical and Computer Engineering, Seoul National University, Seoul, South Korea
§Department of Semiconductor Systems Engineering, Sungkyunkwan University, Suwon, South Korea
cmlim@yonsei.ac.kr, su_hyun@yonsei.ac.kr, jinwoo1029@kist.re.kr, kwanghyun.park@yonsei.ac.kr, leejinho@snu.ac.kr, joonsungkim@skku.edu, youngsok@yonsei.ac.kr

***Abstract*—Graphics Processing Units (GPUs) are promising for analytical SQL query processing, but their limited memory capacity hinders processing large input tables exceeding the GPU memory. The existing engines either 1) statically split input columns into chunks and iteratively perform host-to-GPU transfer and relational operations (streaming engines), or 2) maintain a static and fixed-size cache on GPU memory and distribute input columns and their query workloads to the host CPU and a GPU (CPU-GPU distributive engines). However, we find that they suffer from two primary bottlenecks which eventually lead the engines to severe GPU underutilization: excessive host-to-GPU data movement and CPU-GPU load imbalance. We find that they arise from a conflict between their static input data placement and the dynamic progressive filtering of analytical queries. This conflict leads the engines either to transfer column values that are eventually discarded or to assign a large amount of the workload to the host CPU as the input table size scales.**

In this paper, we present FaScalSQL, a fast and scalable GPU-accelerated SQL query engine that overcomes the severe GPU underutilization of query processing on out-of-memory tables. FaScalSQL introduces a new type of on-demand CPU-GPU co-processing engine which exploits both GPU-initiated data transfer and CPU-GPU co-processing capability. It replaces the static large unfiltered chunks with a dynamic GPU-initiated on-demand fetching of necessary input data, guided by the host CPU's pre-filtering. We evaluate FaScalSQL with Star Schema Benchmark (SSB) and TPC-H. Using SSB with scale factors of 100 and 200, FaScalSQL achieves geometric mean speedups of 2.60× and 2.20× over the existing streaming and CPU-GPU distributive engines.

***Index Terms*—GPU acceleration, OLAP, out-of-memory tables, CPU-GPU co-processing, CPU-GPU load imbalance**

I. Introduction

Graphics Processing Units (GPUs) offer much higher computational throughput and memory bandwidth than Central Processing Units (CPUs) [58], [81]. This makes GPUs promising for analytical Structured Query Language (SQL) query processing, which evaluates the relational operations of a SQL query on multiple input columns and their values [16]. Prior studies show that GPUs can greatly accelerate analytical SQL queries by parallelizing the execution of a relational operation on different input column values across GPU cores [15], [23], [25], [26], [35], [40], [54], [57], [87], [92], [93], [98], [116].

However, the limited capacity of GPU memory poses a critical challenge in GPU-accelerated SQL query processing. The demand for processing large tables is steadily increasing, for instance, Google and Meta report exponential growth in their web-scale SQL analytics over the last decade [72], [104]. Meta has recently reported that the size of scanned input tables increases by nearly 600% over three years [73], [104].

Several GPU-accelerated analytical SQL query engines have been proposed to handle out-of-memory tables. They can be categorized into two types: streaming engines and CPU-GPU distributive engines. Streaming engines statically split input columns into GPU memory-fit chunks, iteratively performing the host-to-GPU transfer and executing relational operations on each chunk in a pipelined manner [17], [37], [51], [68], [69], [101], [129]. CPU-GPU distributive engines maintain a static, fixed-size cache on the GPU memory and a GPU executes relational operations for the column values within the cache. They rely on the host CPU for executing relational operations on the column values, which cannot reside in the GPU memory cache [11], [13], [14], [47], [126].

However, existing engines create two primary bottlenecks which incur severe GPU underutilization: excessive host-to-GPU data movement and CPU-GPU load imbalance. We identify these bottlenecks as arising from conflict between the engines' static input data placement and the queries' dynamic progressive filtering. For example, statically split chunks in streaming engines saturate the PCIe bus with values that are eventually discarded. Similarly, static caching of CPU-GPU distributive engines compels the host CPU to process a large share of the workload as data scales. This can be further exacerbated by host-side contention incurred by co-located applications and kernel routines [4], [43], [65], [109].

In this paper, we present FaScalSQL, a fast and scalable GPU-accelerated SQL query engine that overcomes severe GPU underutilization in out-of-memory query processing, where progressive filtering prevents the GPU from fully exploiting its high degree of parallelism. FaScalSQL introduces a new type of on-demand CPU-GPU co-processing engine that changes the input data placement by exploiting both GPU-initiated data transfer and CPU-GPU co-processing capability. To align data movement with the query's progressive filtering, FaScalSQL replaces the static large unfiltered chunks with a dynamic GPU-initiated on-demand fetching of necessary input data guided by the host CPU's proactive pre-filtering.

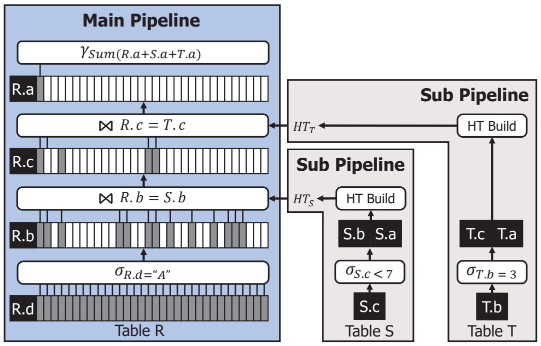


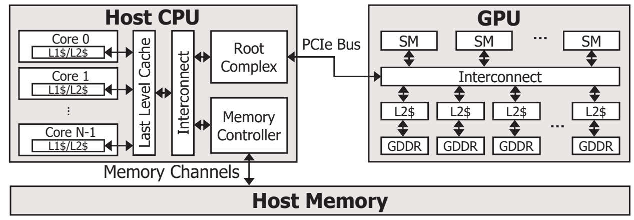
Fig. 1: A pipeline-driven execution of an example SQL query: SELECT *SUM(R.a+S.a+T.a)* FROM *R, S, T* WHERE *R.b=S.b* AND *R.c=T.c* AND *R.d="A"* AND *S.c<7* AND *T.b=3*

FaScalSQL is built upon three synergistic techniques: First, On-Demand Zero-copy Caching (ODZC) enables the GPU's fine-grained access to host memory, ensuring the data transfer reflects the progressive sparsity of valid input column values. Upon ODZC, we propose Asynchronous Filter Pushdown (AFP) to proactively pre-filter the unnecessary input columns that the GPU needs to access in the first place using the host CPU asynchronously with the GPU. This CPU-GPU co-processing, in turn, can cause the host CPU to be a new bottleneck, a CPU-GPU load imbalance. Finally, to make the co-processing robust, a Contention-aware Query Optimizer (CQO) adaptively manages host-side involvement, considering varying CPU resource availability.

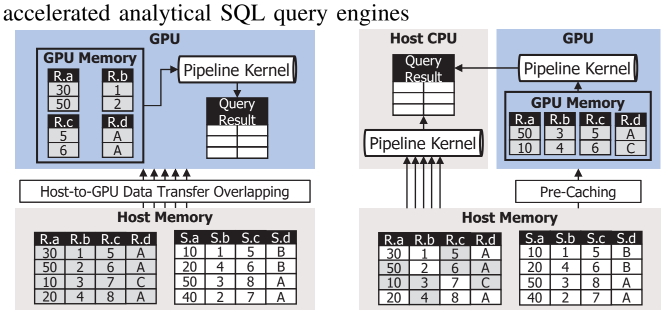
We implement FaScalSQL on a real GPU-equipped system having an NVIDIA RTX A4000 GPU, and then compare FaScalSQL's query processing performance against the state-of-the-art GPU-accelerated analytical SQL query engines. We use the Star Schema Benchmark (SSB) [86] and TPC-H [108] queries. Using SSB with scale factors of 100 and 200, FaScalSQL achieves geometric mean speedups of 2.60× and 2.20× over HetExchange and Mordred, respectively. For the SSB with a scale factor of 100 (∼60 GB of total table size), FaScalSQL reduces host-to-GPU data movement by 39.36× compared to streaming HetExchange [17], and achieves a geometric mean speedup of 11.48× over CPU-GPU distributive Mordred [126] under severe host CPU-side contention.

In summary, this paper makes the following contributions:

- We identify and analyze the conflict between existing engines' static input data placement and analytical queries' dynamic progressive filtering, leading to excessive host-to-GPU data movement and CPU-GPU load imbalance.
- We propose FaScalSQL, a new on-demand CPU-GPU co-processing engine that significantly enhances the effectiveness of GPU-initiated data fetching for analytical queries. It resolves GPU underutilization and achieves scalable performance by synergistically combining on-demand GPU-initiated data fetching, CPU-driven asynchronous pre-filtering, and contention-aware query optimization.
- We implement FaScalSQL on real GPU-equipped systems and show its superior performance over the existing GPU- and CPU-based SQL query engines using SSB and TPC-H.



(a) The underlying system architecture for out-of-memory GPU-accelerated analytical SQL query engines



(b) Streaming (c) CPU-GPU distributive

Fig. 2: Working models of the existing GPU-accelerated SQL query engines for out-of-memory input columns

II. Background

A. Characteristics of GPU-Based Pipeline-Driven Execution

Recent studies [11], [14], [17], [26], [47], [65], [68], [87], [88], [98], [116], [126], [130] have proposed GPU-accelerated analytical SQL engines that execute SQL relational operations on GPUs using a pipeline-driven execution model [59], [78], [87], [88], [98]. With column-oriented tables in host memory [1], query planners decompose the SQL query into pipelines by pipeline breakers (e.g., joins) [78], and assign each pipeline an input table and its operations. Relational operators in each pipeline are fused into a single GPU kernel, invoked according to pipeline dependencies.

GPU-based pipeline-driven execution has two key traits: dynamic progressive filtering and kernel fusion. Predicates applied in the pipeline progressively filter out rows, so only alive rows proceed, with their liveness tracked in a bit-vector. Kernel fusion—widely used in these engines [17], [25], [26], [87], [98], [116]—merges multiple operators into a single GPU kernel to avoid costly materialization, retaining intermediate data in on-chip storage. As shown in Fig. 1, as filtering progresses (e.g., R.d="A"), the valid rows become sparser, and only these must be accessed in later operations (e.g., R.b, R.c for hash probes), making data access increasingly selective across the pipeline. Sub-pipelines build hash tables first, and then the main pipeline emits the final result.

B. Processing Out-of-Memory Columns on a GPU

Processing large tables exceeding GPU memory requires storing them in host memory and transferring them via the PCIe bus, as shown in Fig. 2a. Two main approaches have been proposed for the large tables, as summarized in Fig. 2.

- **Streaming engines** [17], [37], [51], [68], [69], [101], [129] statically split input columns into large, GPU-fit chunks. These chunks are streamed to the GPU and processed by the pipeline kernel, which allows overlapping the data transfer with kernel execution. For instance, as shown in Fig. 2b, chunks of table R are sequentially transferred and processed.

- **CPU-GPU distributive engines** [11], [13], [14], [38], [47], [50], [126] use a partial caching model. Based on offline profiling, frequently accessed data is cached in GPU memory. The GPU processes these statically cached chunks, while the host CPU handles the remaining data, merging the results at the end. For example, as shown in Fig. 2c, parts of table R are pre-cached and processed by the GPU, while uncached portions fall back to the CPU. Finally, the host CPU merges the intermediate results to produce the query output.

III. Motivation

We observe that both streaming and CPU-GPU distributive engines, the two primary approaches for processing out-of-memory columns, fail to achieve high scale-up performance. This failure stems from two critical bottlenecks: excessive host-to-GPU data movement and CPU-GPU load imbalance. We identify that these come from the conflict between the static input data placement of existing engines (i.e., fixed and dense chunks or caches) and the dynamic progressive filtering of analytical queries. This leads streaming engines to wastefully transfer data that is eventually discarded, and CPU-GPU distributive engines to offload a large share of the workload to the host CPU as data scales. We use a GPU-equipped system detailed in §V-A for the subsequent analyses.

A. Excessive Host-to-GPU Data Movement

Streaming engines statically split host-resident input columns into dense chunks. They treat each chunk as a separate in-memory execution, forcing the engine to move entire chunks over the limited PCIe bus, oblivious to the progressive sparsity created by the predicates of relational operations. To quantify excessive host-to-GPU data movement, we evaluate streaming HetExchange [17] and compare it with DuckDB [20], a widely-used CPU-based SQL query engine. Fig. 3 shows the latency breakdown of SSB query executions of HetExchange and DuckDB with a scale factor of 100. The results show that the host-to-GPU data transfer latency significantly overshadows the fast GPU query executions since the GPU kernel execution latencies of HetExchange are much lower than DuckDB. Especially, this excessive data movement overhead accounts for up to 93.5% of the total query execution latency with Q1.3. Since the host-to-GPU data movement is conducted through PCIe buses, whose bandwidths (32 GB/s for PCIe 4.0 x16) are far lower than intra-GPU memory bandwidths (448 GB/s for RTX A4000), this bandwidth disparity exacerbates the host-to-GPU data movement bottleneck.

B. CPU-GPU Load Imbalance

A static and fixed-size cache makes the GPU only process the part of the input columns that resides in its cache; otherwise, the workloads are assigned to the host CPU. This static workload distribution cannot adapt to dynamic progressive filtering, inevitably causing severe CPU-GPU load imbalance

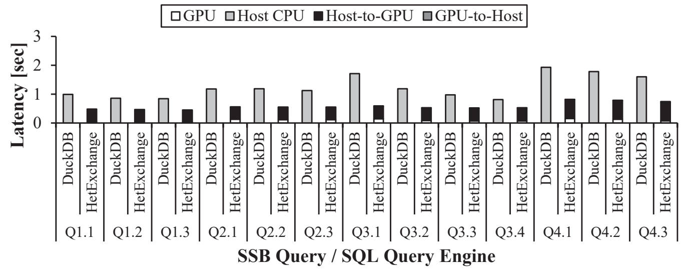
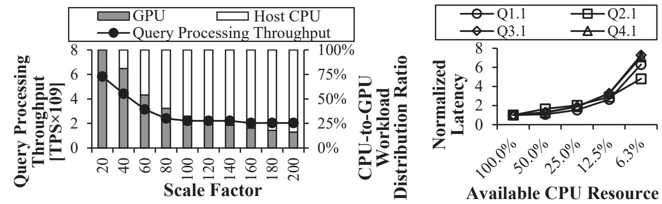


Fig. 3: Latency breakdown of the SSB queries of DuckDB [20] and HetExchange [17] on RTX A4000 (PCIe 4.0 x16)



(a) Query processing throughput and workload distribution ratio

(b) Normalized query execution latencies

Fig. 4: Performance analyses of Mordred [126] with varying scale factors and CPU resource availability

as data scales. To quantify the CPU-GPU load imbalance, we evaluate state-of-the-art CPU-GPU distributive Mordred [126]. As shown in Fig. 4a, when the scale factor grows, the workload share offloaded to the host CPU rises to 83.7%, causing a 65.0% drop in throughput. This demonstrates that the static workload distribution is inherently not scalable with data size.

Even worse, static workload distribution makes these engines significantly vulnerable to host CPU-side contention which has been commonly observed in multi-tenant cloud environments, database servers, and modern data centers [28], [49], [60], [119]. Such CPU-intensive query processing often requires sharing physical resources with other co-located applications, background system routines, and long-running services [77], [110], [123], [127]. To quantify this critical vulnerability, we evaluate the engine's performance under varying CPU resource availability by reducing the core count from 16 to 1 (i.e., 100% to 6.3%). As Fig. 4b shows, query execution latencies increase sharply as available resources diminish, demonstrating that the static workload distribution is not robust and vulnerable to the host CPU-side contention.

IV. FaScalSQL

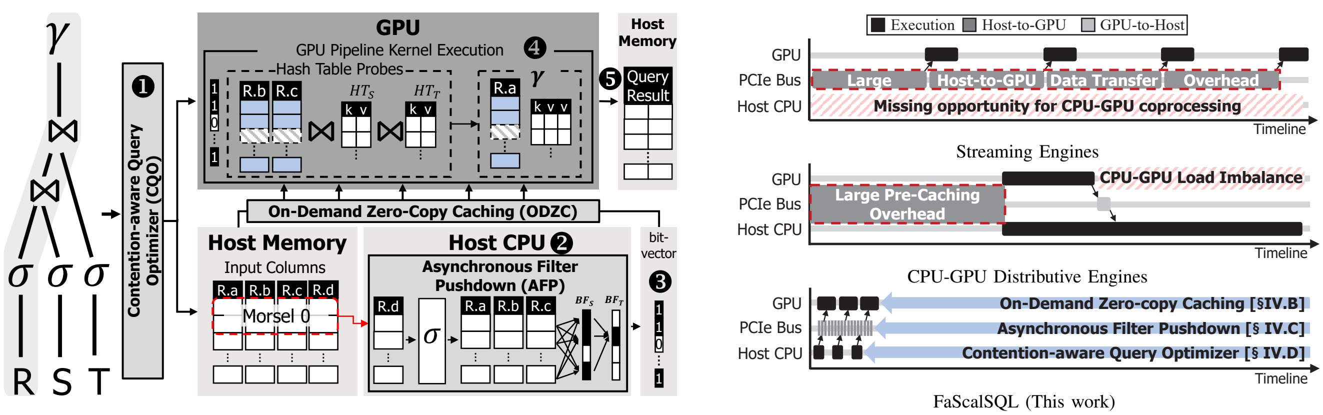
A. Design Goals and Overview

We present FaScalSQL, a fast and scalable GPU-accelerated SQL query engine for out-of-memory tables. Our primary goal is to resolve the conflict between the static input data placement and the dynamic progressive filtering of analytical queries. FaScalSQL directly tackles this by fundamentally changing the input data placement from large unfiltered chunks into GPU-initiated on-demand input data fetching. To achieve this, we propose a new class of engine: an *on-demand CPU-GPU co-processing* engine built upon three design goals.

1) *Align with dynamic sparsity:* The engine must align physical data access with the query's progressive filter-

TABLE I: Summary of prior GPU-accelerated SQL query engines and the novelty of FaScalSQL over the prior engines

Engine	Key Ideas	Performance Characteristics	Conflict w/ Dynamic Progressive Filtering	FaScalSQL's Novelty Over Prior Engines
In-Memory: Process analytical SQL queries for tables which fully reside in the GPU memory				
DogQC [26] Pyper [87] Themis [40]	Mitigation of GPU pipeline kernels' thread divergence	**Fast ✓** **Not scalable ✗** (GPU mem. size-bound)	N/A (No support for out-of-memory tables)	Can process out-of-memory tables using fine-grained on-demand host memory access (ODZC), which avoids the need to fit all data in capacity-limited GPU memory
Streaming: Process analytical SQL queries by streaming GPU memory-fit table chunks from the host memory				
Sioulas et al. [101] Triton Join [69]	GPU-centric, join-specific pipelining	**Scalable ✓** **Not fast ✗** (PCIe B/W-bound)	**Conflict** (Excessive data movement)	Provide a generic data reduction scheme that applies to any selective predicate within a query
Raza et al. [92]	GPU-initiated data transfer for SQL query processing			Combine on-demand GPU-initiated data transfer with CPU-side pre-filtering for efficient host-to-GPU data transfer
Lutz et al. [68] Vortex [129]	Input column streaming through fast interconnects			Exploit query semantics to logically reduce the data movement, instead of relying on fast interconnect
Saber [51] HeavyDB [37] HetExchange [17]	GPU-centric or static work sharing			Replace static chunk-based streaming with fully dynamic caching by ensuring only valid data, determined during query processing, get transferred to GPU memory
CPU-GPU Distributive: Distribute analytical SQL query processing workloads between a static fixed-size GPU memory cache and the host memory				
Ocelot [14], [38] HERO [47]	Adaptive CPU-GPU operator placement	**Fast** (on cache) **✓** **Not scalable ✗** (Host CPU-bound)	**Conflict** (CPU-GPU load imbalance)	Make CPU-GPU collaboration be dynamically adaptive to underlying system setups and host CPU-side contention
CoGaDB [11], [13] Kinetica [50] Mordred [126]	Locality-based input column partitioning onto the static GPU cache			Eliminate the inflexibility of static, profile-based caching by removing pre-caching and dynamically balancing the host CPU and GPU loads
On-Demand CPU-GPU Co-Processing: Combine GPU-initiated dynamic on-demand data transfer and the host CPU-side pre-filtering				
FaScalSQL (This work)	ODZC, AFP, and CQO	**Fast ✓** **Scalable ✓**	**No conflict**	Align with analytical queries' dynamic sparsity with ODZC, proactively minimize data movement with AFP, and ensure robust CPU-GPU co-processing with CQO



(a) The working model of FaScalSQL (b) Query execution timelines

Fig. 5: An overview of FaScalSQL and a comparison of its query execution timeline

ing. This is realized by *On-Demand Zero-copy Caching (ODZC)*, which shifts from a static to a dynamic GPU-initiated on-demand host memory access.

2) *Proactively minimize data movement:* The engine must intelligently guide the GPU to fetch only essential data. *Asynchronous Filter Pushdown (AFP)* achieves this by using the host CPU to pre-filter rows before they are ever moved over the PCIe bus.
3) *Ensure robust co-processing:* This CPU-GPU collaboration must be both robust and scalable. The *Contention-aware Query Optimizer (CQO)* adaptively manages the AFP workload based on system conditions, preventing the host CPU from becoming a new bottleneck.

As detailed in Table I, this unique combination of fine-grained, asynchronous, and adaptive techniques distinguishes FaScalSQL from prior streaming and distributive engines.

Working Model. Fig. 5 shows the working model of FaScalSQL. The pipeline execution process is as follows:

❶ CQO determines the optimal placement of AFP operations by analyzing system resource availability.
❷ The host CPU applies AFP operations (e.g., bloom filter lookups) to data morsels, proactively pruning unnecessary rows before they are ever accessed by the GPU.
❸ To minimize data movement, the host CPU marks the positions of pruned rows in a shared bit-vector, making the data's dynamic sparsity visible to the GPU.
❹ The GPU asynchronously begins its kernel execution, using ODZC and the bit-vector to fetch only valid, sparsely located column values, thus aligning GPU's input column access with the query's dynamic behavior.
❺ This co-processing continues until all input values are processed and the final result is returned to host memory.

Fig. 5b shows the execution timelines of the three types of engines. Unlike streaming engines, stalled by large data transfer overheads, FaScalSQL minimizes data movement via AFP and overlaps the data transfer with computation. Unlike distributive engines that suffer from CPU-GPU load imbalance and large pre-caching overheads, FaScalSQL's adaptive co-processing, guided by CQO, removes CPU-GPU load imbalance.

B. Aligning with the Dynamic Progressive Filtering Feature

The first step toward a scalable out-of-memory engine is to align the input data placement with the dynamic progressive filtering of analytical queries. We propose On-Demand Zero-copy Caching (ODZC), a technique enabling GPU's sparsity-aware, fine-grained, on-demand access to the host memory.

*1) **Opportunity: On-Demand Access to Host Memory**:* ODZC leverages modern GPU capabilities for direct host memory access: Unified Virtual Memory (UVM) [33] and zero-copy [82]. UVM migrates coarse-grained 4 KB pages upon a fault [27], whereas zero-copy allows GPU kernels to fetch data at a finer granularity, from a sector to a cache line (e.g., 32–128 B) [3], [74], [75]. They present an opportunity to fetch input column values only when required.

*2) **Challenge: Invalid Assumptions on GPU Working Set Size**:* The key challenge in applying on-demand host memory access comes from the assumption that a GPU's working set is always a dense, GPU-resident column. While this assumption is valid for in-memory scenarios, enabling upfront, coalesced loading to maximize internal GPU bandwidth [26], [88], [98], which breaks down for out-of-memory processing.

Naively extending this assumption to out-of-memory data leads to a severe bottleneck: massive data transfer amplification over the PCIe bus. This occurs because analytical queries inherently create progressive sparsity by filtering rows. Consequently, the true working set is no longer a dense block, but rather a sparse, logically-defined subset of values scattered across host memory, which must be processed on demand.

Algorithm 1: Sparsity-Aware Load Reordering

```
   Input  : OriginalSequence (Input pipeline kernel code's operation sequence)
   Output: ReorderedSequence (Reordered sequence for the input pipeline)
1  ReorderedSequence ← ∅, loadedColumns ← ∅
2  for each op in OriginalSequence do
3      if op is RelationalOperation then
4          for each col in op.inputColumns do
5              if col ∉ loadedColumns then
6                  loadOp ← OriginalSequence.find(col)
7                  ReorderedSequence.append(loadOp)
8                  loadedColumns.append(col)
9              end
10         end
11     end
12     if op is not LoadOperation then
13         ReorderedSequence.append(op)
14     end
15 end
```

*3) **Key Idea: On-Demand Zero-copy Caching (ODZC)**:* To make GPUs' on-demand access capability effective for analytical query processing, ODZC first introduces a sparsity-aware load reordering algorithm (Algorithm 1). This repositions memory load operations of column values at the very front of the first relational operation that actually consumes them. Algorithm 1 starts with taking an initial operation sequence and produces a reordered sequence of the input pipeline. It tracks loaded input columns using *loadedColumns*. It iterates through each operation and checks if its input columns are already loaded for relational operations (Lines 2-5). If not, it inserts load operations for the unloaded columns into *reorderedSequence* and updates *loadedColumns* (Lines 6-8). Finally, it appends the current operation (excluding load operations) to *reorderedSequence*, ensuring load operations are placed just before dependent relational operations (Line 13).

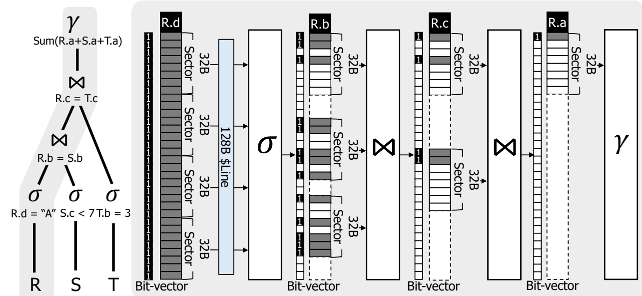


Fig. 6: FaScalSQL's pipeline execution with ODZC

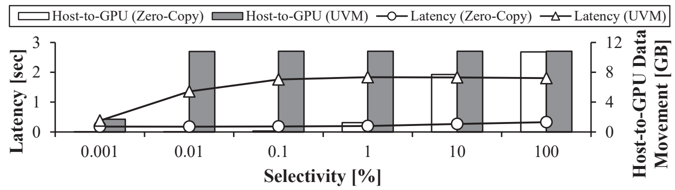
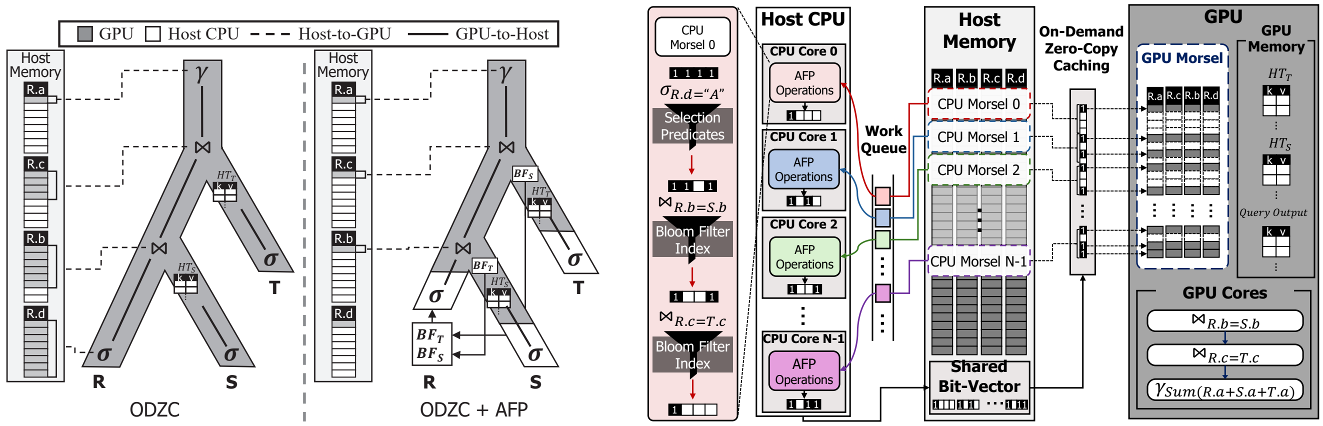


Fig. 7: Latencies and host-to-GPU data movements of ODZC with zero-copy and ODZC with UVM for an example SQL query: SELECT *SUM(R.a)* FROM *R* WHERE *R.d="A"*

ODZC explicitly prefers zero-copy over UVM due to the memory-access granularity with the query-induced sparsity of live tuples, ensuring the GPU fetches only values that survive preceding predicates. Zero-copy enables sector- or cache-line-sized remote loads from host-pinned memory, which matches sparse access patterns and avoids over-fetch, whereas UVM migrates at coarse page granularity and employs multi-page prefetch, inflating transfers under sparsity. Concretely, ODZC's sparsity-aware load reordering delays each column load to just before its first consuming operator so that, as the bit-vector thins, subsequent loads of R.b, R.c, and R.a are triggered only for surviving rows (Fig. 6). Values with anticipated reuse are cached in shared memory, while one-shot values are kept in registers to avoid additional movements. Our microbenchmark (Fig. 7) corroborates this choice: at a selectivity of 0.01% with 4 B column values, the probability of skipping page reads is only 19.4% in ideal $((1-\frac{1}{10^4})^{16\cdot 4KB/4B})$, far below the 99.92% sector-skip rate achievable with zero-copy, since UVM's 4 KB migration and multi-page (e.g., 16-page/64 KB) prefetch amplify data transfer under sparsity.

C. Minimizing Excessive Host-to-GPU Data Movement

Only with ODZC, a GPU inevitably transfers input column values that will be discarded at the beginning of the pipeline. Thus, to further minimize the amount of column values that need to be accessed in the first place, we propose Asynchronous Filter Pushdown (AFP), which leverages CPU-GPU co-processing capability for proactive data reduction.



(a) Collection of selection and join predicates via AFP

(b) AFP-augmented morsel-based pipeline execution model

Fig. 8: Application of Asynchronous Filter Pushdown (AFP) to the query in Fig. 1 and its main pipeline execution

*1) **Opportunity: CPU-Assisted Pre-Filtering**:* Since input columns reside in host memory, we can leverage the host CPU to proactively pre-filter data before GPU access. This enables FaScalSQL to apply established optimization principles like Predicate Pushdown (PP) [120] and Sideways Information Passing (SIP) [100], proven effective in distributed and storage systems for reducing I/O overhead. PP evaluates selective filters early in the pipeline, while SIP propagates filtering information between pipelines (e.g., bloom filters from join build-sides) to eliminate irrelevant rows proactively.

*2) **Challenge: Intra-Pipeline Dependency**:* However, naively applying these filtering principles creates intra-pipeline dependency. When the host CPU processes entire partitions to generate complete filters before transferring to the GPU, it forms a rigid CPU-then-GPU dependency that forces the high-throughput GPU to remain idle. This pipeline stall completely negates the performance benefits of concurrent execution and CPU-GPU co-processing. The core challenge is implementing host-side filtering that enables continuous, incremental data flow to the GPU without hard synchronization points.

*3) **Key Idea: Asynchronous Filter Pushdown (AFP)**:* AFP is designed to break this intra-pipeline dependency. AFP consists of two-step components to maximize data reduction while preserving the efficiency of CPU-GPU concurrency.

First, AFP employs proactive, lightweight predicate collection. As shown in Fig. 8a, it restructures the query plan by pushing down two types of operations to the bottom of the pipeline: 1) simple selection predicates, and 2) lookups into compact bloom filter indexes created from join build-sides and allocates them to the host CPU. AFP avoids complex relational operators, ensuring the workload on the host CPU is minimal and focusing exclusively on discarding input column values.

Second, to remove intra-pipeline dependency, we propose an AFP-augmented morsel-based pipeline execution model. This maximizes the overlapping of host CPU-side AFP operation, the GPU execution, and the host-GPU data transfer by ODZC. As shown in Fig. 8b, the host CPU starts with processing input column values in morsels (i.e., small chunks of tuples [59]). It applies the filtering tasks to a morsel and updates a shared bit-vector. The GPU does not wait for the entire column to be processed. Instead, it begins its pipelined kernel execution on the first set of morsels asynchronously, while the host CPU works ahead on subsequent morsels. With the constantly updated bit-vector, the GPU leverages ODZC to fetch only the sparse, valid input column values from the host memory.

D. Ensuring Fast and Scalable CPU-GPU Co-Processing

Limited and fluctuating CPU availability makes static offloading less effective. We introduce the *Contention-aware Query Optimizer (CQO)*, which selects a subset of AFP tasks to make FaScalSQL robust across various CPU availabilities.

*1) **Opportunity: Fine-Grained Offloading Space**:* AFP decomposes into separable, lightweight filtering tasks, creating a combinatorial offloading space over which an optimizer can pick exactly which filters to execute on the CPU for a given query and system state. This admits a principled placement decision rather than static rules, enabling dynamic CPU–GPU co-processing tuned to predicate selectivities, GPU pipeline structure, and measured CPU availability.

*2) **Challenge: Host CPU-Side Contention**:* We find that the decision to apply AFP involves a trade-off: it reduces GPU work but consumes CPU cycles. In the case of real-world system deployments, 1) numerous various combinations of the host CPU and the GPU having different computational throughput exist, and 2) the host CPU is a shared, contended resource; the host CPU's resource availability is usually contended by co-located applications or background system routines [4], [29], [110]. As shown in Fig. 9, we observe that when available CPU resource decreases from 100% to 6.25%, offloading all AFP operations leads to a 3.48× slowdown in query execution latency. This not only loses its benefit but also can make the engine possibly slower than if AFP were not used at all (up to 4.06× slowdown compared to the ODZC).

*3) **Key Idea: Contention-aware Query Optimizer (CQO)**:* To ensure Asynchronous Filter Pushdown (AFP) remains effective under varying host CPU contention, CQO employs an analytical cost model to find the optimal AFP placement, P^*, that minimizes the end-to-end query latency.

$$P^* = \arg\min_{P \in \mathbb{P}} \max(\text{Cost}_{GPU}, \text{Cost}_{CPU})$$

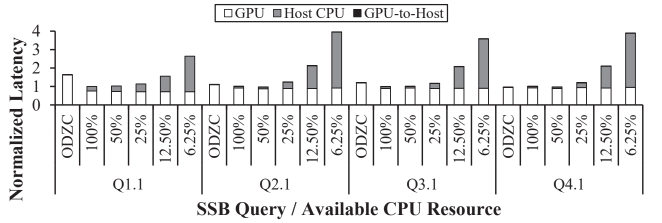


Fig. 9: FaScalSQL's normalized query execution latencies allocating all available AFP operations to the host CPU with varying available CPU resources. ODZC bars represent the baseline without any AFP operations assigned to the host CPU.

Because FaScalSQL executes CPU and GPU tasks in a pipelined manner, the total latency is dominated by the longer-running task. Therefore, our objective function is to minimize $\max(\text{Cost}_{GPU}, \text{Cost}_{CPU})$, reflecting the pipelined execution.

The optimizer finds the best plan P^* by searching all possible AFP placements $\mathbb{P}$. A plan partitions a query's relational operations (RO) into a set for the CPU (AFP_Ops) and the remainder for the GPU (GPU_Ops), except for the selection operations used for s, derived from all combinations of selection ($s \subseteq AFP_s$) and join ($j \subseteq AFP_j$) predicates.

$$\begin{aligned}\mathbb{P} = \{(AFP\_Ops, GPU\_Ops)\ |&AFP\_Ops = s \cup j,\\ &\forall s \subseteq AFP_s,\ \forall j \subseteq AFP_j\\ &GPU\_Ops = RO - s\}\end{aligned}$$

The cost of each operator depends on the fraction of valid tuples it processes (R_i^{AFP} for CPU, R_i^{GPU} for GPU), the cumulative product of prior filter selectivities (λ_k). Our model avoids double-counting filters applied on both CPU and GPU.

$$R_i^{AFP} = \prod_{k=1}^{i-1} \lambda_k^{AFP}$$

$$R_i^{GPU} = \begin{cases} R^{AFP}_{|AFP_{Ops}|+1} & \text{, If } i = 1 \\ R^{GPU}_{i-1} & \text{, If } \exists k : AFP\_Op_k \in AFP\_Ops \text{ and } \\ & AFP\_Op_k \text{ corresponds to } GPU\_Op_{i-1} \\ R^{GPU}_{i-1} \cdot \lambda^{GPU}_{i-1} & \text{, Otherwise} \end{cases}$$

The total costs are calculated as the sum of per-operator costs. $Cost_{CPU}$ is scaled by the available CPU resources ($Available_{CPU}$). $Cost_{GPU}$ includes both computation and the data transfer cost for ODZC. The transfer cost is a function of the probability of accessing a memory sector (P_i^{ODZC}), which depends on the tuple ratio R_i^{GPU}. Operator throughputs (TP) are pre-profiled at system initialization [12], [87], [88], [122].

$$P_i^{ODZC} = 1 - (1 - R_i^{GPU})^{|Sector|/|ColumnValue|}$$

$$Cost_{GPU} = \sum_{j=1}^{|GPU\_Ops|} \#Tuples \cdot \left(\frac{R_j^{GPU}}{TP_j^{GPU}} + \frac{P_j^{ODZC} \cdot Lat_{Sector}}{|Sector|/|ColumnValue|}\right)$$

$$Cost_{CPU} = \sum_{j=1}^{|AFP\_Ops|} \#Tuples \cdot \frac{R_j^{AFP}}{TP_j^{AFP} \cdot Available_{CPU}}$$

CQO requires selectivities derived from existing cardinality estimation methods [21], [31], [32], [55] or one-time profiling [11], [47], [88], [131]. Since CQO runs during the query planning phase, its overhead is negligible.

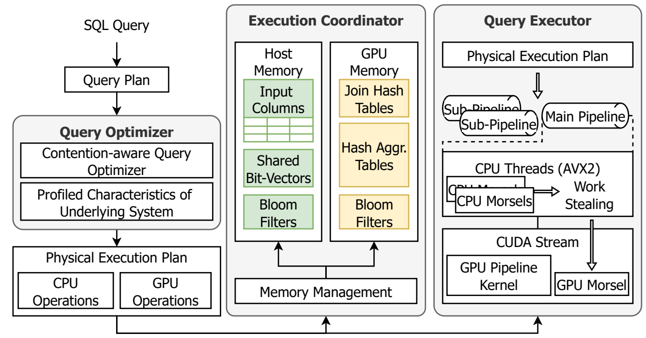


Fig. 10: FaScalSQL system overview

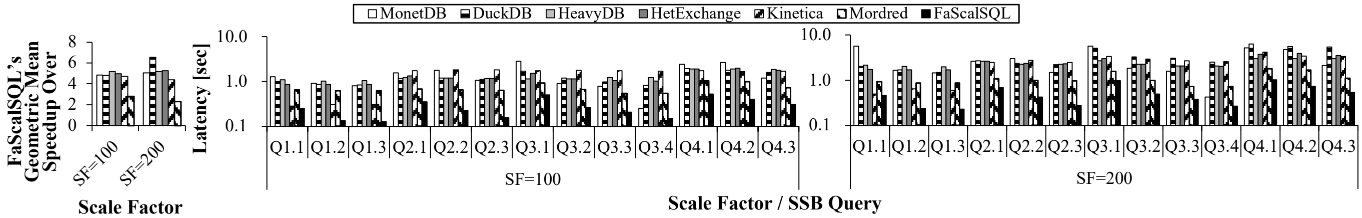
E. Implementation

Fig. 10 illustrates FaScalSQL's architecture, comprising three modules: query optimizer, execution coordinator, and query executor. We extend Crystal [98], an open-source CUDA-based library for analytical query processing, chosen for its modular function library and tile-based execution model which facilitates integration of our hardware-conscious techniques. While built on Crystal, FaScalSQL's principles can also be applied to other GPU query compilers supporting pipeline-driven execution [17], [26], [88].

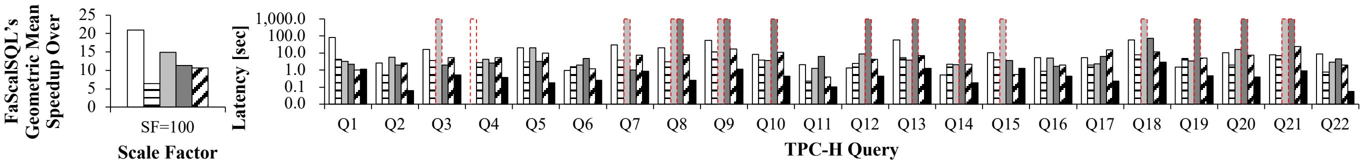
1) Query Optimizer: The query optimizer implements our Contention-aware Query Optimizer (CQO) as a pre-execution pass on the query plan. It extracts bloom filter operations from hash joins, identifies filter predicates suitable for AFP, and evaluates the cost model using pre-profiled hardware characteristics to determine optimal AFP placement. The output physical execution plan specifies which filters execute as AFP. We use initial query plans from Crystal [98] and DogQC [26], both highly optimized for GPU.

2) Execution Coordinator: Execution coordinator is the central component responsible for managing the end-to-end query lifecycle and memory management. Upon receiving a physical execution plan from the optimizer, the coordinator first initializes necessary data structures in host memory, including the shared bit-vector for pruned rows. The coordinator ensures that a zero-copy memory access is enabled by allocating input columns in host-pinned memory with `cudaMallocManaged()` and setting the `cudaMemAdviseSetAccessedBy` flag. For memory management, while input columns reside in host memory, intermediate pipeline outputs (e.g., hash tables) are preferentially allocated in dedicated GPU memory, falling back to zero-copy memory only if GPU memory capacity is exceeded.

3) Query Executor: Query executor includes host-side runtime for AFP and GPU-side runtime for the main pipeline. For each pipeline, the executor spawns a set of CPU threads that dynamically pull tasks from a shared pool of data morsels using a work-stealing approach. Each morsel consists of 4K tuples, a size chosen to ensure the working set fits efficiently within the CPU's Last-Level Cache (LLC). Threads execute filter predicates using AVX2-vectorized operations [91] with bloom filter lookups optimized for LLC residency, updating the shared bit-vector. Following a pipelined model, CPU



(a) SSB queries with scale factors of 100 and 200 (i.e., 3.7× and 7.5× of RTX A4000's memory capacity)



(b) TPC-H queries with a scale factor of 100 (i.e., 6.3× of RTX A4000's memory capacity)

Fig. 11: Query execution latencies of the baseline SQL query engines and FaScalSQL on RTX A4000 + PCIe 4.0 x16. Red dotted bars denote that they cannot be executed due to out-of-memory errors. Note that the y-axes are shown on a log scale.

threads process a batch of morsels to create a GPU morsel, then asynchronously launch the corresponding GPU kernel on a dedicated CUDA stream, allowing for the concurrent processing of subsequent morsels.

V. Evaluation

A. Experimental Setup

System Configuration. We conduct experiments using NVIDIA RTX A4000 [84] and NVIDIA TITAN RTX [83] GPUs attached to the system, which features an AMD Ryzen 3950X CPU [5] with 16 cores and a 64-MB LLC. The system's dual-channel DDR4-2666 memory provides a theoretical bandwidth of 42.6 GB/s. Our primary evaluation is conducted on the RTX A4000, selected for its advanced architecture and support for PCIe 4.0, which offers double the theoretical host-to-device bandwidth compared to PCIe 3.0. For sensitivity analysis (§V-G), we use the TITAN RTX, a comparable high-performance GPU limited to PCIe 3.0. Table II shows the architectural characteristics of RTX A4000 and TITAN RTX.

Baseline SQL Query Engines. We employ six representative query engines as baselines. For a fair comparison of all baseline setups, focusing solely on the in-memory query processing scenario, we evaluate the engines under conditions where all input columns reside in host memory with no swap memory. In addition, the input columns have not been recently accessed or optimized for immediate query execution.

- *MonetDB* [10] is a widely-used CPU-based SQL query engine for in-memory analytical query processing.
- *DuckDB* [20] is a CPU-based SQL query engine optimized for vectorized in-memory analytical query processing.
- *HeavyDB* [37] is a commercial GPU-accelerated SQL query engine optimized for analytical SQL query processing.

TABLE II: Characteristics of the two evaluated GPUs

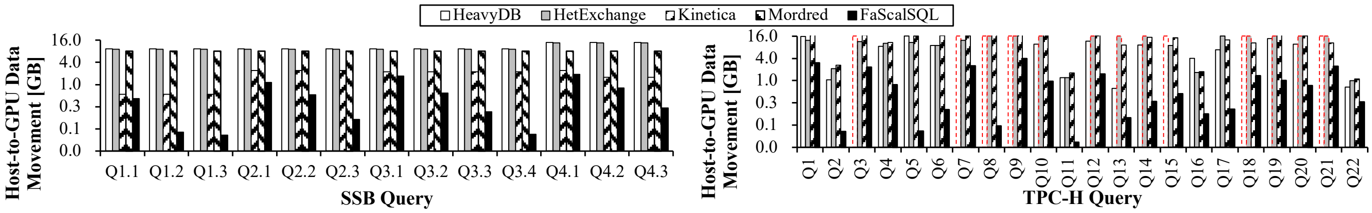
	NVIDIA RTX A4000	NVIDIA TITAN RTX
PCIe Bus	32-GB/s PCIe 4.0 x16	16-GB/s PCIe 3.0 x16
GPU Microarchitecture	Ampere	Turing
GPU Memory Size	16 GB	24 GB
GPU Memory Bandwidth	448 GB/s	672 GB/s

- *HetExchange* [17] is a framework for executing queries using heterogeneous hardware (i.e., a GPU and the host CPU). We reproduce streaming HetExchange by extending Crystal [98], a CUDA-based library, and DogQC [26], an open-source CUDA-based query compiler. We use Crystal for SSB queries and DogQC for TPC-H queries, as Crystal is optimized for SSB workloads while DogQC provides comprehensive support for all TPC-H queries.
- *Kinetica* [50] is a commercial CPU-GPU distributive engine for various domains of analytic queries (e.g., graphs).
- *Mordred* [126] is the state-of-the-art CPU-GPU distributive engine that optimizes SQL query execution with profiling-based input column caching on GPU memory. Note that Mordred is also built with the Crystal library.

Benchmarks. To evaluate FaScalSQL, we use Star Schema Benchmark (SSB) [86], a widely-used collection of 13 SQL queries reflecting real-world data analytics workloads [11], [25], [42], [61], [90], [98], [121], [126], [130]. We employ Scale Factors (SFs) of 100 and 200 ($\sim$60 GB and $\sim$120 GB of total table sizes; 3.7× and 7.5× larger than RTX A4000's GPU memory). We also employ TPC-H benchmark [108] with a SF of 100 ($\sim$100 GB of total table size; 6.3× larger than RTX A4000's GPU memory). We evaluate TPC-H queries except Mordred [126], as it currently lacks support for TPC-H.

B. Fast GPU-Accelerated Query Executions

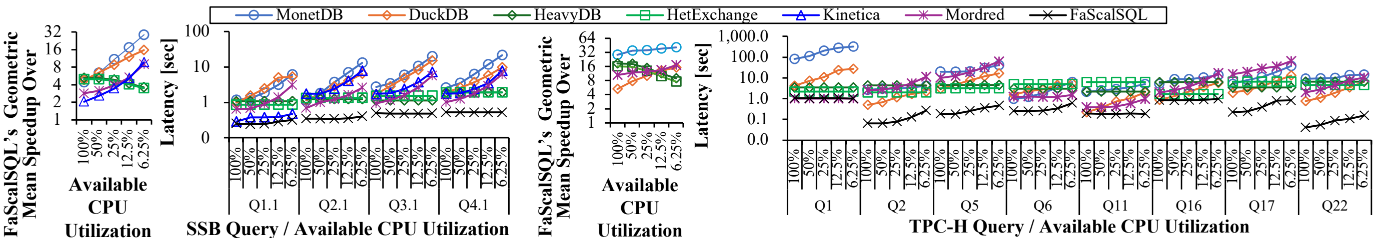
FaScalSQL delivers consistent speedups over all baselines on SSB and TPC-H by aligning with the dynamic progressive filtering feature of analytical queries (Fig. 11). On SSB (SF=100 and 200), FaScalSQL achieves geometric-mean gains of 5.16×, 5.85×, 5.80×, 2.60×, 4.54×, and 2.20× over MonetDB, DuckDB, HeavyDB, HetExchange, Kinetica, and Mordred, respectively, driven by ODZC's sector-level host reads that track the bit-vector's evolving sparsity to curb PCIe traffic, and AFP's predicate/bloom pruning that further reduces sectors ODZC must fetch. Higher-selectivity SSB queries amplify the effects because the valid-tuple ratio and hence the amortized transfer cost; for instance, Q3.4 yields 4.48× over HetExchange. Against Mordred, FaScalSQL's gap widens



(a) SSB queries with a scale factor of 100

(b) TPC-H queries with a scale factor of 100

Fig. 12: Host-to-GPU data transfer size of the baseline GPU-accelerated SQL query engines and FaScalSQL. The red dotted bars denote that they cannot be executed due to out-of-memory errors. Note that the y-axes are shown on a log scale.



(a) SSB queries with a scale factor of 100

(b) TPC-H queries with a scale factor of 100

Fig. 13: FaScalSQL's query execution latencies and speedups over baseline SQL query engines with varying available host CPU resource availability. Note that the y-axes are shown on a log scale.

with increasing SF (achieving geometric mean speedup from 2.15× to 2.24× from SF=100 to 200), consistent with CQO avoiding CPU overload due to misses in static GPU-resident cache. CQO places only the AFP tasks whose CPU cost under $Available_{CPU}$ is amortized by GPU savings, preventing the CPU from becoming the bottleneck and contention.

On TPC-H (SF=100), FaScalSQL achieves geometric-mean speedups of 20.92×, 6.38×, 14.90×, 11.28×, and 10.68× over MonetDB, DuckDB, HeavyDB, HetExchange, and Kinetica, respectively. TPC-H pipelines contain more dependency points and early predicates, which increase the opportunities for AFP to prune before GPU access and for ODZC to delay and sparsify loads; this reduces both compute and the sector-hit probability across more stages than in SSB, compounding the benefit. Unlike engines that pre-stage large inputs in GPU buffers, FaScalSQL keeps inputs host-resident and pulls sectors on demand via ODZC, avoiding GPU buffer pressure and out-of-memory conditions observed in competing setups (HeavyDB, HetExchange) on larger inputs.

C. *Large Reductions in the Data Movement*

To quantify the reduction in host-to-GPU data movement, we compare the data transfer sizes of HeavyDB, HetExchange, Kinetica, Mordred, and FaScalSQL for SSB and TPC-H queries (SF=100). As shown in Fig. 12, FaScalSQL achieves a geometric mean data movement reduction of 39.36× (SSB) and 20.38× (TPC-H) compared to HetExchange. This reduction is not merely an incremental improvement but validates the effectiveness of the alignment with dynamic progressive filtering. ODZC provides the sparsity-aware fine-grained host memory access, and AFP provides significant data movement reduction to guide ODZC to refer to a sparse bit-vector. For queries with high selectivity (e.g., SSB Q3.4), where predicates filter out over 99% of rows [86], this synergy is particularly potent. In contrast, existing engines cannot benefit from such progressive filtering due to static input data placement, reaching host-to-GPU data transfer of gigabyte-scale volumes.

D. *Contention-Aware Query Processing*

To evaluate the robustness of FaScalSQL on host CPU-side contention, we compare query execution latencies using a scale factor of 100 on four representative SSB queries [42] (one from each query group, e.g., Q1.1) and eight TPC-H queries, which were specifically chosen as they could be run on all baseline systems without out-of-memory errors. We simulate varying levels of host CPU-side contention by adjusting available CPU resources from 100% to 6.25%, reducing core count from 16 to 1 [18], [45], [66], [96]. Fig. 13 demonstrates FaScalSQL's consistent performance advantages. As CPU resources decrease, speedups of FaScalSQL over CPU-reliant engines (MonetDB, DuckDB, Kinetica, Mordred) increase, highlighting CQO's adaptive load balancing. It allows FaScalSQL to maintain high performance even with limited CPU resources. Fig. 13a reveals an important difference in CPU resource utilization efficiency. Mordred's latency increases by 845.6% as available CPU resource drops from 100% to 6.25%, FaScalSQL maintains stable latency (133.5% for FaScalSQL). This comes from Mordred's static, fixed-size GPU workload due to the fixed GPU cache, which makes the CPU overloading inevitable. With 6.25% of available CPU resources, FaScalSQL still outperforms CPU-unreliant HeavyDB and HetExchange, showing the robust, scalable CPU-GPU co-processing design of FaScalSQL.

E. *CPU-GPU Load Balance*

To validate the CPU-GPU load balance of FaScalSQL, we break down query execution latencies of four SSB queries (SF=100). For overlapped latencies in the pipeline executions,

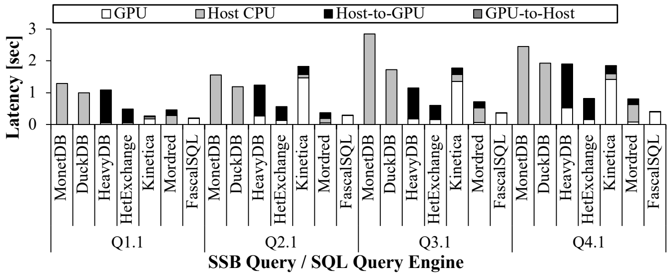


Fig. 14: Latency breakdowns of the baseline SQL query engines and FaScalSQL (SSB, SF=100)

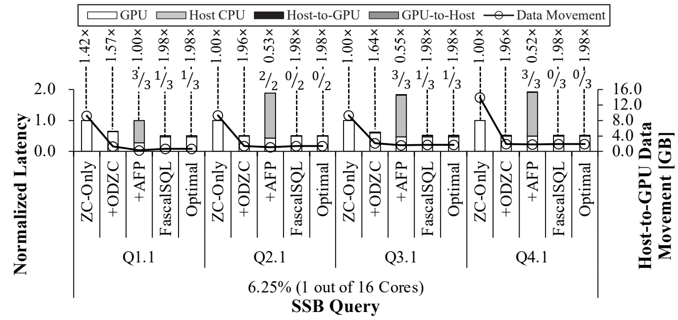
the breakdown prioritizes GPU, host CPU, host-to-GPU, and GPU-to-host latencies. By prioritizing GPU over host CPU and data transfer, we identify slowdowns caused by CPU overload and excessive data movement, showing non-overlapped host CPU and data transfer latencies over GPU execution. Fig. 14 shows that, unlike HeavyDB and HetExchange, which experience excessive host-to-GPU data movement, FaScalSQL significantly minimizes these transfers and overlaps their latencies. FaScalSQL also minimizes the host CPU latencies by AFP-augmented morsel-based asynchronous execution model and CQO; however, Kinetica and Mordred suffer from its severe host CPU reliance and pre-caching overhead. The results show that FaScalSQL's scalability can be achieved without the host CPU being a bottleneck of the GPU.

F. Effectiveness of FaScalSQL's Key Ideas

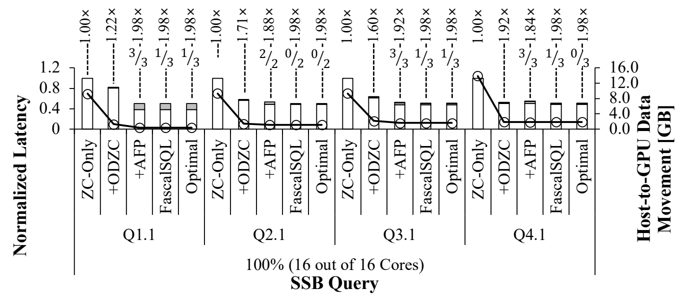
To demonstrate and quantify the effectiveness of each of FaScalSQL's key ideas, we conduct an ablation study. We start with a baseline, ZC-Only, which uses a zero-copy implementation without our optimizations. We then incrementally enable ODZC, AFP, and our full FaScalSQL system, which includes the CQO. We compare these configurations against an Optimal placement determined by offline profiling. Fig. 15 presents the results for four SSB queries (SF=100) under high and low CPU availability, revealing a clear progression. The ZC-Only baseline offers limited benefits, as it still transfers all data. Enabling ODZC's sparsity-aware load reordering (+ODZC) provides a significant speedup by aligning physical access with logical sparsity, which is critical for reducing data movement. Adding AFP (+AFP) further reduces latency by proactively pruning data on the CPU. Finally, our full FaScalSQL system demonstrates the necessity of CQO; under severe CPU contention, it adaptively avoids harmful pushdowns and achieves performance nearly identical to the Optimal configuration, proving its effectiveness of robust and scalable co-processing.

G. Sensitivity Studies

1) PCIe Bus Bandwidth: To assess FaScalSQL's robustness in I/O-constrained environments, we evaluated it on a TITAN RTX GPU with the slower PCIe 3.0 bus. As shown in Fig. 16, FaScalSQL still achieves significant geometric mean speedups, ranging from 2.81× to 6.67× over all baselines. This result demonstrates that FaScalSQL's performance stems from its architectural efficiency in logically reducing data before transfer, rather than relying on high-speed hardware. This makes it highly effective even on bandwidth-limited systems.



(a) Available CPU utilization of 6.25% (1 out of 16 cores)



(b) Available CPU utilization of 100% (16 out of 16 cores)

Fig. 15: Normalized latencies and data transfer sizes by incrementally applying FaScalSQL's key ideas (SSB, SF=100). Latencies are normalized to FaScalSQL. Fraction above each bar denotes the count of placed AFP operations over available operations of the main pipeline from LINEORDER.

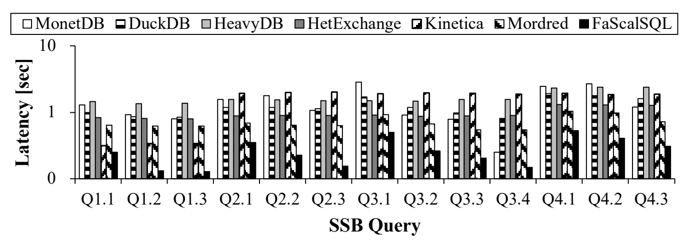


Fig. 16: SSB query execution latencies of the baseline engines and FaScalSQL on NVIDIA TITAN RTX + PCIe 3.0 x16 (SF=100). Note that the y-axis is shown on a log scale.

2) Bloom Filter Size: We evaluated AFP's sensitivity to bloom filter size on SSB (SF=200). Fig. 17 shows that the false positive rate drops to zero at 16 MB, a size that fits within the CPU's LLC. Smaller sizes (e.g., 64 KB) suffer from high false positive rates, while larger sizes offer no further benefit and risk slower lookups due to DRAM access. We therefore use a 16 MB filter to balance effectiveness and cache efficiency.

3) Impact on Data Skew: To evaluate FaScalSQL's robustness against non-uniform data, we compare query execution latencies on the SSB benchmark (SF=100) with varying data skews using a Zipf distribution from 0.0 to 2.0. In Fig. 18, the results show that FaScalSQL maintains stable query execution latency across all levels of data skew. This resilience stems from our key ideas' focus on minimizing data movement

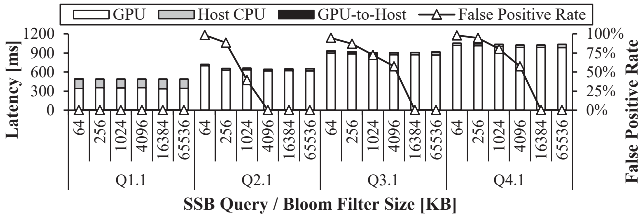


Fig. 17: Impacts of bloom filter size on the SSB query execution latencies and the bloom filters' effectiveness (SF=200)

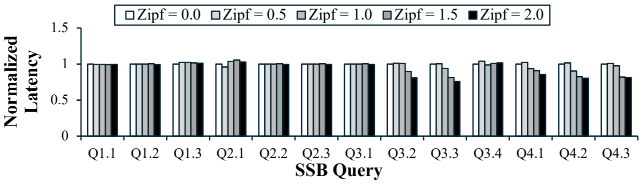


Fig. 18: Query execution latencies varying Zipf factors

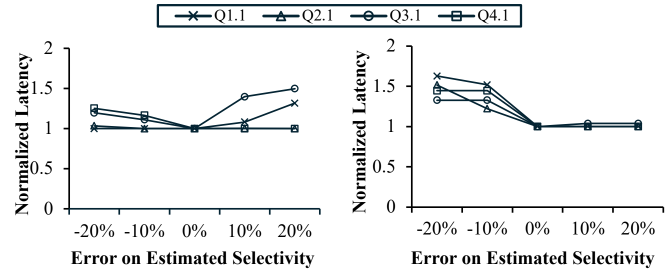
based on query selectivity, rather than being dependent on the underlying data distribution. FaScalSQL effectively prunes rows based on predicate outcomes, regardless of how frequently certain values appear. We observe a slight performance improvement in join-heavy queries under high skew, likely due to better CPU cache locality for frequently joined keys during the hash table build and probe phase [8].

4) Impact of Selectivity Estimation Errors: To evaluate the impact of errors on estimated selectivities, we conduct a latency comparison on four representative SSB queries (SF=100) by injecting intentional errors (-20% to +20%) in the selectivity estimates. These experiments are conducted under two scenarios: severely constrained CPU resources (6.25%) and full availability (100%). In Fig. 19, the results reveal distinct trends depending on CPU availability. Under scarce CPU resources, underestimating selectivity leads FaScalSQL to overestimate the utility of CPU filtering and allocate more work to the constrained CPUs, which in turn increases overall latency. Conversely, under sufficient CPU resources, a V-shaped latency curve emerges. Overestimating selectivity leads CQO to deem CPU filtering inefficient and thus disallow work. Underestimating selectivity leads CQO to allocate too much work to the CPUs, which in turn increases latency due to pipeline imbalance. However, the results show that FaScalSQL can achieve meaningful data reduction and fast performance even using only GPUs, demonstrating robust operation even when there are errors in the given selectivity.

VI. Discussions

A. Overhead of Contention-Aware Query Optimizer (CQO)

In our evaluation, we focus solely on execution latencies to align with baseline engines and isolate the impact of our optimizations. However, for practical, real-world GPU-accelerated query deployment, pre-execution costs, including the overhead of CQO, should be considered. We measure the pre-execution latencies using JIT compilation frameworks for GPU-accelerated SQL queries (i.e., rNdN [52], DogQC [26], Pyper [87], and Themis [40]) on SSB and TPC-H queries. On average, code generation required 14ms, and NVCC compilation needed 10ms per query. In addition, adding CQO



(a) Available CPU utilization of 6.25% (1 out of 16 cores)

(b) Available CPU utilization of 100% (16 out of 16 cores)

Fig. 19: Latency comparison with varying errors to estimated selectivities and available CPU utilizations.

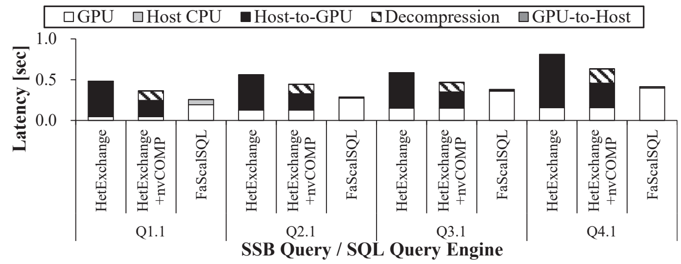


Fig. 20: SSB query execution latency comparison of HetExchange, HetExchange with nvCOMP, and FaScalSQL

to their JIT compilation process incurred only 1ms to 2ms extra in code generation, representing negligible overhead (under 10% of pre-execution time), which we expect to remain insignificant across diverse workloads.

B. Impact of Data Compression

While we focus on reducing data movement for in-memory columnar storage systems [10], [38], [63], [102], data compression [2], [79], [99] is an alternative approach, especially used for storage-backed systems [6], [106]. To evaluate the impact of compression, we conduct an additional experiment comparing FaScalSQL against a streaming HetExchange, augmented with NVIDIA's high-performance nvCOMP compression library [85]. While nvCOMP reduces transfer latency by 46% on average (Fig. 20), FaScalSQL achieves a geometric mean speedup of 1.42× over HetExchange+nvCOMP. It shows that compression itself is insufficient to reach 39× reduction of FaScalSQL and suffers from decompress overhead.

C. Limitations

FaScalSQL is designed for out-of-memory tables residing in host memory within a single GPU environment. While this focus enables deep optimization within our target domain, it introduces a few inherent limitations as follows.

First, single-GPU systems are limited by a single GPU's memory capacity and PCIe bottleneck, which become more pronounced as query complexity increases. While we believe substantial optimization opportunities remain in single-GPU environments, the hardware constraints can limit FaScalSQL's applicability to highly large-scale analytics scenarios demanding multi-node processing and resources.

Second, FaScalSQL's key ideas, particularly ODZC, are optimized for direct GPU access to host memory via PCIe bus. This sub-cache-line granularity data fetching is limited to data which is entirely loaded in the host memory. For storage-resident tables on SSDs or traditional disks, GPUs cannot perform the fine-grained, on-demand access that makes ODZC effective, as storage I/O operates at coarser granularities (typically 4KB pages) and involves higher latencies that would negate the benefits of our sparsity-aware access patterns.

Third, FaScalSQL targets read-intensive analytical queries, and thus the primary challenge lies in efficiently filtering and processing large data. While FaScalSQL is compatible with dataset updates using its on-demand data fetching, it does not fully address the complexities of OLTP or HTAP scenarios that would require additional synchronization mechanisms.

D. Future Work

Scaling Out to Custom Accelerators and Multiple GPUs. Using multiple GPUs is a plausible scale-out solution for processing large tables by using the aggregated GPU memory capacity [89], [114], [124], [129]. The key ideas of FaScalSQL can also be effective in multi-GPU environments, as they are not limited to single-GPU environments. Our key ideas about matching data access patterns to query selectivity can be adapted to FPGAs [22], [64], [67], [76], [112] and custom ASICs [7], [118], as they face the same challenges of limited on-chip memory and expensive data movement via PCIe bus.
Extending to Storage-Resident Tables. A promising direction involves exploiting near-storage processing capabilities that perform AFP-like filtering and operations closer to the tables stored in storage devices. In addition, extending our lightweight indexing approach beyond bloom filters (e.g., B-trees [97], compressed indexes) could enable more sophisticated predicate evaluation at the storage level.
Beyond Analytical SQL Queries. As FaScalSQL caches up-to-date input column values from host to GPU memory upon each SQL query execution, we expect FaScalSQL can be seamlessly integrated with CPU-based transactional databases. Future work could explore versioned on-demand access that allows analytical queries to access consistent snapshots while transactions proceed, and adaptive CPU-GPU workload distribution that shifts between transactional CPU processing and analytical GPU processing based on workload characteristics.

VII. Related Work

GPU-Initiated Host-to-GPU Data Transfer. The concept of allowing GPUs to directly initiate the data transfer from host memory, often leveraging mechanisms (i.e., zero-copy and UVM), has been explored to mitigate the host-to-GPU interconnect bottleneck. Raza et al. [92], for instance, demonstrated the potential of GPU-initiated lazy transfers to reduce data movement, particularly for selective queries, by allowing GPU kernels to pull necessary data on demand. Such techniques have also been utilized in various domains like deep learning [46] and large-scale graph processing [41], [74], [111]. Building upon these pioneering efforts which established the viability of GPU-initiated data transfer, FaScalSQL introduces a new on-demand CPU-GPU co-processing engine specifically designed to maximize its effectiveness for the dynamic filtering nature of analytical SQL query processing.
Predicate Pushdown and Sideways Information Passing. Predicate pushdown and sideways information passing are well-established principles for reducing data movement in various domains [6], [30], [44], [56], [62], [100], [113], [120], [122], [128]. FaScalSQL's novelty lies in adapting these principles into an asynchronous, morsel-based pipeline, which breaks the CPU-GPU dependency stalls that would otherwise cripple a naive co-processing implementation.
GPU-Accelerated Query Execution. A large body of work has focused on optimizing SQL query execution for data that fully resides in GPU memory [15], [19], [24], [26], [34], [36], [39], [40], [42], [70], [87], [88], [98], [110], [115], [125], [130], [132]. Techniques include optimizing compute-intensive operations like joins [48], [53], [69], [71], [89], [94], [95], [101], [103], [105], [107] and using JIT compilation with kernel fusion to mitigate thread divergence and materialization overheads [25], [26], [36], [40], [87], [88], [98], [116], [117]. While optimizing execution on GPU-resident data, FaScalSQL is complementary to and orthogonal to their key ideas.
Storage-Backed GPU Query Execution. Several engines optimize data transfer directly from storage. HippogriffDB [65], HetCache [80], and GOLAP [9] focus on maximizing storage-to-GPU PCIe bandwidth, for instance by bypassing host memory or using on-the-fly decompression. However, they still rely on transferring coarse-grained data blocks. In contrast, FaScalSQL logically reduces (AFP) and fine-grains (ODZC) the data access itself, minimizing the volume of data that needs to be staged from storage in the first place.

VIII. Conclusion

We proposed *FaScalSQL*, a new GPU-accelerated SQL query engine that maximizes the GPU utilization when processing analytical queries involving out-of-memory tables. Using its three key ideas, namely On-Demand Zero-copy Caching (ODZC), Asynchronous Filter Pushdown (AFP), and a Contention-aware Query Optimizer (CQO), FaScalSQL aligns query processing with the dynamic progressive filtering of analytical queries, and exploits both GPU-initiated data transfer and CPU-GPU co-processing capability. Our evaluation using SSB and TPC-H queries shows that FaScalSQL outperforms the existing GPU-accelerated engines, even under host CPU-side contention and limited PCIe bandwidth.

Acknowledgment

This work was partly supported by the National Research Foundation of Korea (NRF) grant (RS-2025-00513906) and the Institute of Information & communications Technology Planning & Evaluation (IITP) grants (RS-2020-II201361, RS-2024-00395134, RS-2025-02217106, RS-2025-02304554) funded by the Korea government (MSIT). Youngsok Kim is the corresponding author of this paper.

AI-Generated Content Acknowledgement

In this paper, ChatGPT was employed only for language polishing and grammar checks. All core research ideas, designs, implementation, experimental analyses, and conclusions were entirely conceived and developed by the authors.

References

[1] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[2] Azim Afroozeh, Lotte Felius, and Peter Boncz. Accelerating GPU Data Processing using FastLanes Compression. In *Proc. 20th International Workshop on Data Management on New Hardware (DaMoN)*, 2024.

[3] Byungmin Ahn, Jaehun Jang, Hanbyeul Na, Mankeun Seo, Hongrak Son, and Yong Ho Song. AI Accelerator Embedded Computational Storage for Large-Scale DNN Models. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2022.

[4] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Accelerating critical OS services in virtualized systems with flexible micro-sliced cores. In *Proc. Thirteenth EuroSys Conference (EuroSys)*, 2018.

[5] AMD. AMD RYZEN 9 3950X, 2019. https://www.amd.com/en/product/8486.

[6] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proc. 2015 ACM SIGMOD international conference on management of data (SIGMOD)*, 2015.

[7] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep Agrawal, Arun Raghavan, Venkatanathan Varadarajan, et al. Rapid: In-memory analytical query processing engine with extreme performance per watt. In *Proc. 2018 International Conference on Management of Data (SIGMOD)*, 2018.

[8] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27, 2014.

[9] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM on Management of Data*, 2, 2024.

[10] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51, 2008.

[11] Sebastian Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14, 2014.

[12] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38, 2013.

[13] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proc. 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[14] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proc. VLDB Endowment (PVLDB)*, 7, 2014.

[15] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endowment (PVLDB)*, 17, 2023.

[16] Donald D Chamberlin and Raymond F Boyce. SEQUEL: A structured English query language. In *Proc. 1974 ACM SIGFIDET workshop on Data description, access and control*, 1974.

[17] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endowment (PVLDB)*, 12, 2019.

[18] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proc. 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[19] Yangshen Deng, Shiwen Chen, Zhaoyang Hong, and Bo Tang. How Does Software Prefetching Work on GPU Query Processing? In *Proc. 20th International Workshop on Data Management on New Hardware*, 2024.

[20] DuckDB. Execution Format, 2025. https://duckdb.org/docs/internals/vector.html.

[21] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endowment (PVLDB)*, 12, 2019.

[22] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal (VLDBJ)*, 29, 2020.

[23] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. GPUQP: query co-processing using graphics processors. In *Proc. 2007 ACM SIGMOD international conference on Management of data*, 2007.

[24] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. NestGPU: Nested query processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021.

[25] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proc. 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018.

[26] Henning Funke and Jens Teubner. Data-parallel query processing on non-uniform data. *Proc. VLDB Endowment (PVLDB)*, 13, 2020.

[27] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proc. 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[28] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proc. 22nd ACM Internet Measurement Conference (IMC)*, 2022.

[29] Chris Gregg, Jeff Brantley, and Kim Hazelwood. Contention-aware scheduling of parallel code for heterogeneous systems. In *2nd USENIX workshop on hot topics in parallelism, HotPar, Berkeley, CA*, 2010.

[30] Tim Gubner, Diego Tomé, Harald Lang, and Peter Boncz. Fluid co-processing: Gpu bloom-filters for cpu joins. In *Proc. 15th International Workshop on Data Management on New Hardware (DaMoN)*, 2019.

[31] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endowment (PVLDB)*, 2021.

[32] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proc. VLDB Endowment (PVLDB)*, 11, 2017.

[33] Mark Harris. Unified Memory for CUDA Beginners, 2017. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[34] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Co-Processing on Graphics Processors. *ACM Transactions on Database Systems*, 2009.

[35] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Joins on Graphics Processors. In *Proc. 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[36] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endowment (PVLDB)*, 15(11), 2022.

[37] HEAVY.AI. HeavyDB, 2022. https://www.heavy.ai/product/heavydb.

[38] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endowment (PVLDB)*, 6, 2013.

[39] Henneberg, Justus and Schuhknecht, Felix. Rtindex: Exploiting hardware-accelerated gpu raytracing for database indexing. 16, 2023.

[40] Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S Bhowmick, and Wook-Shin Han. Themis: A GPU-accelerated Relational Query Execution Engine. *Proc. VLDB Endowment (PVLDB)*, 18(2), 2025.

[41] Lin Hu, Lei Zou, and M Tamer Özsu. GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023.

[42] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. TCUDB: Accelerating Database with Tensor Processors. In *Proc. 2022 International Conference on Management of Data (SIGMOD)*, 2022.

[43] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven code execution for distributed deep learning. In *Proc. 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[44] Zachary G Ives and Nicholas E Taylor. Sideways information passing for push-style query processing. In *Proc. 2008 IEEE 24th International Conference on Data Engineering (ICDE)*, 2008.

[45] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proc. 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[46] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proc. Eighteenth European Conference on Computer Systems (EuroSys)*, 2023.

[47] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endowment (PVLDB)*, 2017.

[48] Tomas Karnagel, René Müller, and Guy M Lohman. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@ VLDB*, 8, 2015.

[49] Gwangsun Kim, Changhyun Kim, Jiyun Jeong, Mike Parker, and John Kim. Contention-based congestion management in large-scale networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[50] Kinetica. Kinetica - The Real-Time Database, 2025. https://www.kinetica.com/.

[51] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proc. 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[52] Alexander Krolik, Clark Verbrugge, and Laurie Hendren. rndn: Fast query compilation for nvidia gpus. *ACM Transactions on Architecture and Code Optimization*, 20, 2023.

[53] Artem Kroviakov, Petr Kurapov, Christoph Anneser, and Jana Giceva. Heterogeneous intra-pipeline device-parallel aggregations. In *Proc. 20th International Workshop on Data Management on New Hardware*, 2024.

[54] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. Accelerating multi-way joins on the GPU. *The VLDB Journal (VLDBJ)*, 31, 2022.

[55] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6, 2021.

[56] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Computer architecture letters*, 19, 2020.

[57] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endowment (PVLDB)*, 14, 2021.

[58] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proc. 37th International Symposium on Computer Architecture (ISCA)*, 2010.

[59] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proc. 2014 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2014.

[60] Feifei Li. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proc. VLDB Endowment (PVLDB)*, 12, 2019.

[61] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endowment (PVLDB)*, 9, 2016.

[62] Yinan Li, Jianan Lu, and Badrish Chandramouli. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM on Management of Data (PACMMOD)*, 1, 2023.

[63] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. Data formats in analytical DBMSs: performance trade-offs and future directions. *The VLDB Journal*, 34, 2025.

[64] Ke Liu, Haonan Tong, Zhongxiang Sun, Zhixin Ren, Guangkui Huang, Hongyin Zhu, Luyang Liu, Qunyang Lin, and Chuang Zhang. Integrating FPGA-based hardware acceleration with relational databases. *Parallel Computing*, 119, 2024.

[65] Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. Hippogriff: Efficiently moving data in heterogeneous computing systems. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016.

[66] Yuxuan Liu, Tianqiang Xu, Zeyu Mi, Zhichao Hua, Binyu Zang, and Haibo Chen. CPS: A Cooperative Para-virtualized Scheduling Framework for Manycore Machines. In *Proc. 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[67] Alec Lu, Jahanvi Narendra Agrawal, and Zhenman Fang. SQL2FPGA: Automated Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms. *ACM Transactions on Reconfigurable Technology and Systems*, 17, 2024.

[68] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proc. 2020 International Conference on Management of Data (SIGMOD)*, 2020.

[69] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Triton join: efficiently scaling to a large join state on GPUs with fast interconnects. In *Proc. 2022 International Conference on Management of Data (SIGMOD)*, 2022.

[70] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. RTScan: Efficient Scan with Ray Tracing Cores. *Proc. VLDB Endowment (PVLDB)*, 17, 2024.

[71] Vasilis Mageirakos, Riccardo Mancini, Srinivas Karthik, Bikash Chandra, and Anastasia Ailamaki. Efficient GPU-accelerated Join Optimization for Complex Queries. In *Proc. 2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022.

[72] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. Dremel: A decade of interactive SQL analysis at web scale. *Proc. VLDB Endowment (PVLDB)*, 13, 2020.

[73] META. META Data Centers, 2022. https://datacenters.fb.com/.

[74] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen mei Hwu. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs. *Proc. VLDB Endowment (PVLDB) (PVLDB)*, 14(2), 2020.

[75] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endowment (PVLDB)*, 2021.

[76] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. Resource-efficient database query processing on FPGAs. In *Proc. 17th International Workshop on Data Management on New Hardware (DaMoN)*, 2021.

[77] Vivek Narasayya and Surajit Chaudhuri. Multi-tenant cloud data services: State-of-the-art, challenges and opportunities. In *Proc. 2022 International Conference on Management of Data (SIGMOD)*, 2022.

[78] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endowment (PVLDB)*, 4, 2011.

[79] Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. 2025.

[80] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics. In *Proc. 2023 Conference on Innovative Data Systems Research (CIDR)*, 2023.

[81] John Nickolls and William J. Dally. The GPU Computing Era. *IEEE Micro*, 30, 2010.

[82] NVIDIA. CUDA C++ Best Practices Guide, 2014. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#zero-copy.

[83] NVIDIA. NVIDIA TITAN RTX Datasheet, 2019. https://www.nvidia.com/content/dam/en-zz/Solutions/titan/documents/titan-rtx-for-creators-us-nvidia-1011126-r6-web.pdf.

[84] NVIDIA. NVIDIA Ampere GA102 GPU Architecture, 2021. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[85] NVIDIA. NVIDIA nvCOMP, 2024. https://github.com/NVIDIA/nvcomp.

[86] Pat O'Neil, Betty O'Neil, and Xuedong Chen. Star Schema Benchmark – Revision 3, 2009. https://www.cs.umb.edu/~poneil/StarSchemaB.pdf.
[87] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endowment (PVLDB)*, 14, 2020.
[88] Johns Paul, Jiong He, and Bingsheng He. GPL: A GPU-based pipelined query processing engine. In *Proc. 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.
[89] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. MG-Join: A scalable join for massively parallel multi-GPU architectures. In *Proc. 2021 International Conference on Management of Data (SIGMOD)*, 2021.
[90] Ben Perach, Ronny Ronen, and Shahar Kvatinsky. Accelerating Relational Database Analytical Processing with Bulk-Bitwise Processing-in-Memory. In *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*. IEEE, 2023.
[91] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.
[92] Syed Mohammad Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos Christos Anadiotis, and Anastasia Ailamaki. GPU-accelerated data management under the test of time. In *Online Proc. 10th Conference on Innovative Data Systems Research (CIDR)*, 2020.
[93] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)*, 55, 2022.
[94] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Performance analysis and automatic tuning of hash aggregation on GPUs. In *Proc. 15th International Workshop on Data Management on New Hardware (DaMoN)*, 2019.
[95] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proc. 29th international conference on scientific and statistical database management*, 2017.
[96] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. CC-NIC: a Cache-Coherent Interface to the NIC. In *Proc. 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.
[97] Amirhesam Shahvarani and Hans-Arno Jacobsen. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *Proc. 2016 International Conference on Management of Data (SIGMOD)*, 2016.
[98] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proc. 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.
[99] Anil Shanbhag, Bobbi W Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in GPU. In *Proc. 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2022.
[100] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013.
[101] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.
[102] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018.
[103] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. An empirical performance comparison between matrix multiplication join and hash join on GPUs. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, 2023.
[104] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM on Management of Data*, 1, 2023.
[105] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. Distributed GPU joins on fast RDMA-capable networks. *Proc. ACM on Management of Data*, 1, 2023.
[106] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endowment (PVLDB)*, 2, 2009.
[107] Diego G Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A Boncz. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *ADMS@ VLDB*, 2018.
[108] Transaction Processing Performance Council (TPC). TPC-H Benchmark, 2025. https://www.tpc.org/tpch/.
[109] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. Near-data processing in database systems on native computational storage under HTAP workloads. *Proc. VLDB Endowment (PVLDB)*, 15, 2022.
[110] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proc. VLDB Endowment (PVLDB)*, 7, 2014.
[111] Qiange Wang, Xin Ai, Yanfeng Zhang, Jing Chen, and Ge Yu. HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023.
[112] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. Column-oriented database acceleration using FPGAs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019.
[113] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *Proc. VLDB Endowment (PVLDB)*, 7, 2014.
[114] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. Terabyte-Scale Analytics in the Blink of an Eye. *arXiv preprint arXiv:2506.09226*, 2025.
[115] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM on Management of Data*, 3, 2025.
[116] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proc. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.
[117] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.
[118] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. Q100: the architecture and design of a database processing unit. In *Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
[119] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proc. 19th International Middleware Conference (Middleware)*, 2018.
[120] Cong Yan, Yin Lin, and Yeye He. Predicate pushdown for data science pipelines. *Proc. ACM on Management of Data (PACMMOD)*, 1, 2023.
[121] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS. *Proc. VLDB Endowment (PVLDB)*, 14, 2021.
[122] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *VLDB Journal (VLDBJ)*, 33, 2024.
[123] Sui Yi, Li Yuhe, and Wang Yu. Cloud computing architecture design of database resource pool based on cloud computing. In *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 2018.
[124] Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endowment (PVLDB)*, 17, 2024.
[125] Bobbi Yogatama, Brandon Miller, Yunsong Wang, Graham Markall, Jacob Hemstad, Gregory Kimball, and Xiangyao Yu. Accelerating user-defined aggregate functions (UDAF) with block-wide execution and JIT compilation on GPUs. In *Proc. 19th International Workshop on Data Management on New Hardware*, 2023.

[126] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proc. VLDB Endowment (PVLDB)*, 15, 2022.
[127] Tao Yu, Jie Qiu, Berthold Reinwald, Lei Zhi, Qirong Wang, and Ning Wang. ntelligent database placement in cloud environment. In *2012 IEEE 19th International Conference on Web Services (ICWS)*, 2012.
[128] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.
[129] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. *Proc. VLDB Endowment (PVLDB)*, 18(4), 2025.
[130] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endowment (PVLDB)*, 6, 2013.
[131] Kai Zhang, Jiayu Hu, Bingsheng He, and Bei Hua. DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017.
[132] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proc. VLDB Endowment (PVLDB)*, 6, 2013.