

TQP++: Bridging ML Compilers and Analytical Query Processing on GPUs

Wei Cui
Microsoft, CA
weicu@microsoft.com

Peng Cheng
Microsoft, US
pengc@microsoft.com

Carlo Curino
Microsoft, US
carlo.curino@microsoft.com

Rathijit Sen
Microsoft, US
rathijit.sen@microsoft.com

Matteo Interlandi
Microsoft, US
matteo.interlandi@microsoft.com

ABSTRACT

The convergence of unified, cloud-native Lakehouse platforms such as Microsoft Fabric, and the widespread deployment of AI-optimized hardware in datacenters is driving the rise of GPU-based analytical engines. However, building a GPU query engine that is both competitive with hand-tuned implementations and portable across vendors (NVIDIA, AMD, custom silicon) remains an open challenge, as existing systems either lock into a single vendor’s toolchain or sacrifice performance for generality. In this paper, we present TQP++, an ML-compiler-native analytical query processor that repurposes ML compiler infrastructure to close this gap, achieving high performance and hardware portability from a single codebase. TQP++ integrates the Antares compilation framework with tiered GPU resource scheduling for SQL operators, a map-reduce-oriented fusion schema that eliminates intermediate materializations, and a multi-gated execution graph that adapts operator algorithms to runtime data characteristics. On TPC-H SF 100, TQP++ executes all 22 queries in 1.1 second on an A100 (7× faster than HeavyDB, 15× faster than CPU baselines) and under 0.7 seconds on H100 and MI300, while targeting 9 devices across 3 vendors—including Xbox—without code changes. To our knowledge, this is the most hardware-diverse GPU query processor reported to date, while achieving sub-second TPC-H SF100 on a single GPU.

PVLDB Reference Format:

Wei Cui, Peng Cheng, Carlo Curino, Rathijit Sen, and Matteo Interlandi. TQP++: Bridging ML Compilers and Analytical Query Processing on GPUs. PVLDB, 19(12): 4116 - 4129, 2026.
doi:10.14778/3827998.3828020

1 INTRODUCTION

The Cloud Computing industry is transitioning from CPU-only to heterogeneous hardware, driven by the growing demand for compute in AI [12], game streaming [41] and other workloads, leading to massive investments in accelerators [38]. Moreover, these accelerators are more widely available now. For instance, as AI workloads transition to newer GPU generations, previous generations become underutilized and this is factored in on GPU prices. For example, on-demand H100 prices have dropped 62.5% from 2023 to

around 2025 [67]. Similarly, gaming services like xCloud [41] have specific utilization patterns, whereby GPU devices are mostly idle during the night. We believe that this ongoing transition presents an opportunity to rekindle query processing.

The shift towards hardware acceleration for analytics is particularly relevant in the context of modern Lakehouse architectures. For instance, Microsoft Fabric [42] unifies analytics services on a single SaaS platform where data is stored in open columnar formats (e.g., Parquet) in a shared storage layer. In this context, we have recently introduced CoddSpeed [22]: A GPU-accelerated query processor integrated natively with our Data Warehousing offering [4]. In this paper, we describe some early research we used to bootstrap CoddSpeed, and share with the community our insights and some of the lessons we learned when moving from research to production. **TQP.** Our first stab at running SQL analytical queries on GPU was TQP: the first ML-compiler-native Python-based GPU analytical database [20]. Differently than what was attempted in the past, with TQP we took the bold decision to build a query processor not directly on vendor-specific technology (e.g., CUDA [46] or ROCm [6]) nor on cross hardware language stacks (e.g., Vulkan [28] or OpenCL [1]) to avoid lock-in and poor portability [28]. Instead, we directly implemented SQL operators in PyTorch [51]. TQP takes a SQL query as input and uses binding to external query engines/databases such as Apache Spark [70] and SQL Server [40] for generating an optimized physical plan. TQP then compiles the physical plan into an executable plan composed of PyTorch [51] operators. In summary, TQP compiles input queries into PyTorch programs, which, as such, can be run on any hardware supported by PyTorch (currently NVIDIA and AMD GPUs, Apple Silicon, Google TPUs [23] through XLA [2], and CPUs). While developing TQP, we found several limitations regarding both performance and portability. PyTorch operators are mainly developed for ML workloads, and expressing relational operators using exclusively ML operators limits performance. When we started extending TQP with more efficient SQL-specific custom operators (e.g., hash join or group-by) we soon found ourselves with having to re-implement each of them for every architecture we wanted to support—this is exactly the $N \times M$ problem we wanted to avoid in the first place!

Our Approach. To solve the above dilemma of achieving performance, while not giving up portability, we decided to leverage a compiler stack surfacing a high-level language, a cross-hardware IR, an optimizer, and a code generator. The twist is that instead of building our own compiler stack, we repurposed Antares, a state-of-the-art ML compilation framework [39]. Our bet is that by building

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828020

an OLAP query processor on top of state-of-the-art AI compilers, we could ride the wave of innovation happening in the AI space also for analytical database workloads. Antares allows us not only to add efficient and portable SQL-specific custom operators to PyTorch and TQP, but thanks to its compiler infrastructure, we can fuse multiple PyTorch/Antares operators into single, more efficient, kernels. This is akin to compiler-based databases [44], but with the added benefit that we can easily switch between compiled (Antares with just-in-time fusion or JTF) and interpreted execution (Antares with ahead-of-time compiled custom operators or ATF).

The key challenge with this approach is that ML operations are typically *dense* (e.g., matrix multiplication), while SQL operations (such as hash joins and group-by aggregates) are *sparse*. To reconcile this duality, in this work we have added new algorithmic and infrastructure innovations to Antares, including *tiered GPU resource scheduling* for SQL operators, a *map-reduce-oriented fusion schema*, and a *multi-gated execution graph* that adapts to data characteristics at runtime. TQP++ is the name of this new system integrating TQP with a compiler backend based on Antares. TQP++ allows not only to achieve incredible performance, but also unmatched flexibility, e.g., we can target hardware from different vendors (e.g., NVIDIA and AMD GPUs), APUs (e.g., Xbox [15]), and eventually CPU and other accelerators (not explored in this paper).

In our experiments, we compare TQP++ against research and production state-of-the-art GPU engines (i.e., Crystal [57] and HeavyDB [21]), as well as two CPU baselines (i.e., DuckDB [53] and SQL Server [40]). To prove TQP++’s portability, we run experiments over a diverse set of GPUs: P100, V100, A100, H100 and GH200 from NVIDIA; MI50, MI200 and MI300 from AMD; and Xbox. As far as we know, not only is this the first paper comparing such a large set of GPU hardware, but we are also reporting the best public performance for TPC-H on a single GPU. This is more than an order of magnitude faster than what commercial CPU/GPU baselines achieve today on a similar hardware. Our contributions are:

- We present TQP++, a query processor that leverages ML compiler infrastructure to achieve both state-of-the-art performance and hardware portability for analytical SQL workloads.
- We introduce tiered GPU resource scheduling techniques that exploit hierarchical memory for three key SQL operators: LIKE, group-by aggregation, and hash join.
- We design a map-reduce-oriented operator fusion scheme and a multi-gated execution graph with runtime switch nodes that adapt algorithmic selection to data characteristics on-the-fly.
- We provide a comprehensive cross-hardware GPU evaluation, spanning 9 devices from 3 vendors, demonstrating sub-second TPC-H SF 100 performance on a single GPU.

The remainder of the paper is organized as follows. Section 2 provides background on GPUs, ML compilers, and TQP. Section 3 discusses the key challenges. Section 4 introduces the Antares IR and shows how it can be used to express SQL operators. Section 5 presents the design of TQP++’s tiered resource scheduling. Section 6 describes the compilation modes and fusion strategies, while Section 7 covers code generation. Section 9 presents our experimental evaluation. Section 10 discusses our lessons learned. Section 11 surveys related work, and Section 12 concludes.

2 BACKGROUND

2.1 GPUs and GPU Databases

Modern GPUs offer massive parallelism, high memory bandwidth, and memory hierarchies that align well with the data-parallel nature of analytical database workloads. Execution is organized into *warps* (32 threads on NVIDIA GPUs, 64 threads on AMD) that execute in lockstep; threads within a warp can communicate through fast warp-level primitives (e.g., shuffle, ballot) and can synchronize via barriers. Warps are grouped into *thread blocks* which are the unit of scheduling on a Streaming Multiprocessor (SM). A GPU *kernel* launch specifies a grid of thread blocks; the hardware scheduler maps blocks onto available SMs, and multiple blocks may execute concurrently on the same SM if resources permit.

Each level of the compute hierarchy is paired with a corresponding tier of memory, with increasing capacity and latency: (1) *thread-private registers* (TPR), which offer the lowest latency but are limited to a few hundred per thread; (2) *block-shared memory* (BSM), a fast on-chip scratchpad visible to all threads within a thread block, typically ranging from 48–228 KB per block depending on the architecture; and (3) *global HBM* (High Bandwidth Memory), the device’s main memory, which provides the largest capacity (up to 192 GB on modern devices) but incurs significantly higher access latency. GPUs achieve their highest throughput when memory accesses are *coalesced*—i.e., consecutive threads access consecutive memory addresses—and when data is staged in registers and shared memory to minimize costly HBM accesses. Maximizing the number of active warps further helps hide memory latency. These principles are central to TQP++’s tiered resource scheduling (Section 5).

Prior efforts have demonstrated the potential of GPU acceleration in several settings, from research [21, 57, 58] to production [16, 21, 22]. Yet, a comprehensive optimization framework that systematically exploits GPU-specific features—such as fine-grained data placement across cache levels, warp-level primitives, and coalesced global memory access—while remaining portable across GPU vendors, remains largely unexplored. This paper addresses this gap through a compiler-based approach that co-designs data access patterns with the GPU memory hierarchy.

2.2 Tensor Query Processor

TQP [9, 20] accelerates analytical queries by implementing relational operators using PyTorch’s tensor API. The advantage of using PyTorch not only relies on the ease of portability to diverse and rapidly evolving hardware [15, 18, 20], but also on taking advantage of any algorithm improvement coming from the ML community, while maintaining a familiar programming interface.

TQP is composed of a set of utilities for (1) converting input data into tensor format; and (2) a query compiler transforming input queries into tensor programs. For the former, TQP supports Pandas DataFrame [49], Parquet [17], Numpy [63], and CSV as input formats. TQP also supports integers, floating point as well as ASCII string (in dictionary and value-encoded formats), and date inputs¹. For the latter, given an input query, TQP uses an external query optimizer (e.g., Catalyst [70], SQL Server) to generate an initial physical query plan for single-node execution. TQP then

¹Notably, decimals are not supported. Decimal columns are currently loaded as floating-point numbers.

tensorizes [20] the plan by mapping each relational operator or expression into a tensor program (that can be either expressed directly in Python or exported in Torcscript format [3]) that is fed with the input data (in tensor format) to generate the query results. In TQP++ we reuse TQP’s stack (see Section 8) but add additional capabilities for supporting more efficient SQL query processing through ML compilers, that we describe next.

2.3 Tensor Compilers

The rapid growth of deep learning has produced a rich ecosystem of *tensor compilers*—including TVM [14], Triton [61], Antares [39], XLA [2], and PyTorch 2’s `torch.compile` [52]—that automatically generate optimized GPU kernels from high-level operator descriptions. A common design principle across these systems is the separation of *what* to compute from *how* to map it onto hardware, enabling the compiler to search for efficient schedules (tiling, vectorization, thread binding) independently of the algorithm definition.

Tensor IRs and einsum notation. At the core of these compilers lies a tensor intermediate representation that specifies computation declaratively. The most widely used formalism is the *Einstein summation* (einsum) notation, which describes tensor operations through named subscript indices. Each tensor is annotated with index variables that identify its axes, and the relationship between input and output indices determines the computation. Consider the following expression for matrix multiplication:

```
output[N, M] +=! input0[N, K] * input1[K, M]
```

Here, N , K , and M are index variables that range over the corresponding tensor dimensions. The index K appears on the right-hand side but not in the output subscript, so it is implicitly *reduced* over (the `+=!` operator denotes this parallel sum reduction). The compiler infers loop bounds from the tensor shapes and is free to choose tiling, thread mapping, and memory staging strategies.

This subscript-based style extends naturally to other common ML operations. Element-wise addition is simply `output[N] = A[N] + B[N]` (all indices appear on both sides, so no reduction occurs), while a batched dot product is written as `output[B] +=! A[B, K] * C[B, K]`, reducing over the shared dimension K independently for each batch element B . In summary, einsum is concise and maps naturally to the dense, regular access patterns of ML workloads such as convolutions, attention heads, and batched linear layers.

3 CHALLENGES AND WORKFLOW

Building a high-performance, portable GPU query processor on AI compiler infrastructure introduces four challenges:

Challenge 1: A Unified IR for ML and SQL. Repurposing an ML compiler for SQL query processing requires an intermediate representation that is expressive enough to capture both ML and relational operators, yet structured enough for the compiler to generate efficient GPU code. Standard ML IRs such as einsum notation are designed for dense, regular computations. They lack primitives for data-dependent control flow and indirect memory access. These limitations make pure einsum insufficient for SQL operators, which are inherently sparse and data-dependent. Simply extending a general-purpose language (e.g., CUDA or HIP) would sacrifice portability and automatic optimization. The challenge is to find an IR at the right level of abstraction: generic enough to express predicate filters, group-by aggregates, and hash-based joins, but

structured enough to enable automatic kernel generation, thread scheduling, and cross-vendor code emission. TQP++ addresses this challenge by extending the Antares IR [39], an expression-based language that generalizes einsum with conditionals, multiple reduction modes, and user-defined functions (Section 4).

Challenge 2: Bridging Dense and Sparse Operations. ML compilers such as TVM [14] and Triton [61] are designed for *dense* computations with regular memory access patterns. SQL operations, however, are fundamentally *sparse*: hash joins produce unpredictable output cardinalities, group-by aggregations scatter values into irregular buckets, and string matching requires overlapping memory accesses. TQP++ addresses this by introducing tiered GPU resource scheduling that decomposes sparse SQL operations into a hierarchy of fast on-chip computations (registers, shared memory) and coordinated off-chip accesses (HBM), drawing inspiration from analogous patterns in ML workloads (Section 5).

Challenge 3: Compilation Overhead vs. Execution Efficiency. Runtime compilation of fused kernels can take seconds (i.e., Section 9.7), which is unacceptable when query execution itself takes tens of milliseconds. Yet pre-compiled kernels cannot adapt to data characteristics (e.g., group-by cardinality, join selectivity). TQP++ addresses this with two fusion modes: Just-in-Time Fusion (JTF) for recurring queries, and Ahead-of-Time Fusion (ATF) for zero-overhead ad-hoc queries. Additionally, TQP++ leverages data-dependent switch nodes that can pick at runtime optimal operator algorithms without recompilation (Section 6).

Challenge 4: Architectural Heterogeneity. Modern GPUs vary significantly: NVIDIA uses CUDA with 32-thread warps; AMD MI series uses ROCm with 64-thread wavefronts; Xbox uses DirectX 12/HLSL without native 8-bit types support. Hand-crafting kernels for each target is prohibitively expensive. TQP++ leverages Antares’s expression-based IR to describe SQL operators above vendor-specific code, enabling a single description that can be compiled and tuned for each architecture (Section 7).

4 ANTARES IR

A key enabler of TQP++’s portability is Antares [39], an ML compilation framework whose IR we repurpose for SQL operator compilation. The Antares IR is an expression-based language that extends Einstein summation notation with conditionals, type casts, built-in function calls, and user-defined external functions, making it expressive enough for relational operations.

From Einsum to Antares IR. Standard einsum assumes dense, fixed-shape tensors and supports only a small set of reduction operators. Antares IR preserves this subscript-based style but adds several extensions that are critical for SQL:

- **Conditional execution** via `.when([cond], else_val=None)`, enabling predicated operations such as null handling and filters.
- **Gather/scatter indexing** via indirect subscripting (e.g., `input[ids[N]]`), essential for hash-based joins and group-by.
- **Multiple aggregation modes:** beyond summation (`+=!`), Antares supports `min (<=!)`, `max (>=!)`, logical AND and OR (`&=!`, `|=!`).
- **Function calls:** `.cast(“int32”)`, `.call(“exp”)`, and `.extern _call()` for invoking built-in and user-defined functions.

Given an IR expression and input tensor specifications, the Antares compiler automatically generates GPU kernels, fuses them (when

applicable), schedules thread blocks, and applies architecture-specific optimizations (e.g., vectorized loads, shared-memory tiling as we will see in the next sections) for each target device.

SQL Operators in Antares IR. We show how three representative SQL operations to better illustrate the expressivity of the IR:

Predicate filter (used in WHERE clauses) selects rows that satisfy a condition and compacts the result:

```
output[N] = data[N].when(pred[N])
```

The `.when(pred[N])` clause evaluates specified `pred[N]` for each row: when the predicate is true the corresponding value from `data` is written to output. This pattern maps directly to SQL selection: Antares compiles it into a branchless, vectorized kernel where every thread evaluates the predicate independently, avoiding warp divergence on modern GPU architectures. Note that the output retains dimension `N`: false rows are treated as null sentinels rather than being compacted out. Producing a *dense, compacted* result (i.e., one with no gaps) is a separate step, e.g., performed by the kernel of Section 5.3. Avoiding an explicit densification step, simplifies operation fusion between filters and successive computations.

Filtered scatter-sum (used in group-by aggregation) accumulates values into group buckets, skipping null rows:

```
output[ids[N]] +=! data[N].when(mask[N])
```

Here, `ids` maps each row to its group bucket, `mask` contains whether a row is valid or null, and `+=!` denotes a parallel sum reduction. Antares compiles this into an atomic-add kernel with architecture-specific conflict mitigation. Note that the output dimension `ids[N]` corresponds to a buffer of size equal to the group-by domain. TQP++ uses different strategies to compute the group-by domain, as we will further describe in Section 6.2.

Hash-table build (used in the build phase of hash joins) inserts keys into a hash table (`htable` tensor) via an external function:

```
build_key[N].extern_call('hash_build', [htable, htable_meta, N])
```

The `extern_call` invokes a hand-optimized C++ UDF (i.e., “`hash_build`”) that hashes each key and inserts it into `htable`, handling collisions via linear probing parameterized by the information stored in `htable_meta`. The surrounding IR handles input iteration and thread scheduling over the number of records `N`, while the UDF provides the logic that cannot be expressed in pure `einsum`.

Why an external function? We encapsulate hash-table mechanics in a hand-tuned UDF, while the surrounding `einsum` continues to govern input iteration, fusion, and scheduling. Recent vectorized GPU hash-table designs [34, 56] can be used as drop-in replacements as well, without disturbing the surrounding pipeline.

In summary, the Antares IR allows TQP++ to express SQL operators at a high level of abstraction, while being declarative enough to retain control over data access patterns, enabling strategies such as tiered GPU resource scheduling, described next.

5 IR SCHEDULING ON TIERED MEMORY

This section presents TQP++’s strategies for leveraging the layered memory and computational capabilities of GPUs to optimize core SQL operators. We report examples of how we exploit GPUs tiered memory for read optimization for string matching in LIKE statements (Section 5.1), write optimization in SQL group-by aggregation (Section 5.2), and efficient parallel writing of sparse results

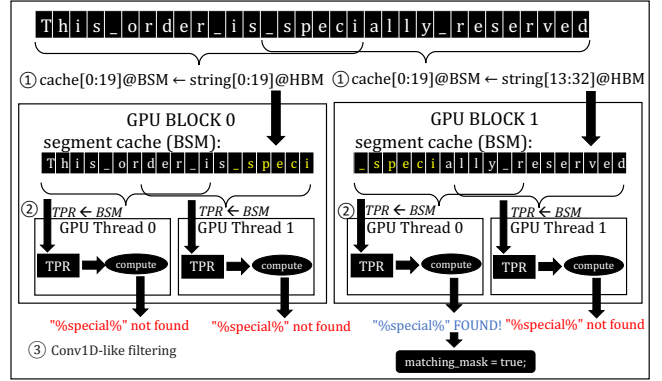


Figure 1: Tiered memory for string match algorithm for LIKE

for joins (Section 5.3). As discussed in Section 2, modern GPUs organize compute units hierarchically (threads → warps → blocks → kernels) with a corresponding memory hierarchy: thread-private registers, block-shared memory, and global HBM. TQP++’s key insight is to map SQL computations across these tiers to minimize expensive HBM accesses while maximizing on-chip computation.

5.1 Read Schedule for LIKE statements

Pattern matching within a main string in LIKE statements is a common but computationally expensive operation in query processing. Typically, this computation partitions the main string among multiple workers, each solving a local matching problem within their respective segments. Naive implementations, which allow individual threads to independently load string segments from HBM, bring severe memory access inefficiency. This inefficiency arises from substantial overlap in the regions of the main string accessed by concurrent threads, resulting in redundant HBM load operations on identical memory regions. We observe that this pattern-matching problem closely resembles the Convolution 1D problem in DNNs, as evidenced by their respective Antares IR expressions:

DNN Convolution1D: `output[N] +=! (x[N + K] × weight[K])`

SQL String Matching: `output[N] &=! (x[N + K] == substr[K])`

Motivated by these structural similarities, we leverage an analogous compilation strategy for SQL LIKE statements, *overlapped-segment cache*, to improve HBM loading efficiency through more effective memory access scheduling. As illustrated in Figure 1, the compiler schedules local threads within a block to process substring matching tasks in adjacent regions of the main string, ensuring that all threads within a block access contiguous regions with minimal total access length. To further eliminate redundant HBM accesses among intra-block threads, the compiler infers the memory access ranges in the main string required by any thread within the block (e.g., 0–19 for block 0 and 13–32 for block 1), and then copies these regions into intra-block shared-memory (`BSM ← HBM`) with a single HBM access. All thread-private character reads are then served from the shared-memory cache (`TPR ← BSM`) instead of HBM.

5.2 Write Schedule for Adaptive Aggregation

In contrast to overlapped segment cache which reduces load operations going to HBM for string pattern matching, in this Section

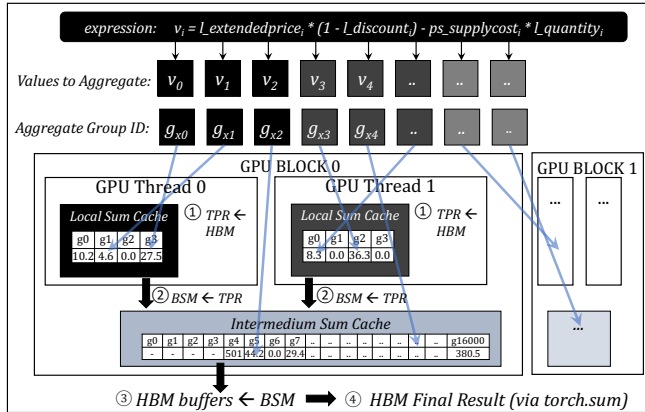


Figure 2: Group-by with Adaptive Aggregation

we describe a design for group-by aggregate IRs that reduces HBM store operations. This schedule targets the high HBM latency associated with updating output, as illustrated in the example below.

```
output[group_buckets[N]..] +=! (expr[N]..).when(pred[N]..)
```

While shared memory provides significant performance benefits over HBM, its effectiveness is limited by architecture-dependent capacity constraints. Consequently, this scheduling strategy focuses on maximizing on-chip memory utilization to accommodate group-by aggregation results. Given the device-specific BSM constraints, we propose a novel compilation stage mechanism that adaptively offloads portions of the aggregation process to registers and shared memory, which maximizes their synergistic contributions to the computation of the intermediate aggregate of data. In Figure 2 we show an example of a group-by aggregate over the expression `l_extendedprice * (1-l_discount)-ps_supplycost * l_quantity`. Assume that the vector v stores the results calculated from this projection for each row i ². Each element in the vector is scheduled to be accumulated into a single column entry of the output table based on another input vector that identifies their corresponding group bucket g_{xi} ³. Within the aggregation cache mechanism, we first initialize the architecture-dependent largest possible *sum-cache* in TPR and the BSM, such that K_1 groups are allocated in the registers *local sum-cache* and K_2 groups are allocated in the shared-memory *intermediate sum-cache*, where $K_1 < K_2$. Finally, we allocate K_3 groups as the *HBM-resident data*, composed of multiple disjoint HBM buffers to mitigate contention that would otherwise arise from directing all atomic updates to a single HBM buffer.

Afterwards, we orchestrate the subsequent aggregation operations through a four-stage progressive accumulation:

Stage 1: Accumulating data on the fastest possible tier. Each local thread across all GPU blocks fetches the assigned inputs (without overlaps). For each input, if the group index resides within the K_1 range, the corresponding value will be directly accumulated in the local sum-cache. Otherwise, if the group index falls within the K_2 boundary, the value will be intra-block atomically aggregated into the intermediate sum-cache. In the remaining cases, values undergo atomic accumulation into a buffer in HBM data.

Stage 2: Merging local caches into intermediate caches. This stage performs per-block column-wise aggregation of local

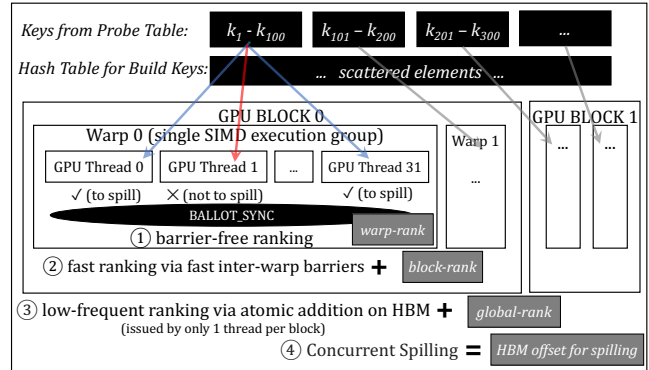


Figure 3: Strategy used to “densify” sparse data from joins

thread-private caches, which can be reliably implemented through warp-shuffle optimized cooperative block reduction which totally eliminates atomic operations. The aggregated results are then progressively copied into the intermediate cache in shared-memory.

Stage 3: Merging intermediate caches in HBM-resident data. This stage performs inter-block aggregation. Since the intermediate cache is shared among all threads within a GPU block, each local thread is scheduled to uniformly divide those K_2 elements to process their “atomic” aggregation into the HBM-resident buffers.

Stage 4: Dimensionality reduction on HBM buffers. Multiple copies of HBM-resident data, originally designed to mitigate potential inefficiencies triggered by inter-block atomic operations, will be further reduced into a single unified result as the final aggregation answer, which can be simply achieved by a naive `torch.sum()`.

5.3 Schedule for Sparse to Dense Operations

While GPU architectures achieve peak throughput through coalesced memory access and high SIMT occupancy on dense data structures, fundamental relational operators such as hash joins inherently produce sparse results even if inputs are in dense formats. This disparity creates significant challenges for hardware utilization. In this Section we propose a scheduling mechanism that implements a fine-grained, intra-kernel sparse-to-dense transformation that densifies data in-place before spilling to HBM.

Consider inner hash join as a representative example. The output generation of a join consists of all tuples from the probe side matching tuples from the build side. The sequential implementation of the output generation would be to simply append each tuple to the output. From this perspective the output is “dense” as in there are no gaps. To achieve a dense output format on GPUs, a naïve multi-pass strategy would be to materialize the output in different buffers (one per block), followed by a standalone densification kernel. This naïve strategy would result in prohibitive use of HBM capacity and memory I/O bandwidth. Conversely, relying exclusively on atomic global index accumulation for each match introduces severe memory contention, effectively serializing execution, and negating the throughput benefits of the underlying parallel architecture.

As illustrated in Figure 3, TQP++ decomposes the “sparse-to-dense” materialization process into three phases. (1) *Warp-level Ranking*: We leverage high-speed warp-level intrinsics to determine the local relative offset for each element within its respective warp. (2) *Block-level Prefix Sum*: We perform an intra-block prefix sum across the shared memory buffer to compute the relative offset for

²Note that TQP++ is able to fuse the full expression in a single kernel (see Section 6)

³Without loss of generality, we assume here that the bucket ids have been densified s.t. values are ranging from 0 to $N - 1$ where N is the number of distinct group entries.

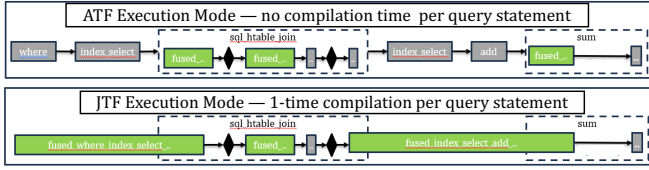


Figure 4: Relationship between JTF & ATF

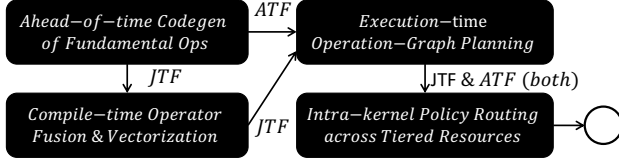


Figure 5: ATF and JTF Workflow

each warp within the block. (3) *Global Aggregation*: To minimize atomic contention happening on HBM, only the leader thread of each GPU block performs a standard atomic addition to acquire the block’s global HBM offset. By orchestrating these three hierarchical offsets, we derive the absolute HBM address for each element to write out to their dense output space concurrently.

6 ML-NATIVE COMPILATION IN TQP++

When queries can execute in few tens of milliseconds, spending seconds just for compiling them into efficient code is detrimental. To balance the advantages of ML-native compilation with the disadvantages of its latency, TQP++ introduces two distinct fusion modes (see Figure 4), each targeting different operational scenarios:

Just-in-time Fusion (JTF) triggers a full query compilation. As long as the query structure remains unchanged, this mode will not trigger re-compilation when parameters of the query change. JTF is particularly suitable for scenarios where input data and query constants evolve, while the query structure itself is stable, making it ideal for daily dashboard reporting, web services and logging systems that require recurrent query executions. As a matter of fact, this type of queries are the majority of production workloads [8, 71].

Ahead-of-time Fusion (ATF) eliminates the one-time compilation overheads of JTF. This strategy leverages pre-optimized GPU kernels to drive the whole query execution, which achieves zero compilation overhead for every new query statement, however, at the cost of missing opportunities of fusion. Thus, it is tailored for one-shot query scenarios, or queries that don’t benefit from fusion.

The four steps forming the TQP++ compilation workflow are described in Figure 5. While JTF goes through all the four steps $a \rightarrow b \rightarrow c \rightarrow d$, ATF path only passes through $a \rightarrow c \rightarrow d$:

(a) **Ahead-of-time: Code generation of fundamental operators.** Fundamental operators that are reused across many queries (e.g. radix sort, string pattern matching, topk with ties, hash table build) are compiled and tuned once for each target architecture. This eliminates repetitive compilation costs at runtime, while ensuring optimal parallelism and cache utilization for each GPU type.

(b) **Compilation-time: Operator Fusion & Vectorization.** After query optimization, TQP++ analyzes the structure of the physical plan to identify fusible operator sequences. Using the map-reduce rules (Section 6.1), it generates fused kernels that eliminate intermediate materializations. The fused code is vectorized to handle arbitrary input sizes within a single compiled kernel.

(c) **Execution-time: Operation-Graph Planning.** The multi-gated execution graph routes data through operator paths based on runtime data characteristics (e.g., join selectivity, group-by cardinality). Switch nodes inspect intermediate results and dynamically select the best algorithm without recompilation (Section 6.2).

(d) **Intra-kernel Policy Routing across Tiered Resources.** Divergent policy routing also occurs within a single device kernel, primarily driven by the non-deterministic output cardinality inherent in sparse relational join and group-by computations. This variable workload directly dictates the activation of specific memory levels within our hierarchical resource scheduler (e.g., thread-private registers, shared-memory, and HBM), as detailed in the tiered resource scheduling framework in Section 5.

6.1 Map-Reduce Fusion Schema

As individual queries may exhibit distinct fusion patterns involving user-defined functions, predicate filters, and expressions in aggregations, the combinatorial explosion of such operation permutations makes exhaustive ahead-of-time fusion infeasible. Thus, not all operations can be pre-optimized upfront. TQP++ just-in-time compilation generates one fused code that accommodates all executions across different input sizes and parameters. Specifically, we categorize fusible structures into two types:

Map Operations: which perform one-to-one transformation-like computations, such as: $\text{out}[N] = (1.0 - \text{l_discount}[N]) * \text{l_tax}[N]$.

Reduce Operations: such as hash table build or numerical aggregations consolidating multiple data rows into a single row, such as: $\text{out}[\text{ids}[N]] += (1.0 - \text{l_discount}[N]) * \text{l_tax}[N]$.

To compose the query-specific fusion blocks during the query statement preparation, each potentially fusible operator is recognized as either a **MapOp** or **ReduceOp**. Given two consecutive operations denoted as *precede_op* and *succeed_op*, TQP++ obeys the following rules to expand the scope of fusible blocks:

Map Rule: if both *precede_op* and *succeed_op* are **MapOps**, they can be fused into a new **MapOp**.

Reduce Rule: if *precede_op* is a **MapOp** while *succeed_op* is a **ReduceOp**, they can be fused into a new **ReduceOp**.

Stop Rule: the fusion expansion terminates if *precede_op* is identified as a **ReduceOp**, irrespective of the operation type exhibited by *succeed_op*. Consequently, *precede_op* and *succeed_op* are separated into distinct fusion blocks.

Execution granularity. Within a single fusion block, all map operations are pipelined into one kernel: a thread reads its input row(s), runs the entire chain of fused expressions in registers and shared memory, and writes only the block’s reduced output. No intermediate tensor is materialized between the map and the immediately-following reduce. Across fusion blocks, intermediates are materialized to HBM; this is by design and is what makes the multi-gated execution graph (Section 6.2) possible. The plan-divergence introduced by switch nodes is therefore at the granularity of *fusion blocks*, not threads inside a block: a switch picks one of several pre-compiled fusion-block alternatives, so divergent code paths never coexist within a single warp. This avoids the warp-divergence penalty and keeps each kernel SIMT-friendly.

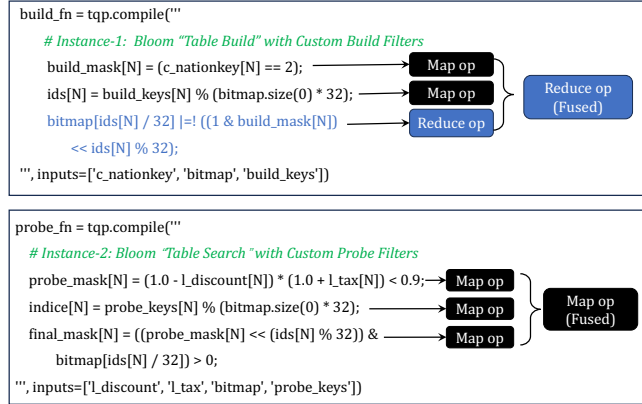


Figure 6: Fusion for the Bloom Filter IR - Build & Search (“Table Build” as a ReduceOp, “Table Search” as a MapOp)

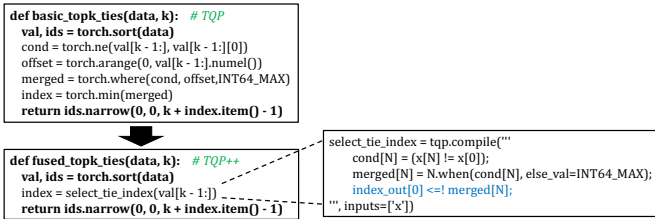


Figure 7: Operator fusion for top-k with ties

A bloom filter implementation example is shown in Figure 6. Table Build hashes *build_key* into the Bloom Table when *c_nationkey* values are equal to 2. The whole process is compiled into a **ReduceOp**. Table Search determines whether *probe_keys* matches with entries in the Bloom Table. The computations there are all compiled as a **MapOp**. Figure 7 illustrates another instance of another SQL operator: top-k with ties. This operation requires to calculate top-k results from input data while outputting all tied values when multiple entries share the top-k positions. In this operation, we aim to determine the index of the last data point that deviates from the top-k tied instances. Compared with the original TQP that utilizes comparison, condition, and reduction operators for tied top-k index identification, TQP++ integrates them into a fused operator implemented through an equivalent compilation interface as depicted in the right block of the Figure. This architectural advancement enables complete on-chip execution with one single data pass, instead of multiple operations and related materialization points.

6.2 Operator-Graph Planning

Traditionally, an optimized logical plan is composed of several operators such as join, group-by, projection, and sorting. For each logical operator, the optimizer then picks a physical implementation based on the query semantics and statistics. In TQP++ additionally we allow the runtime to pick the best algorithm given the input data characteristics. This runtime, data-driven strategy introduces uncertainty into the execution flow, which is at odds with the predetermined, fixed structure typical of DNN inference tasks.

To bridge this gap and enable SQL execution graphs to accommodate late-bindings of algorithms at runtime, TQP++ generates a data-driven graph with multiple output algorithm paths for online self-routing. Unlike traditional DNN execution graphs composed

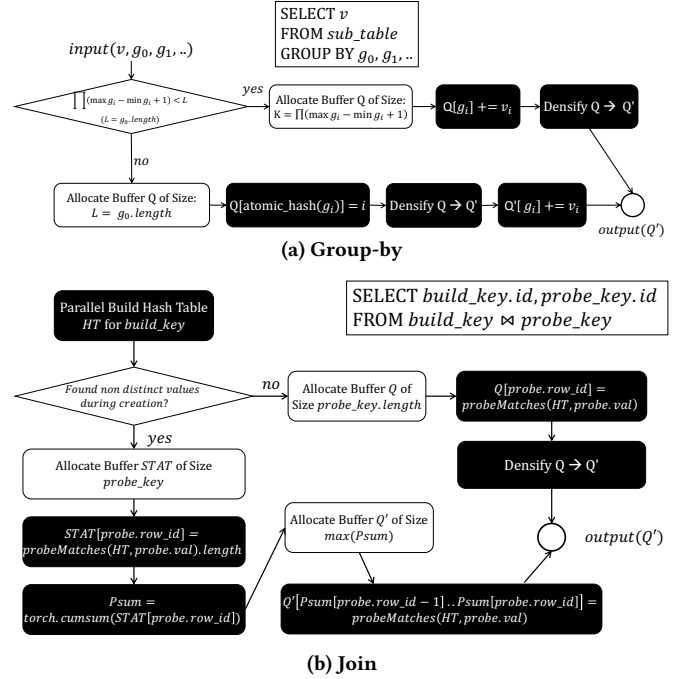


Figure 8: Operator graphs for group-by and join

solely of “op” nodes, this multi-path version integrates both “op” and “switch” nodes. Here, each “switch” node acts similarly to a decision-maker in traditional database runtime-optimizers [33]: using minimal overhead, it analyzes the current data distribution to select the optimal path for subsequent query operators. Furthermore, this decision process repeats throughout the execution, every time with additional “switch” nodes creating further branching. Consequently, the overall query execution graph forms an interwoven network of multiple potential paths, none of which are predetermined to be executed, but determined by on-the-fly data streams, and controlled by a set of “switch” nodes.

We present a detailed example using group-by aggregation to help illustrate how switch nodes are activated during execution. Group-by is used to partition rows based on one or more specified columns. In this process, rows with identical values are aggregated into a single output row. In the GPU environment, by leveraging the random-access capability of HBM, values belonging to identical groups can be accumulated to its target location in constant time $O(1)$ across different processors. However, depending on the distinctness of the underlying data, the memory space allocated for aggregated results may be sparse. For example, if the scores in a class only fall below 50 or above 60 with no scores in between, the group by operation should exclude the empty range (e.g. 50 - 60), ensuring that only the existing score values appear in output rows consecutively and densely. This example effectively demonstrates that two stages are required to perform a group-by aggregation: (1) an *aggregation* stage that accumulates the present data values into output buffer; and (2) a *densification* stage that removes gaps caused by data points not presented but reserved in output buffer.

TQP++ schedules a switch node in the generated graph that determines whether the densification stage is executed before or after the aggregation stage, ensuring that the worst-case memory complexity remains $O(L)$, where L is the number of columns used

for grouping, regardless of the distinctness of output instances in the result table. As shown in Figure 8a, we first compute the total range K of the data columns specified in the GROUP BY clause. If K is smaller than the length of the input columns L , we can create an output buffer of size K and perform the aggregation directly in this workspace. Once the aggregation is complete, we further densify the output buffer. However, if K is larger than L , creating an output buffer of size K directly would be inefficient and memory consuming. In this case, we use a workspace of size L to perform densification first (e.g. [10, 45, 10, 7, 7] can be densified to [0, 1, 0, 2, 2], where $L = 5$ and $L' = 3$), ensuring that every distinct original value can be mapped in $O(1)$ time to a dense domain ranging from 0 to L' , where L' is the number of distinct elements. Based on this densification map where L' is always less than or equivalent to the query-requested column length L , we further perform the aggregation within this compact domain, thereby eliminating any follow-up densification procedure. This realizes a “perfect-hash group-by” when $K \leq L$ (the group-by domain is small enough to be used directly as an index) and a regular hash group-by otherwise—the latter using the same UDF described in Section 4. The choice is made automatically at runtime by the same switch node.

Joins and exploding outputs. Joins follow a similar two-phase pattern (Figure 8b). The build phase populates the hash table via the UDF described in Section 4 and emits a small *side stream* of statistics such as the number of distinct build keys. A switch node consumes these statistics and routes the probe phase down one of two pre-compiled paths. (1) The *PK-FK fast path* assumes at most one match per probe row; it scans the probe in a single kernel that emits one tuple per match into a sparse output buffer of size $|probe|$ which is then densified. This path is selected when the build statistics report that the build side has unique keys (PK side). (2) The *many-to-many path* targets exploding joins where the output cardinality may exceed $|probe|$. This process consists of three steps: (i) count the total number of key matches; (ii) compute the prefix sum (Psum) of these counts; and (iii) re-scan the probe keys to perform the join and materialize the output, using the Psum to determine the precise write offset for each match. As a result, the runtime switch gracefully between the fast path and safe path based on the statistics emitted by the build phase.

7 BACKEND-SPECIFIC CODE GENERATION

In this section, we detail how TQP++ transforms computations expressed in our IRs into optimized device-specific code. For the sake of explanation, we use a representative SQL query illustrated below and involving a nested subquery and a global aggregate:

```
SELECT MAX(l_discount) AS max_discount FROM lineitem WHERE
  l_receiptdate > l_commitdate AND l_orderkey IN
  (SELECT o_orderkey FROM orders WHERE o_orderstatus = 'F')
```

The Graph Planner decomposes the complex query into a 2-stage execution graph as depicted in Figure 9.

Query Stage 1: Hash Table Materialization IR. The engine processes the inner nested subquery by inserting qualified build key entries into a global hash table.

Query Stage 2: Outer Query Aggregation IRs. The second stage probes the hash table generated in Query Stage 1. Rows passing the filter conditions are then sent to the MAX aggregate.

```
[Query Stage-1]
o_orderkey[M].extern_call('sql_ht_build',
  [htable_1, htable_metadata_1, M]).when(o_orderstatus[M] == ord('F'))

[Query Stage-2]
pos_in_htable[N] = l_orderkey[N].extern_call(
  'sql_ht_lookup', [htable_1, htable_metadata_1, N])

max_discount[0] >= l_discount[N].when(
  (l_receiptdate[N] > l_commitdate[N]) & (pos_in_htable[N] >= 0))
```

Figure 9: Fusion stages for the example query

CUDA/ROCm Backends. Figure 10 illustrates the generation of device code for Query Stage 2. The generated code makes use of both the Map-Reduce fusion schema of Section 6.1 (which instructs the filter IR **MapOp** and aggregation IR **ReduceOp** to be coalesced as a single fused **ReduceOp**), and the tiered write schedule for adaptive aggregation of Section 5.2 (which utilizes the thread-private registers and the block-shared memory to address the bottleneck of inter-thread contention of the HBM during global aggregations).

```
__global__ void custom_op_stage_2(int64_t* l_receiptdate, int64_t* l_commitdate,
  int64_t* l_orderkey, int64_t* htable_1, uint32_t* htable_metadata_1,
  double* l_discount, double* max_discount_buff, int N, int buff_cnt) {

  // Map-reduce fusion & thread-private register accumulation
  decltpe<*l_discount> local_0[1] = {-INF};
  for (int n = blockIdx.x * blockDim.x + threadIdx.x;
    n < N; n += blockDim.x * blockDim.x) {
    auto local_1 = LINEAR_HASH::sql_ht_lookup(
      l_orderkey[n], htable_1, htable_metadata_1, n);
    auto local_2 = (l_receiptdate[n] > l_commitdate[n]) && local_1;
    if (local_2) {
      auto local_3 = l_discount[n];
      local_0[0] = max(local_0[0], local_3);
    }
  }

  // Merge thread-private registers to block-shared memory
  __shared__ decltpe<*l_discount> shared_0[blockDim.x];
  shared_0[threadIdx.x] = local_0[0]; block_barrier();
  for (int t = blockDim.x / 2, id = threadIdx.x; t > 0; block_barrier(), t /= 2)
    if (id < t)
      shared_0[id] = max(shared_0[id], shared_0[id + t]);

  // Merge block-shared memory to HBM buffers
  if (threadIdx.x == 0)
    atomicMax(&max_discount_buff[blockIdx.x * (buff_cnt - 1)], 0, shared_0[0]);
}
```

Figure 10: CUDA Code Generation (Query Stage 2)

Xbox Backend. This backend leverages DirectX 12 utilizing High-Level Shading Language (HLSL). Most shader constructs within this framework maintain high semantic parity with CUDA primitives: groupshared memory is mapped to shared memory, RWStructured Buffer represents global memory buffers, and InterlockedCompare Exchange is employed for atomic operations. However, a significant constraint in HLSL is the absence of native support for 8-bit data types, which are critical for representing relational primitives such as booleans, and small integers in SQL workloads. TQP++ resolves this incompatibility through a compile-time data packing and unpacking mechanism. Under this scheme, retrieving 8-bit values involves bitwise extraction and masking from underlying HLSL 32-bit integers. To maximize throughput, our vectorized compilation strategy ensures that a single thread processes four contiguous 8-bit values simultaneously within a single uint32 register, which consolidates four accesses into a single memory transaction.

8 IMPLEMENTATION

In this Section we put all the technical contributions together and describe the end-to-end compilation workflow of TQP++. As shown

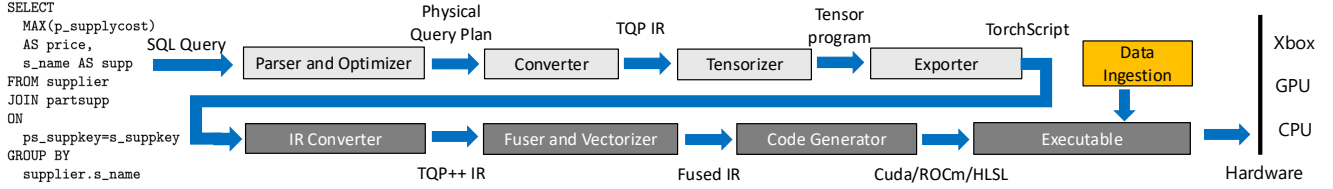


Figure 11: End-to-end compilation workflow. Light gray boxes are TQP components; dark gray boxes are new TQP++ components.

in Figure 11, TQP++ is built on top of TQP and leverages the TQP stack to generate tensor programs from input queries (light gray boxes, as described in [19] and in Section 2.2). When we run TQP++ in JTF mode, the full workflow is exercised: TQP components are used to generate Torchscript from input SQL queries; afterwards, we convert Torchscript programs into the TQP++/Antares IR (IR Converter). Once the query is in our IR format, we run the Fuser and Vectorizer (Section 6.1) followed by the Code Generator (Section 7). The final Executable runs the tensorized data over the different hardware supported, and using the switch nodes at runtime to properly route the data to the best algorithm (Section 6.2). When executed in ATF mode, TQP++ only uses TQP components at compile time. Nevertheless, TQP++ adds new operator implementations into TQP (Tensorizer) generated offline using the Fuser and Vectorizer, and Code Generator components. In an earlier implementation of TQP++ we additionally had a Tuner component doing a search over the hardware-sensitive parameter for each kernel. Over time we dropped this component since IRs coming from SQL queries are in general simpler than the one used by ML and AI workloads. We therefore decided to use simple heuristics for the hardware-specific parameters. This also made compilation time much shorter.

9 EXPERIMENTAL EVALUATION

In this section we compare TQP++ against state-of-the-art query processing systems over two sets of benchmarks. We answer the following questions: (1) What are TQP++ benchmark **performance**, and how does TQP++ compare against the CPU and GPU baselines? (2) How **flexible** is TQP++ in targeting different hardware efficiently? (3) When adopting compilation for query processing on GPU, what are the different **trade-offs and limitations**?

Workloads. In our evaluation we used two well-known benchmarks: The Star Schema Benchmark (SSB) [48] and TPC-H [62]. SSB contains 13 queries divided into 4 groups. Three queries have no joins, while the remaining have multiple joins (up to 4). All queries have filters and aggregates returning a single scalar value. TPC-H is one of the standard database benchmarks for measuring OLAP performance. It is composed by 22 queries, containing a wide range of operators such as joins (inner, semi, anti and left outer), group by aggregates, case statements, sub-queries, string and dates scalar operations (i.e., substring, datepart), etc. We pick these benchmarks because they closely represent many of the customers analytical workloads we see in production. In our experiments, we use SSB at scale factor (SF) 16 and TPC-H at SFs 10, 50 and 100.

Baselines. We compare TQP++ against 4 baselines: Crystal [57] and HeavyDB [21] on GPU, and DuckDB [53] and SQL Server on CPU. Crystal is the research prototype that has the state-of-the-art performance on the SSB benchmark, and achieved this result by manually implementing in CUDA/C++ all the queries. Because of this, Crystal *only* supports SSB. HeavyDB is a generic and mature database system, that can target both CPU and GPU query

execution. HeavyDB can execute both SSB and TPC-H. Internally, it uses Calcite [10] and LLVM [35] to generate GPU/CPU code for executing queries. DuckDB is a vectorized CPU-only database that has recently gained popularity because of its performance and portability. Finally, SQL Server is a commercial database.

Hardware Setup. To validate the performance and flexibility of TQP++, in the experiments we employ several devices ranging from GPUs (from 2 different vendors), a gaming console, and a CPU-only machine equipped with an Intel Xeon Platinum 8370C (Ice Lake) processor with 64 hyper-threaded cores @ 2.80GHz and 256 GB of DDR4 memory. Table 1 summarizes the specs of the hardware used for the experiments. Except for the GH200 and the Xbox, all other machines are publicly available Azure VMs. Both DuckDB and SQL Server are configured to use all 64 logical cores of the CPU.

Software Setup. All the machines (except Xbox) runs Ubuntu 20.04. Xbox runs a custom Windows version. For the experiments, we used CUDA 12, HeavyDB 8.1, DuckDB 1.1, Crystal as cloned from the repository [57], and the March 2025 version of SQLServer.

Experimental Setup. We generate data using benchmark data generation tools. For TPC-H we use doubles for decimals since TQP++ does not support them. We load data into each system and move it to GPU memory when applicable. For SQL Server, we create columnar indexes; for HeavyDB, we sweep fragment sizes and report the best per-query time. For each experiment, we do 10 runs: first 5 are warm-up runs; we report the median of the last 5.

9.1 SSB

We evaluate SSB at SF 16 on A100 (GPU systems) and CPU. For Crystal we use both the original [57] and optimized [58] versions. Results in Figure 12 (log scale) show Crystal Opt is fastest at 18.3ms total; TQP++ achieves 26.5ms (30% slower than Crystal Opt, but 40% faster than Crystal and 10× faster than HeavyDB at 251ms). TQP++ closely matches hand-tuned CUDA kernels through automatic compilation while maintaining full cross-vendor portability. CPU baselines are orders of magnitude slower (DuckDB: 509ms, SQL Server: 739ms) due to SSB being memory-bound.

9.2 TPC-H

We compare TQP++ against HeavyDB, DuckDB and SQL Server. As previously mentioned, Crystal only support SSB so we won't compare against it here. For HeavyDB, we have tweaked query performance by manually unnesting all sub-queries, and moving out count distinct clauses. HeavyDB currently is not able to run Q21 because of a Calcite limitation [7]. We will be evaluating the systems over the A100 and CPU-only machine and at SF 100. For TQP we will also report numbers on H100 and MI300 GPUs. The results are shown in Figure 13 (log scale).

Let's first focus on TQP++ and HeavyDB running on the A100. As we can see TQP++ is constantly faster than HeavyDB, for certain queries by more than an order of magnitude (e.g., Q2, Q4, Q11, Q16,

	Xeon	P100	V100	A100	H100	GH200	MI50	MI200	MI300	Xbox Series X
Vendor	Intel	NVIDIA	NVIDIA	NVIDIA	NVIDIA	NVIDIA	AMD	AMD	AMD	MICROSOFT
Driver Interface	-	CUDA	CUDA	CUDA	CUDA	CUDA	ROCm	ROCm	ROCm	D3D12
Available Memory (GB)	256	16	16	80	80	144	32	64	192	29
Memory Bandwidth (GB/s)	300	732	900	2039	3250	5000	1024	1638		560
Interconnect (GB/s, PCIe/C2C)	-	16	32	32	64	450	32	32	64	-
Theoretical TFLOps	2.6	9.5		19.5			13.3	12.0		

Table 1: Hardware Characteristics.

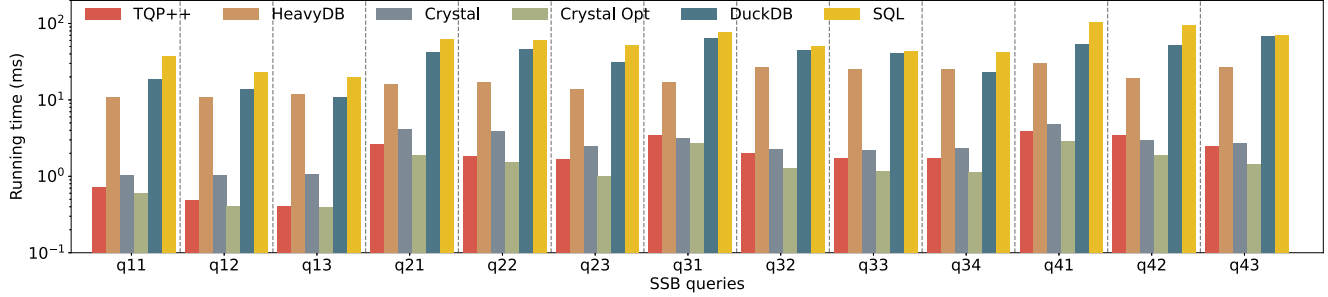


Figure 12: Performance (in log scale) over SSB at SF 16. Runtimes are in milliseconds. TQP++, HeavyDB and Crystal (Opt.) are run on the A100; DuckDB and SQL Server (SQL) on the CPU machine.

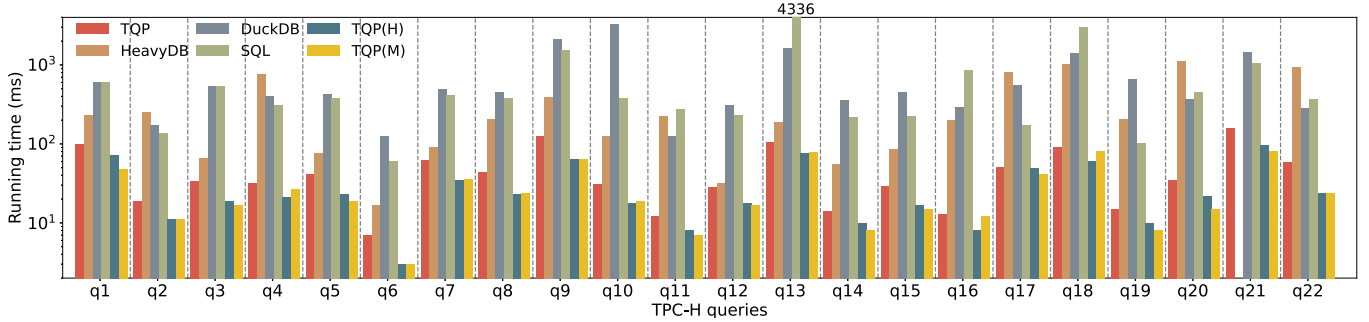


Figure 13: Performance (in log scale) over TPC-H at SF 100. Runtimes are in milliseconds. TQP++, HeavyDB are run on the A100; DuckDB and SQL Server (SQL) on the CPU machine. TQP(H) and TQP(M) run on an NVIDIA H100 and AMD MI300, respectively.

Q17, Q18, Q22). In aggregate, TQP++ runs the whole set of queries in 1.1s, while HeavyDB runs the 21 queries it can currently support in about 7.1s. Let’s now compare TQP++ against the CPU baselines. SQL Server takes 16.1s to run all the 22 queries, while DuckDB is slightly slower at 16.4s. TQP++ is hence about 15× faster.

Finally, we compare TQP++ on A100, H100 and MI300. H100 and MI300 are newer GPU generations than A100, and they are powered by HBM3 which is more than 50% faster than HBM2 used in the A100. We can see that H100 and MI300 run TPC-H queries in about half the A100 time. In aggregate, TQP++ can execute the full set of queries in 0.69s and 0.65s for the H100 and MI300, respectively. Interestingly, MI300 numbers are marginally better than H100 one despite having 30% higher memory bandwidth. This is due to better support for atomic operations in NVIDIA hardware than AMD.

9.3 Performance Across Devices

Figure 14 compares TQP++ on NVIDIA P100, V100, A100, H100, AMD MI50, MI200, and Xbox Series X using SSB SF 16 and TPC-H SF 10. Key takeaways: (1) within the same vendor, performance scales with memory bandwidth across generations; (2) AMD GPUs lag NVIDIA by tens of percent at similar specs, largely due to less mature software; (3) Xbox performs within 50% of the P100 despite its lower-cost hardware.

9.4 Memory Footprint

Table 2 reports the memory consumed by each TPC-H query at SF 100. We show the breakdown into the input working set (raw scanned columns materialized as tensors) and the live intermediates at the peak. Peak memory consumption ranges from 5GB (q16, q22) to 55GB (q21), with an average of 31GB. This means certain

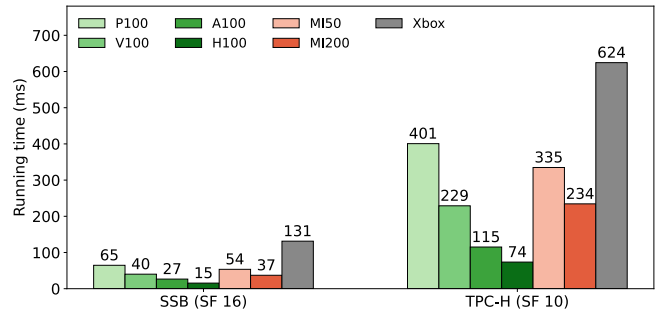
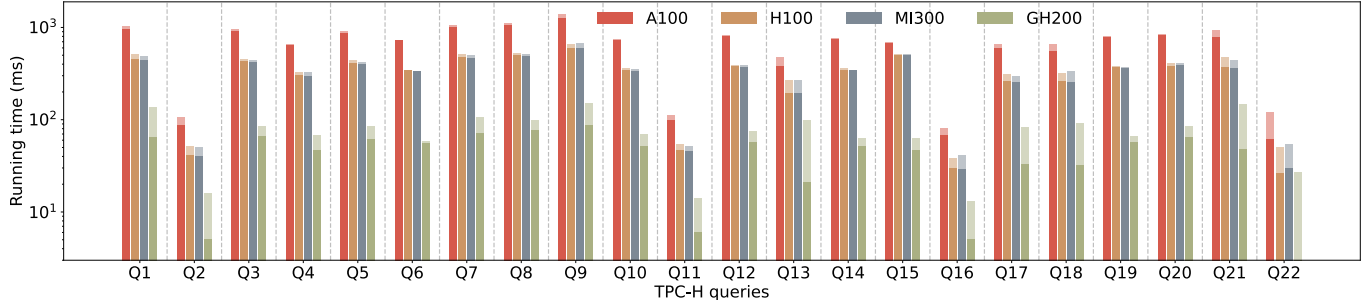


Figure 14: Total runtime of TQP++ across different devices, grouped by benchmark (SSB at SF 16 and TPC-H at SF 10). NVIDIA GPUs are shown in shades of green, AMD GPUs in shades of red, and Xbox Series X in grey.

Table 2: Peak memory usage (input working set + intermediates) and intermediate-only for TPC-H SF 100 queries.

Memory (GB)	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22	avg
Peak	49.2	10.3	36.6	28.1	37.8	18.9	52.3	39.8	52.5	32.8	9.3	37.2	19.8	29.0	28.6	5.0	36.6	40.8	30.3	28.2	55.6	5.7	31.1
Intermediate	25.6	8.1	13.7	12.1	16.0	0.9	27.2	13.3	21.1	14.6	6.9	17.3	17.9	10.8	10.6	3.5	22.9	26.8	10.8	8.2	36.2	4.2	14.9


Figure 15: Performance (in log scale) over TPC-H at SF 100 when data is warm in CPU memory. Runtimes are in ms and include both the time to move the data from CPU to GPU memory (in dark color), and the time to execute each query (in light color).
Table 3: Speedup of using JTF over ATF (AvsR) and JTF over custom kernels (CvsR). TPC-H SF 50.

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22	avg
AvsR	1.22	1.21	1.08	1.04	1.24	6.36	1.83	1.12	1.16	1.54	1.58	2.36	1.01	1.39	1.72	1.53	1.26	1.01	5.61	1.26	1.08	1.16	1.31
CvsR	8.99	5.37	7.00	2.74	4.73	16.43	6.28	5.33	8.51	17.79	5.32	6.22	10.98	6.39	3.93	12.78	13.17	4.86	16.91	3.84	7.48	1.38	7.85

Table 4: Compilation overhead in seconds (SF independent).

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22
Code Generation	0.05	0.14	0.07	0.03	0.06	0.01	0.08	0.08	0.08	0.11	0.07	0.04	0.05	0.03	0.06	0.11	0.02	0.11	0.02	0.12	0.09	0.08
Compilation Time	0.81	2.7	1.48	0.46	1.17	0.13	1.62	1.45	1.47	2.26	1.15	0.78	1.09	0.63	1.08	2.03	0.44	1.93	0.42	1.39	1.75	1.59

queries could support larger scale factors, but memory-hungry queries (q1, q7, q9, q21) bound the maximum achievable SF on a single GPU. Regarding the intermediates, Q6 uses under 1 GB of intermediate state — as expected for a single-table scan with one filter and one sum. The opposite regime is set by Q21, where 36.2 GB of intermediates dominate the 19.4 GB input due to the four-way self-join chain that exercises the many-to-many path of Section 6.2.

9.5 What if we start with Data in CPU Memory?

We evaluate TQP++ on TPC-H SF 100 when data starts from CPU memory, using four GPU devices (A100, H100, MI300, GH200) spanning PCIe 4, PCIe 5, and NVIDIA C2C interconnects [47]. Figure 15 shows that data movement dominates: A100 (PCIe 4) needs 15.7s total (14.6s in data movement); H100 and MI300 (PCIe 5) take about 7s each. GH200 with its high-bandwidth C2C interconnect runs all 22 queries in just 1.67s—about 10× faster than SQL Server (Figure 13).

9.6 Fusion Alternatives

We show the benefits of using JTF vs ATF vs using custom kernels directly in Pytorch (TQP). Table 3 shows the speedup over TPC-H SF 50. JTF is on average about 30% faster than ATF, and more than 7× better than TQP using custom kernels. The gap varies drastically: for q13, JTF matches ATF (meaning fusion does not bring additional benefits for this query), while for q6 JTF is more than 6× faster. This is because q6 can be compiled into just two kernels. Custom kernels are constantly slower due to less efficient code and the excessive data materialization generated by additional (un-fused) kernels.

9.7 Compilation Overhead

Table 4 shows the overhead of generating and compiling kernels (starting from the Torchscript representation). Code generation is fast (10–140ms depending on kernel count), but nvcc compilation takes 0.1–2.7s per query. Runtime compilation is therefore not viable when short running queries. For productization, we envision a separate compilation service [5] where common query patterns are compiled and cached, using ATF as fallback [13, 30, 31].

10 LESSONS LEARNED

TQP++ was our attempt of pushing the boundaries on the best achievable performance on single GPU. TQP++ has laid the foundations for several other works including distributed query processing on many GPUs [64], using compression to speed up out-of-GPU-memory reads [66], and more recently a new GPU-accelerated version of Fabric Data Warehouse [22]. Next, we describe some of the works built off of TQP++, and the lessons we learned.

Scaling out TQP++. Wu et al. [64] extends TQP++ to multi-GPU clusters by implementing shuffle and broadcast using group communication primitives. Each GPU runs a TQP++ process on its local data partition, with exchange operators automatically injected before joins and aggregations. Leveraging NVLink intra-node and RDMA over InfiniBand for inter-node communication, the system runs all 22 TPC-H queries at 1 TB scale in 0.53 s on 40 H100 GPUs, achieving over 60× speedup over high-end CPU machines.

Data processing on compressed inputs. When data exceeds GPU memory, CPU-to-GPU transfers over PCIe become the dominant bottleneck (as shown in Section 9.5). ZipFlow [66] addresses

this with a compiler-based framework that classifies compression algorithms into three parallel patterns, and applies hardware-aware scheduling to fuse and pipeline decompression kernels, overlapping PCIe transfers with on-device decompression. Integrating ZipFlow with TQP++ allowed compressed input data to be decompressed on-the-fly during GPU transfers, reducing both transfer time and memory footprint. On TPC-H and an A100 on PCIe 4, TQP++ with ZipFlow can achieve 2.08 $\times$ speedup over nvCOMP and 3.14 $\times$ over DuckDB in end-to-end query latency when data is in CPU.

Productization In 2023 we made the decision to start the productization of TQP++ as the GPU runtime for Fabric Data Warehouse (DW) [22]. The initial code base of the Fabric DW GPU runtime is a C++ port of TQP++, including all the ATF kernels, the custom kernel dispatching logic and the integration with PyTorch. For the product team, JTF is currently out of question due to the long compilation times. While TQP++ provided a strong foundations for the product, over the years we ended up manually rewriting all the ATF kernels directly in CUDA. This rewrite was driven by two product realities: (i) long-tail correctness requirements (proper overflow detection, ANSI-compliant null handling, locale-aware string and date arithmetic, and edge cases that were not the focus of the initial research prototype); and (ii) NVIDIA-only target deployment in the first product version, which removed the cross-vendor pressure that motivates portable kernels in the first place.

11 RELATED WORK

Single-GPU query processing. A large body of work studies single-GPU query processing, including operator optimization [57, 58, 60, 65], data placement [25, 68], interconnects [26, 36, 37], storage I/O [11], compression [59], system integration [24, 32, 43], and performance analysis [55, 69]. Crystal [57] achieves state-of-the-art SSB performance through hand-tuned CUDA kernels but is limited to fixed queries on NVIDIA hardware. SiriusDB [67] integrated with DuckDB and provides a hardened and more generic implementation of the ideas pioneered by Crystal, but it is still limited to NVIDIA hardware and lacks compilation-based optimization. HeavyDB [21] uses Calcite [10] and LLVM for code generation but suffers from query-parsing overheads and limited cross-vendor support.

ML compilers for hardware acceleration. The ML community has developed compiler frameworks for mapping computations onto heterogeneous hardware. Halide [54] pioneered separating algorithm from schedule; TVM [14] adopted this for deep learning with automated schedule search; XLA [2] compiles TensorFlow/JAX via HLO IR with algebraic simplifications and kernel fusion; Triton [61] exposes a block-programming model with automatic thread mapping; and PyTorch 2’s torch.compile [52] combines TorchDynamo tracing with TorchInductor code generation. Antares [39], which TQP++ builds upon, extends this paradigm with an expression-based IR and cross-vendor code generation. Recent work has explored integrating ML and relational operators: Raven [27] extends SQL engines with ML prediction operators, and Park et al. [50] further optimizes end-to-end ML prediction queries by pushing ML operations into the query plan. While these efforts bring ML into relational query plans, TQP++ takes the opposite direction: it adapts ML compiler infrastructure for SQL operations.

Compiled query execution and operator fusion. HyPer [44] pioneered compiling queries into machine code via LLVM with pipelining to avoid intermediate materialization. Umbra [45] is the direct successor to HyPer and addresses its the main practical limitation of the high LLVM compilation latency, by introducing a lightweight custom IR with an in-house x86 backend that emits machine code orders of magnitude faster than LLVM, and by reserving expensive LLVM optimization for long-running queries via an adaptive interpret-first / compile-on-demand mechanism [31]. We leave as future work how similar techniques can be ported to TQP++ to improve compilation time on GPU. Adaptive compilation [31] switches between interpreted and compiled execution at runtime; LingoDB [24] explores MLIR-based compilation; Amazon Redshift [5] uses compilation with an interpreted fallback. In ML compilers, kernel fusion reduces launch overhead, and recent work on incremental fusion [30] and recurrent query compilation [13] amortizes fusion costs across executions. TQP++ differs by targeting GPUs via an ML compiler (Antares) rather than LLVM, with GPU-native optimizations. Our dual ATF/JTF model (Section 6) switches between pre-compiled and runtime-fused GPU kernels, while our map-reduce-oriented fusion groups SQL operators into fused stages, reducing kernel launches and global memory round-trips.

GPU hash tables. Specialized GPU hash-table designs have been studied independently of full query engines: DyCuckoo [34] dynamically resizes cuckoo hash tables on GPUs to handle skewed key distributions, while Shahrokhi and Shaikhha [56] present a vectorized hash table for batch CPU/GPU computations. Our hash-table UDF (Section 4) follows similar warp-cooperative principles, and these designs can be used as drop-in replacements.

Sparse-tensor compilers. The TACO line of work [29] compiles sparse-linear-algebra expressions assuming statically-known sparsity formats (CSR, COO, ELL). Relational operators do not fit this model: the sparsity that emerges from a join, a group-by is *data-dependent* and visible only at runtime. We therefore build on Antares and defer materialization decisions to the gated execution graph (Section 6.2) rather than encoding them in a static sparse format.

12 CONCLUSIONS

We presented TQP++, an ML-compiler-native analytical query processor that repurposes the Antares compilation framework to achieve both high performance and hardware portability for GPU-accelerated SQL workloads. TQP++ bridges the gap between dense ML computations and sparse SQL operations through tiered GPU resource scheduling, map-reduce-oriented operator fusion, and a multi-gated execution graph with runtime switch nodes. On TPC-H SF 100, TQP++ executes all 22 queries in 1.1s on A100 (7 $\times$ faster than HeavyDB, 15 $\times$ faster than CPU baselines) and under 0.7s on H100, using a single portable codebase targeting 9 devices. As GPU-accelerated analytics [22] starts to appear in Lakehouse architectures like Microsoft Fabric [42], TQP++ provides a practical foundation for the next generation of compiler-based accelerated analytics.

13 ACKNOWLEDGMENTS

We acknowledge the use of AI-based language tools to assist with drafting and reviewing the paper; all technical content and claims remain the responsibility of the authors.

REFERENCES

- [1] 2020. OpenCL. <https://www.khronos.org/opencl/>. Accessed: June, 2026.
- [2] 2020. Tensorflow XLA. <https://www.tensorflow.org/xla>. Accessed: June, 2026.
- [3] 2022. *TorchScript Documentation*. <https://pytorch.org/docs/stable/jit.html>. Accessed: June, 2026.
- [4] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bockrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Susic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3204–3216. <https://doi.org/10.14778/3415478.3415545>
- [5] Amazon. 2024. Amazon Redshift Query Compilation. <https://docs.aws.amazon.com/redshift/latest/dg/c-result-caching.html>. Accessed: June, 2026.
- [6] AMD. 2022. ROCm. <https://rocm.docs.amd.com/en/latest/>. Accessed: 2022.
- [7] Apache Software Foundation. 2024. Apache Calcite Issue Tracker. <https://issues.apache.org/jira/projects/CALCITE>.
- [8] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriam Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 2205–2217. <https://doi.org/10.1145/3514221.3526045>
- [9] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *PVLDB* (2022), 3598–3601.
- [10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 221–230.
- [11] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. <https://doi.org/10.1145/3698812>
- [12] Rishi Bommasani, Drew A Hudson, et al. 2022. On the Opportunities and Risks of Foundation Models. *arXiv:2108.07258*
- [13] Nicolas Bruno and Surajit Chaudhuri. 2024. Workload-Driven Query Processing. *Proc. VLDB Endow.* (2024).
- [14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*.
- [15] Wei Cui, Qianxi Zhang, Spyros Blanas, Jesús Camacho-Rodríguez, Brandon Haynes, Yinan Li, Ravi Ramamurthy, Peng Cheng, Rathijit Sen, and Matteo Interlandi. 2023. Query Processing on Gaming Consoles. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) (DaMoN '23). Association for Computing Machinery, New York, NY, USA, 86–88. <https://doi.org/10.1145/3592980.3595313>
- [16] VOLTRON DATA. 2025. Theseus The Enterprise SQL Engine. <https://voltrondata.com/>. Accessed: 2025.
- [17] Apache Software Foundation. 2021. Apache Parquet. <https://parquet.apache.org/>. Accessed: March, 2025.
- [18] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2022. The Tensor Data Platform: Towards an AI-centric Database System. In *CIDR*.
- [19] Dong He, Supun Nakandala, Steven Dalton, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. In *Proceedings of the VLDB Endowment*, Vol. 15. 2811–2824.
- [20] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *PVLDB* (2022), 2811–2825.
- [21] HEAVY.AI. 2025. HeavyDB. <https://www.heavy.ai/product/heavydb>. Accessed: 2025.
- [22] Matteo Interlandi, Nicolas Bruno, Brandon Haynes, Carlo Curino, Rathijit Sen, Yinan Li, Kaushik Rajan, Bailu Ding, Lukas Maas, Wei Cui, and et al. 2026. CoddSpeed: Hardware Accelerated Query Processing in Microsoft Fabric. In *SIGMOD-Companion '26*. ACM. To appear at SIGMOD 2026 (Industry Track).
- [23] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snellman, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR abs/1704.04760* (2017).
- [24] Michael Jungmaier, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. <https://doi.org/10.14778/3551793.3551801>
- [25] Marko Kabic, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, 3 (2025), 187:1–187:25. <https://doi.org/10.1145/3725324>
- [26] Marko Kabic, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proc. VLDB Endow.* 18, 11 (2025), 4350–4363. <https://doi.org/10.14778/3749646.3749698>
- [27] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. <http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf>
- [28] Khronos Group. 2024. Vulkan: Cross-Platform Graphics API. <https://www.vulkan.org/>. Accessed: 2024.
- [29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, Article 77 (2017), 29 pages.
- [30] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive code generation for data-intensive analytics. In *Proceedings of the VLDB Endowment*, Vol. 11. 1172–1184.
- [31] A. Kohn, V. Leis, and T. Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208. <https://doi.org/10.1109/ICDE.2018.00027>
- [32] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquin, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: modular relational analytics over heterogeneous distributed platforms. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3308–3321. <https://doi.org/10.14778/3484224.3484229>
- [33] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. 2018. RIOS: Runtime Integrated Optimizer for Spark. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 275–287. <https://doi.org/10.1145/3267809.3267814>
- [34] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. DyCuckoo: Dynamic Hash Tables on GPUs. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 744–755.
- [35] LLVM Project. 2024. The LLVM Compiler Infrastructure. <https://llvm.org/>. Accessed: 2024.
- [36] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. <https://doi.org/10.1145/3318464.3389705>
- [37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1017–1032. <https://doi.org/10.1145/3514221.3517911>
- [38] McKinsey. 2024. AI Infrastructure Investment: Trends and Forecasts. <https://www.mckinsey.com/>. Accessed: June, 2026.
- [39] Microsoft. 2023. Antares: an automatic engine for multi-platform kernel generation and optimization. <https://github.com/microsoft/antares>. Accessed: June, 2026.
- [40] Microsoft. 2024. Microsoft SQL Server. <https://www.microsoft.com/en-us/sql-server>. Accessed: June, 2026.
- [41] Microsoft. 2024. Xbox Cloud Gaming. <https://www.xbox.com/en-US/play>. Accessed: June, 2026.

- [42] Microsoft. 2025. Microsoft Fabric: A Unified Analytics Platform. <https://learn.microsoft.com/en-us/fabric/>. Accessed: October, 2025.
- [43] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 877–890. <https://doi.org/10.14778/3636218.3636239>
- [44] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. In *Proceedings of the VLDB Endowment*, Vol. 4. 539–550.
- [45] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. [www.cidrdb.org](http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf). <http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf>
- [46] NVIDIA. 2024. CUDA Toolkit. <https://developer.nvidia.com/cuda-toolkit>. Accessed: June, 2026.
- [47] NVIDIA. 2024. NVIDIA NVLink-C2C. <https://www.nvidia.com/en-us/data-center/nvlink-c2c/>. Accessed: June, 2026.
- [48] Patrick O’Neil, Elizabeth O’Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*. 237–252.
- [49] The pandas development team. 2020. *pandas-dev/pandas: Pandas*.
- [50] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. ACM. <https://doi.org/10.1145/3514221.3526141>
- [51] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.
- [52] PyTorch Team. 2023. *torch.compile – PyTorch 2.0*. <https://pytorch.org/docs/stable/torch.compiler.html>. Accessed: 2023.
- [53] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. [www.cidrdb.org](http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf). <http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf>
- [54] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 519–530.
- [55] 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>
- [56] Hesam Shahrokhi and Amir Shaikhha. 2023. An Efficient Vectorized Hash Table for Batch Computations. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, USA*. 27:1–27:27.
- [57] 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*. ACM, New York, NY, USA, 1617–1632.
- [58] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. 1617–1632.
- [59] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD ’22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. <https://doi.org/10.1145/3514221.3526132>
- [60] 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>
- [61] Philippe Tillet, H T Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.
- [62] Transaction Processing Performance Council. 2024. TPC-H Benchmark Specification. <http://www.tpc.org/tpch/>. Accessed: June, 2026.
- [63] S. van der Walt, S. C. Colbert, and G. Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* 13, 2 (2011), 22–30. <https://doi.org/10.1109/MCSE.2011.37>
- [64] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. *Proc. VLDB Endow.* 19, 2 (2025), 141–155. <https://www.vldb.org/pvldb/vol19/p141-sen.pdf>
- [65] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1, Article 39 (Feb. 2025), 27 pages. <https://doi.org/10.1145/3709689>
- [66] Gwangoo Yeo, Zhiyang Shen, Wei Cui, Matteo Interlandi, Rathijit Sen, Bailu Ding, Qi Chen, and Minsoo Rhu. 2026. ZipFlow: a Compiler-based Framework to Unleash Compressed Data Movement for Modern GPUs. In *PVLDB*. Under submission at VLDB 2026 (Industry Track).
- [67] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devsh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2026. Rethinking Analytical Processing in the GPU Era. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. [www.cidrdb.org](http://cidrdb.org). <https://vldb.org/cidrdb/2026/rethinking-analytical-processing-in-the-gpu-era.html>
- [68] 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>
- [69] 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 (aug 2013), 817–828. <https://doi.org/10.14778/2536206.2536210>
- [70] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.
- [71] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD ’22)*. Association for Computing Machinery, New York, NY, USA, 2299–2311. <https://doi.org/10.1145/3514221.3526052>