Bridging the Indexing Gap in Fused GPU Query Engines

Tianjun Bu
Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
butianjun24@mails.ucas.ac.cn

Gaoyuan Zhou
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhougaoyuan@iscas.ac.cn

Xuhui Li
Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
lixuhui25@mails.ucas.ac.cn

Qiusong Yang*
Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences
Beijing, China
qiusong@iscas.ac.cn

ABSTRACT

GPU query backends achieve high throughput on analytical workloads through massive parallelism, but lack indexing support that accelerates selective queries in CPU databases. Existing GPU index implementations face three limitations: (1) supporting conjunctive predicates only, (2) materializing intermediate results between index access and query execution, and (3) assuming query boundaries align with pre-built index bins.

We present a fused bitmap indexing approach that addresses these limitations. We introduce virtual query program that executes arbitrary boolean predicates with low overhead. We fuse index access with subsequent column lookups, joins, and aggregation, keeping intermediate results in registers and eliminating global-memory round-trips. To handle misaligned query boundaries, we propose GPU friendly candidate checking that tracks three-valued row states (certain-in, certain-out, uncertain) through in-register boolean operations and verifies only the necessary candidates, without accessing global memory.

On Star Schema Benchmark SF=140 with RTX 5090 D, our fused bitmap index achieves up to 6.9× geometric-mean speed over our optimized non-indexed baseline built upon the Crystal GPU database query backend (Dense layout), and 4.3× with practical Sparse layout using less memory. Compared to current best compressed GPU bitmap implementation under perfect bin alignment (best case), our sparse layout achieves 1.4× speed end-to-end. We show that generic elementwise-style GPU fusion achieves only 1.34× speed, while our pipeline reaches 3.19× with 0.8% overhead versus dedicated compile-time kernels. Results on an NVIDIA H800 server GPU show the approach remains stable across GPU architectures, with smaller but still consistent fusion benefits on server GPUs.

PVLDB Reference Format:
Tianjun Bu, Gaoyuan Zhou, Xuhui Li, and Qiusong Yang. Bridging the Indexing Gap in Fused GPU Query Engines. PVLDB, 19(10): 2853 - 2865, 2026.

*Qiusong Yang is the corresponding author.

doi:10.14778/3828612.3828637

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/benbii/fusebmpidx.

1 INTRODUCTION

Indexing could accelerate query processing in data warehouses. Among various indexing techniques, bitmap indexes, which represent data by discretizing attribute values into bins and encoding each bin as a bit vector, are particularly effective for low selectivity (<10%) analytical workloads [38, 39], such as those in the Star Schema Benchmark (SSB[24]). The speedup is from only accessing bits rather than a data word for each row, which reduces memory access by 63/64 best-case, although with high selectivity most data still must be fetched, diminishing the benefits. This reflects the column-store principle of reading only necessary data [1], and for low-cardinality columns, bitmap indexes are especially compact and efficient compared to scanning raw data [7].

Modern GPUs offer memory bandwidths exceeding 1 TB/s and can execute thousands of threads concurrently. GPU memory capacities have also grown significantly, with devices now offering 20–200 GB of such high-bandwidth memory, sufficient to hold some working sets for many analytical workloads. For example, the AMD Instinct MI210 provides 64 GB of HBM2e memory with a bandwidth of 1.6 TB/s, while even consumer-grade GPUs like the NVIDIA RTX 5090 deliver 32 GB of GDDR6X and 1.7 TB/s global memory bandwidth.

Despite GPUs' massive parallel processing power and high memory bandwidth, current GPU-based query engines lack indexing support. While GPU query backends like Crystal [30, 31] excel at

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828637

full table scans, they cannot leverage indexing strategies that accelerate selective queries in CPU databases. Prior efforts on bringing indexes to GPU face several constraints. RTIndeX [12] and RTScan [19] repurpose ray tracing hardware for index operations, but as a consequence of ray-tracing primitives being only able to *intersect* with surface triangles, they support only conjunctive (AND) predicates on single tables. Disjunctions (OR) and joins cannot be expressed. Existing GPU bitmap implementations [2, 36, 37] focus on encoding and decoding compressed formats but operate on bitmap bins one pair at a time, without optimizing for arbitrary multi-predicate queries.

We identify three specific limitations that prevent practical adoption of bitmap indexing in GPU query engines. We use SSB Q1.1 as a running example:

```
SELECT SUM(extendedprice * discount)
FROM lineorder
WHERE discount BETWEEN 1 AND 3
AND quantity < 25
AND orderdate BETWEEN 19920101 AND 19921231
```

Bins are prebuilt offline: `discount` divided into $[0, 2)$, $[2, 5)$, ...; `quantity` into $[1, 11)$, $[11, 21)$, ...; `orderdate` into one bin per year. Special values like NULL can either merge into any bin or be binned separately.

Limitation 1: No unified support for arbitrary predicates. Most prior works [2, 22, 36, 37] port CPU bitmap algorithms to GPU, optimizing operations between two bitmap bins or two columns, but do not provide a unified mechanism for executing arbitrary combinations of conjunctions and disjunctions across multiple columns. As a result, complex predicates like evaluating `discount BETWEEN 1 AND 3` (OR-ing bins $[0, 2)$ and $[2, 5)$, then AND-ing with the quantity and date bitmaps require multiple kernel launches and intermediate materialization (i.e., explicitly storing intermediate results back to memory) on *all* these works. RTIndeX and RTScan [12, 19] support multi-column conjunctive selections via ray tracing primitives, but cannot express disjunction or joins.

Limitation 2: No integration with query pipelines. All prior GPU indexing works focus exclusively on the index access phase, which produces a list of qualifying row identifiers or a result bitmap with qualifying bits set to 1. They do not address how these results are integrated into subsequent query operations—such as column lookups, joins, and aggregations. On CPU, execution models such as Volcano-style iterators or push-based pipelines (pivoted by DuckDB [26, 27]) naturally decouple operators while cache hierarchies hide inter-stage latency. These decoupling-friendly models do not translate to GPU: massively parallel stream processing requires all threads to execute the same code path, and stage boundaries force global memory round-trips rather than register or on-chip memory residency. Kernel launches add millisecond of overhead, and more crucially intermediate results traverse global memory at 1–2 TB/s, orders of magnitude slower than on-chip storage. Breaking operator fusion, a key optimization in GPU query backends like Crystal, reintroduces the very memory bandwidth bottlenecks that indexing was intended to alleviate.

Limitation 3: No handling of misaligned query boundaries. Bitmap index bins are created ahead of time without knowledge of future queries [35]. In Q1.1, the predicate `lo_discount BETWEEN 1 AND 3` maps to bins $[0, 2)$ and $[2, 5)$: both are *boundary bins* whose ranges extend beyond the query range. Rows with `discount`$= 0$ or `discount`$\in [3, 5)$ are in these bins but do not qualify. They are *uncertain* and require verification against the actual column value. This is a form of three-valued logic: rows are either *certain-in* (fully inside the query range), *certain-out* (fully outside), or *uncertain* (in a boundary bin). Unlike bloom filters, bitmap bins have no false positives *within* a bin; uncertainty arises solely at boundaries. No prior GPU indexing work addresses this scenario; all assume perfect alignment between queries and bins. CPU-based systems manage this by either optimizing the subsequent verification step [32, 41] or strategically placing bin boundaries [28], but these rely on sequential access patterns ill-suited for GPU execution.

Addressing these limitations individually would yield incremental improvements, but their integration is where GPU-specific design becomes critical. Unlike CPU systems where decoupled execution models (Limitation 2) and cache hierarchies naturally tolerate loose coupling between components, GPU's massively parallel stream processing offers no such opportunity and each interface between components risks a global memory round-trip that dominates execution time. CPU techniques for candidate verification (Limitation 3) similarly assume sequential access patterns incompatible with GPU execution. Our contribution is not merely solving three problems, but co-designing their solutions so that data flows from index access through candidate checking to query execution entirely within registers and shared memory, a property that requires careful resource budgeting (Section 6) and would provide negligible benefit in CPU systems where such tight coupling is unnecessary. We present a fused bitmap indexing approach that addresses these limitations with GPU-native solutions:

Virtual query program for arbitrary predicates (Section 3.1). We introduce a lightweight abstraction that represents arbitrary predicates as a sequence of simple instructions operating on virtual registers backed by shared memory. This is a purely runtime technique: the GPU kernel is compiled once and interprets different instruction sequences for different queries, requiring no query-specific code generation. Index access phase reduces to a single step with no global intermediate results. Ablation studies show this abstraction incurs only modest overhead compared to hand-written kernels specialized for each query (0.8% on consumer and 4.6% on server GPU; Section 8.4).

Fused index-to-data pipeline (Section 4). We design a complete pipeline that fuses bitmap index access with subsequent column lookups and aggregations. We managed to keep index access results in registers and eliminates several materialization steps. Fusion provides 1.58× speed on consumer and 1.33× on server GPU (less due to constrained register resources) by eliminating global-memory materialization of intermediate index results.

Candidate checking for misaligned boundaries (Section 5). As shown in the Q1.1 example, boundary bins produce three-valued outcomes: certain-in-out or uncertain. We track two bits per row, derive propagation rules through boolean operators, and carry the uncertainty flag into the fused pipeline so that only uncertain rows trigger verification against actual column values.

Database integration (Section 7). We integrate our bitmap indexing approach into the Crystal query backend [30]. Our implementation incorporates key GPU optimizations including predicated loads [6] and GPU-optimized group-by aggregation such as

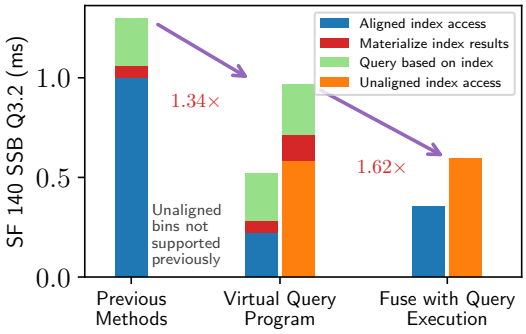

Figure 1: Performance progression on SSB Q3.2 (SF=140) on an RTX 5090 D GPU. Our virtual query program supports both aligned and misaligned bins with 1.35× speed over previous methods on Q3.2. Fusing with query execution multiplicatively bumps speed to 1.62×.

warp-level atomic reduction techniques inspired by [8]. Our optimized baseline already outperforms vanilla Crystal; adding bitmap indexing provides further speedups.

We evaluate our approach on all 13 SSB queries at scale factors 20, 140 and 300, as well as synthetic workloads with varying selectivity and skew. With a practical memory-saving Sparse bin layout our fused bitmap index achieves 4.31× geometric mean speed over an optimized non-indexed baseline at SF=140. Denser layouts provide up to 6.9× speed. We focus on *uncompressed* bitmaps throughout this work. While most prior GPU bitmap work targets compressed formats (e.g., WAH [22, 36, 37]), decompressing a bitmap requires variable-length sequential processing. Its parallelization and merging with downstream is outside our scope. Still, section 8.2 shows memory overhead of uncompressed bitmaps is modest (within 1% for our Sparse layout) and constructs at 4.4× speed. Figure 1 illustrates our approach and improvements on SSB Q3.2.

2 BACKGROUND AND RELATED WORK

2.1 GPU Thread Blocks and Kernel Fusion

A GPU kernel launches as a grid of *thread blocks*, each containing a fixed number of threads; a block is scheduled onto a single streaming multiprocessor (SM) [14, 23]. Within a thread, *registers* provide the fastest private storage. Threads within the same block cooperate via *shared memory*, a programmer-managed on-chip scratchpad that enables data reuse and low-latency communication. Read-only parameters that are uniformly accessed by a warp can be stored in *constant memory* (*cmem*), which supports efficient warp-wide broadcast when all threads read the same address.

In contrast, *global memory* provides high capacity and is globally accessible, but has substantially higher latency and lower bandwidth than on-chip storage (empirically 1:100 ratio). Achieving high effective bandwidth relies on *coalescing*: when threads in a warp access consecutive addresses, the hardware can combine these requests into a small number of wide memory transactions. These properties make the thread block the natural granularity for staging data on chip and for composing multiple operations.

The number of threads per block (*nt*) and values each thread processes (*vt*) are compile-time constants balancing occupancy, register pressure, and instruction-level parallelism (ILP). A larger *vt* increases ILP but also increases per-thread resource consumption, potentially reducing occupancy. Here, occupancy refers to the ratio of active warps to the maximum number of warps that can be concurrently supported on a streaming multiprocessor (SM), and it directly determines the utilization efficiency of key hardware resources such as registers, shared memory, and above all, degree of concurrent access to global memory. The thread block is the fundamental unit for shared memory allocation: all threads in a block share the same shared memory region, and it is the natural granularity for *kernel fusion*.

Kernel fusion combines multiple logical operations into a single GPU kernel, passing intermediate results through registers or shared memory rather than global memory. For memory-bound applications like databases, skipping the global memory round-trip directly boosts performance. Crystal [30] demonstrated that fused analytical query kernels execute at near-memory-bandwidth speeds. Effective kernel fusion requires its fused operations satisfy three constraints:

Block-granularity data format. Data must be partitioned into blocks where each block fits in shared memory and processed independently by a single thread block without synchronization.

Tiled routine. Each operation must be expressible as taking a tile (block of data) in shared memory as input and producing a tile as output, *with maximum tile size known at compile time.*

Tight resource provisioning. The sum of all intermediate storage must not exceed hardware limits. For example, if using atomic operations for group-by aggregation [10], the maximum number of groups must be bounded; otherwise, intermediate results spill to global memory, negating fusion benefits.

2.2 GPU-Based Indexing

Early work on GPU bitmap indexing focused on compression formats. Andrzejewski and Wrembel [2] first demonstrated GPU acceleration of Word-Aligned Hybrid (WAH) bitmap operations, focusing on encoding and decoding. Subsequent work extended this to Position List WAH (PLWAH) [3], but neither addressed query execution. Tran et al. [36, 37] applied these techniques to indexing but processed bins one-after-another without global memory access optimization. Nelson et al. [22] fused access to multiple bins within a single column, but inter-column operations remained sequential due to uncertain query flights at compile time.

A parallel line of work explored GPU-accelerated indexing other than bitmaps. Mallia et al. [20] designed GPU decoding for Variable Byte and Bit Packing formats, though list intersection and union still operated bin-by-bin. RTIndeX [12] and RTScan [19] repurposes GPU ray tracing cores for index operations by encoding predicates as geometric intersection tests. While useful on *conjunctive selections*, these approaches fundamentally cannot support disjunctions or multi-table joins due to ray-triangle intersection semantics.

All these works focus on improving the *index access phase* in isolation. We unify various aspects into an index access engine capable of handling arbitrary predicates in one fused procedure,

then integrate it into a full query pipeline from index access through final result computation.

2.3 GPU Query Engines

GPU query processing has evolved from accelerating individual operators [15, 29, 33] to complete query backends. Crystal [30] marked a breakthrough by providing composable CUDA device functions that execute analytical queries in customizable fused kernels. Crystal is not a full database engine but rather a *backend*, i.e., a collection of hand-written GPU kernels demonstrating achievable query performance without core frontend features like SQL parsing; we integrate our indexing techniques there, thus creating another backend. Crystal demonstrated that fully-fused GPU query execution can approach memory bandwidth limits, delivering an order of magnitude speedup over CPU implementations on SSB. Comparative studies have shown that Crystal outperforms commercial GPU databases like HeavyDB (formerly OmniSci) and BlazingSQL on analytical workloads [6, 30, 40], establishing it as the state-of-the-art GPU query backend. Subsequent work improved Crystal in various dimensions: trading speed for reduced memory usage through compression [31], exploring efficient hash table designs and load skipping [6]. However, Crystal's query interface is limited to SSB-style star/snowflake schema joins, and all Crystal-based research has consequently focused on SSB [6, 31]. Other works focus on scaling to multiple GPUs [9, 18, 25] or reusing existing frameworks [11].

To the best of our knowledge, no GPU query engine incorporates bitmap indexing (or any indexing at all given the limitations in section 1). Our work bridges this gap by providing a GPU backend with complete pipeline from index access to result computation.

3 FUSED INDEX ACCESS ON ARBITRARY PREDICATES

This section addresses Limitation 1 from Section 1: the lack of unified support for arbitrary predicates in GPU bitmap indexing. We introduce a lightweight abstraction called a *virtual query program* that fuses index access for arbitrary boolean combinations of bitmap bins into a single kernel. This is a *runtime* technique: the virtual program is interpreted at query time without any GPU kernel compilation or code generation. A new query only requires populating a small instruction buffer and bin pointer array, which takes microseconds on the CPU.

3.1 Virtual Query Program

Recall that a bitmap index represents each column predicate as a bit vector over rows: bit i is set if row i satisfies the predicate. Evaluating a multi-predicate query reduces to bitwise operations on these vectors. A *virtual query program* consists of a sequence of instructions that operate on *virtual registers*. Each virtual register holds an intermediate bitmap during query evaluation. Importantly, virtual registers are logical temporaries backed by on-chip storage, and should be differentiated from hardware CUDA registers. Instructions can be combined to express arbitrary boolean predicates. For instance, the query $(A \vee B) \wedge C$ could compile to: `LOAD r0, A; LOAD r1, B; OR r0, r1; LOAD r1, C; AND r0, r1`. We establish a "call convention" that the final result bitmap returned by this program resides in register r_0, which subsequent pipeline stages could consume directly.

Expressiveness. Here we discuss which predicates can be expressed given n live temporaries, while the hardware resource cost of choosing n is analyzed later in Section 6.

We model a predicate as an expression tree whose leaves are *atomic bitmap operands* (e.g., a bin bitmap, or a union of multiple bins within one column) and whose internal nodes are binary operators (AND/OR/...). Any such tree can be compiled to a straight-line program that evaluates subtrees and combines them. The key question is the *maximum number of intermediate results that must be kept simultaneously* during evaluation.

Let $\rho(T)$ be the minimum number of virtual registers needed to evaluate an expression tree T without spilling intermediates to global memory. For a leaf operand, $\rho(\text{leaf}) = 1$. For an internal node with two children T_1 and T_2, we can evaluate the child that needs more registers first, so the standard recurrence is:

$$\rho(T) = \begin{cases} \max(\rho(T_1), \rho(T_2)) & \text{if } \rho(T_1) \neq \rho(T_2), \\ \rho(T_1) + 1 & \text{if } \rho(T_1) = \rho(T_2), \end{cases}$$

which corresponds to the minimal stack depth of a postfix evaluation order. Therefore, a program with n virtual registers can express any predicate whose expression tree satisfies $\rho(T) \leq n$.

This characterization matches intuition. For example, successive conjunction and/or disjunction are left-deep chains: $a_1 \times a_2 \times \cdots \times a_k$ (where $\times \in \{\wedge, \vee\}$) need only one loader register (for `LOAD` a_k and one accumulator register (for $\times$). Two accumulator registers suffice for predicates with two independent groups combined at the top level, e.g., $(a \vee b \vee \cdots) \wedge (c \vee d \vee \cdots)$. With three accumulator registers, we can handle an additional level of nesting, e.g., $((a \vee b \vee \cdots) \wedge (c \vee d \vee \cdots)) \vee (e \wedge f \wedge g \wedge \cdots)$, sufficient for expressing most practical predicates. Section 3.3 discusses complex memory-operand instructions that skip loader registers and further expand expressibility (and performance) for a given virtual register budget. For Star Schema Benchmark, three virtual registers (two accumulators) already cover all query predicates we study. We keep the detailed resource/occupancy implications of this choice to Section 6.

Frontend Friendliness. Compiling a query into a virtual program is straightforward. For each column predicate, frontend has to inspect the relevant bin metadata and store their pointers in the program's column array. Since bitmap AND/OR are commutative (and associative within an AND/OR group), the frontend can reorder operands without changing semantics. A common heuristic is to process low-selectivity predicates first, which tends to produce intermediate bitmaps with more zero bits. Section 3.3 shows how this property enables subsequent instructions to skip global memory loads.

3.2 Implementation

Algorithm 1 lists the complete index access phase. Threads execute it in parallel, each processing vt words from its assigned positions. The outer loop iterates over instructions in the virtual program; the inner loop provides instruction-level parallelism by processing vt bitmap words per thread per instruction. The `AND` instruction

Algorithm 1 Fused Index Access Device Function

Input: Program P
Output: Result bitmap in r_0 (shared memory)

```
1:  tid ← threadIdx.x
2:  base ← blockIdx.x · nt · vt
3:  for each instruction I in P do
4:      for k ← 0 to vt − 1 do
5:          pos ← tid + k · nt
6:          wordIdx ← base + pos
7:          if I.opcode = AND then
8:              r_{I.dst}[pos] ← r_{I.dst}[pos] ∧ r_{I.src}[pos]
9:          else if I.opcode = ANDM then
10:             srcword = r_{I.src}[pos], dstword = 0
11:             if srcword == 0 then
12:                 continue
13:             col ← P.cols[I.src]
14:             for each bin b in col.bins do
15:                 dstword ← dstword ∨ b[wordIdx]
16:             r_{I.dst}[pos] = srcword ∧ dstword
17:         else if other instructions ... then
```

performs a register-to-register bitwise AND. Memory-operand instructions (e.g., `ANDM`) load bin bitmaps from global memory and combine them with a register; Section 3.3 describes these in detail.

Memory accesses are coalesced in Algorithm 1. Consider line 15 of Algorithm 1. When threads within a warp execute this line, thread t accesses $b[\text{base} + t + k \cdot nt]$. For $k = 0$ and consecutive thread IDs $t = 0, 1, \ldots, 31$, these accesses target consecutive words $b[\text{base}], b[\text{base} + 1], \ldots, b[\text{base} + 31]$ and allows GPU coalesce adjacent loads into single transactions. Other instructions involving loads maintain this access pattern.

Shared Memory Layout. Virtual registers are backed by shared memory with stride nt: thread tid owns words at positions $tid, tid + nt, tid + 2nt, \ldots$. This layout ensures bank conflict-free access when threads write to their local slots. Each virtual register consumes $nt \times vt$ words of shared memory. Given the limited size of shared memory, we can see the number of virtual registers trades off with instruction-level parallelism. Increasing vt improves ILP but reduces the number of registers we can afford. We discuss this trade-off quantitatively in Section 6. Furthermore, *the r_0 Calling Convention* allows threads to load r_0 into CUDA registers (vt words each thread), then deallocate all shared memory backing virtual registers. The entire shared memory becomes available for later stages (relevant in Section 4.2).

3.3 Optimization

Optimization 1: Constant Memory Storage. Since the virtual query program is not modified during query execution, we pass it to constant memory in CUDA kernel. This prevents storing the program itself from competing for resources with virtual registers in shared memory. With 8B pointers to bins, 4B instructions (opcode, destination, two source operands) and up to 32 bin bitmaps accessed *per column per query* (a column with > 32 bins still only accesses the few bins overlapping the predicate range; number of bins a column can be split into is unbounded), a query with 50 columns and 50 instructions only consumes 13 out of 64K constant memory. This theoretical scalability is sufficient for complex queries with numerous predicates across dozens of columns.

Optimization 2: Complex Memory Operand Instructions. Analytical queries typically have low selectivity. For example, SSB Q1.1 has 1.9% selectivity and Q1.2 has 0.065% selectivity. At such low selectivities, intermediate results often contain long runs of zero bits. Having 128 consecutive zero bits (the granularity of a single CUDA memory transaction) is common. We exploit this by introducing *memory operand instructions* that can inspect another operand before loading from memory, skipping the load when the result is predetermined. An example is `ANDM` (AND-with-memory): `ANDM r0, col` both loads and computes $r_0 \leftarrow r_0 \wedge \bigvee_x \texttt{col}[x]$. When r_0 is already zero, the AND operation cannot produce a non-zero result regardless of the memory operand, so we skip loading bins entirely. A memory transaction is skipped when 4 adjacent thread all have zero registers. Such a predicated-load design aligns well with key GPU optimization principles summarized in [6, 13].

We further introduce complex instructions with multiple memory operands. For example, `ANDORM r0, col1, col2` computes $r_0 \leftarrow r_0 \wedge \bigvee_x (\texttt{col1}[x] \vee \texttt{col2}[x])$. In plain terms: it ANDs r_0 with the union of all bins from two columns simultaneously, loading both columns' bins in a single pass. This enables the query $(a \vee b) \wedge (c \vee d)$ to be compiled as: `LOAD r0, a; ORM r0, b; ANDORM r0, c, d`. If 128-bit segments of $a \vee b$ are zero, we skip loading both c and d for those segments—the zero-skipping check on r_0 gates the loads for *both* columns at once. Without this instruction, we would need: `LOAD r0, a; ORM r0, b; LOAD r1, c; ORM r1, d; AND r0, r1`, which requires 1 additional register and cannot skip c and d because they are loaded into a separate register before the AND, so the zero check on r_0 cannot gate those loads.

Section 8.4 shows that the virtual query program overhead remains modest on average: the overhead is 0.8% at SF=140 (consumer GPU) and 4.6% at SF=300 (datacenter GPU), compared to hand-written kernels specialized for each query at compile time.

4 INTEGRATION WITH QUERY PIPELINE

The virtual query program from Section 3 evaluates arbitrary predicates and produces a result bitmap with bit i set if row i qualifies. A complete query pipeline requires further operations: fetching column values for qualifying rows, performing joins, computing aggregations, and producing final results. This section addresses Limitation 2 and integrates index access with subsequent pipeline.

A natural question is why general-purpose GPU kernel fusion (e.g., PyTorch or TVM elementwise fusion) is insufficient. There are three structural reasons: (1) the set of bins accessed is determined at query time by the virtual program, not a fixed operator graph; (2) the index phase produces a sparse bitmap. Section 4.1 analyzes two methods for translating sparse bitmaps into column accesses, identifies their respective trade-offs and conclude the elementwise one undefperforms often; (3) join and group-by access dimension tables at qualifying row positions, not dense arrays. Section 4.2 fuses all phases in a single kernel and adaptively selects the better method based on runtime conditions. Experimentally, elementwise-style fusion achieves only 1.34× speed, while our VQP-aware fusion achieves 3.19× (Section 8.4).

4.1 Seeking Columns from Bits

After the index access phase produces a result bitmap, the query engine must fetch actual column values for qualifying rows and thus allow subsequent joins and aggregations seamlessly follow existing non-indexed routines. We consider two approaches.

Method 1: Direct Bitmap Scanning. Each thread iterates through its assigned bitmap words. For each word, it scans individual bits and directly fetches column values when a bit is set. This method requires no additional logic beyond a simple loop, consumes no extra registers or shared memory, and keeps the implementation straightforward. However, Method 1 suffers from divergence. A few threads with multiple qualifying rows continue reading column data. Column accesses are uncoalesced as well. Adjacent threads access adjacent *word* positions, which is 32 *bits* apart, and is therefore not coalesced.

Method 2: Materialized Position Array. This method first converts the sparse bitmap into a dense array of qualifying row positions. Each thread counts the number of ones in its bitmap words. A parallel prefix sum across threads determines where each thread should write its positions in the output array. Threads then write their qualifying row positions to this array. When fetching column values, thread t loads `column[positions[t]]`, where `positions` is the dense array.

Method 2 is neither free of uncoalesced loads nor divergence. We have no control over the contents of the position array. However, since positions are sorted, consecutive threads access row indices that are as close together as possible given the data distribution. Any column accesses that *can* coalesce *will* coalesce, and is the theoretical best achieveable for sparse data. Divergence is also reduced: the divergent part is writing positions to the array, cheaper than multiple irregular column loads. The cost is the prefix sum overhead and the memory consumed by the position array itself.

Figure 2 compares the two methods on synthetic data (Zipfian skewness 1.04, see section 8.6 for dataset details) with selectivity ranging from 1/255 (0.39%) to 1/8 (12.5%). Higher selectivity than 1/8 would cause overflow in shared-memory position array. At low selectivity, the two methods are nearly tied because the query phase has little work, making Method 2's *fixed* overhead, i.e., counting each thread's 1-bits and prefix summing, relatively noticeable. As selectivity increases, Method 2's advantage from improved coalescing dominates; at 1/8 selectivity, Method 2 is about 12% faster than Method 1 in our experiment. The crossover occurs at very low selectivity. Since Method 2 tends to win as selectivity grows and the penalty at very low selectivity is small, we prefer Method 2 whenever possible.

4.2 Fusing with Index Access

We now describe how to fuse index access with column seeking, progressing from basic to advanced optimizations. Result bitmaps are backed by vt CUDA registers after the virtual query program executes (see Section 3.2). As a prerequisite to fusion, we first treat the **index access result as the input tile** to the column seeking phase and eliminates one global memory round-trip.

Block-Local Position Array. Standard prefix sum requires a multi-pass algorithm with a "spine" phase aggregating partial sums across blocks. We instead confine the position array within each thread block. Each block independently performs a prefix sum, producing a position array in shared memory that subsequent operations consume without leaving the block.

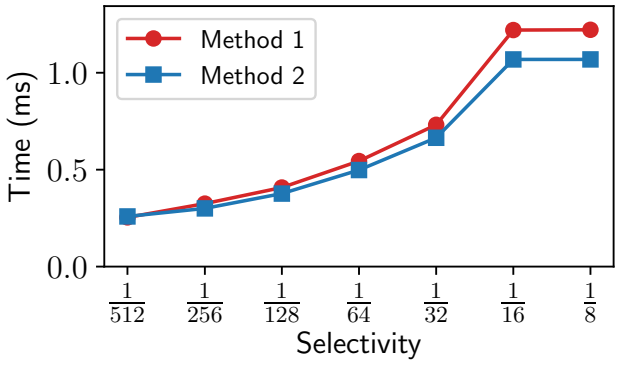

Figure 2: Method 1 (direct scanning) vs. Method 2 (materialized position array) on synthetic workloads with varying selectivity. Higher selectivity causes shared memory overflow in method 2.

This confinement limits the maximum number of qualifying rows per block. A block processing $nt \times vt$ bitmap words covers $32 \times nt \times vt$ rows. At selectivity s, the position array requires $32 \times nt \times vt \times s \times 4$ bytes (4-byte integers). With $nt = 256$ and 12 KB shared memory, Method 2 supports selectivity up to $12288/(32 \times 256 \times vt \times 4) = 0.375/vt$. For $vt = 2$, this yields 18.8%; for $vt = 4$, this yields 9.4%. Both comfortably exceed SSB's maximum selectivity of 3.4%.

Dynamic Per-Block Method Selection. Data skew can cause some blocks to exceed shared memory capacity even when average selectivity is low. We let each block decide independently: perform a cheap popcount over the result bitmap, then choose Method 2 if the position array fits, otherwise fall back to Method 1. This handles skewed distributions gracefully—blocks with low local selectivity use Method 2 even when global selectivity is high.

With these optimizations, the fused kernel executes index access, position materialization, column lookups, and aggregations in a single pass. Section 8 shows that fusing index access with data lookup provides substantial end-to-end speedups over a non-fused baseline that materializes intermediate index results to global memory: on RTX 5090 D (consumer card, `sm_120`) at SF=140 we observe an 1.58× speed from fusion, while on H800 (datacenter card, `sm_90`) at SF=300 the speed is 1.33×.

5 FUSED CANDIDATE CHECKING

This section addresses Limitation 3 from Section 1: handling queries whose predicate boundaries do not align with pre-built index bins. As introduced in Section 1, boundary bins contain both qualifying and non-qualifying rows that cannot be distinguished without checking actual column values—a three-valued outcome of certain-in, certain-out, or uncertain. The challenge on GPU is tracking and propagating this uncertainty through boolean operations without spilling to global memory.

There are two common strategies for handling misaligned boundary bins. *Eager strategy* checks predicate columns immediately

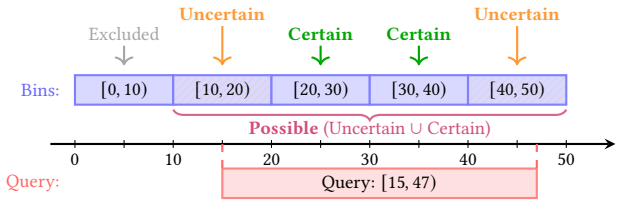

Figure 3: Bitmap index bins vs. query range misalignment. Certain bins $[20, 30)$ and $[30, 40)$ are fully within $[15, 47)$. Uncertain bins $[10, 20)$ and $[40, 50)$ partially overlap and require verification. Possible = Uncertain ∪ Certain.

when an uncertain row is encountered during index access. This can cause excessive and often redundant reads: in a multi-column AND, column A may trigger a verification load for a row that column B would immediately exclude. *Deferred strategy* postpones checking until after all index predicates are applied. This reduces futile reads but typically requires tracking candidates and/or uncertainty *per predicate column*. On GPUs, such per-column bookkeeping quickly becomes the bottleneck: it consumes scarce shared memory quickly and, once exhausted, introduces extra passes over intermediate state.

The eager-deferred trade-off has been studied in other contexts, e.g., interval refinement for predicates [21], designing novel index structures [5] and selective late materialization in (unindexed) columnar engines [17]. However, these approaches do not provide a GPU-friendly mechanism, relying on sequential access patterns. We design a scheme that tracks uncertainty at row granularity, propagate it throughout index access and strike a balance between eager and deferred, detailed in our two-bit representation and its propagation rules below.

Two-Bit Representation. To balance bookkeeping overhead and redundant data accesses, we track two bits per row during index access: *certain* (c) and *uncertain* (u). A row is certain if it definitively qualifies based on index information alone. A row is uncertain if it falls within a boundary bin and requires data verification. We define *possible* as $p = c \vee u$: rows that might qualify. Rows with $p = 0$ are definitively excluded; rows with $p = 1, u = 0$ are definitively included; rows with $p = 1, u = 1$ require candidate checking.

The case without candidate checking (Section 3) is a special case where $u = 0$ everywhere: all bins align perfectly with query boundaries, so every possible row is certain.

Propagation Rules. We derive rules for propagating c and u through boolean operations. Let subscripts denote operands (1, 2) and result (r).

For conjunction (AND), a row is certain in the result only if it is certain in both operands. A row is uncertain if it is possible in both operands but not certain in at least one:

$$c_r = c_1 \wedge c_2 \tag{1}$$

$$u_r = (p_1 \wedge p_2) \wedge \neg c_r = (p_1 \wedge p_2) \wedge (u_1 \vee u_2) \tag{2}$$

For disjunction (OR), a row is certain if it is certain in either operand. A row is uncertain if it is possible but not confirmed certain by either operand:

$$c_r = c_1 \vee c_2 \tag{3}$$

$$u_r = (p_1 \vee p_2) \wedge \neg c_r = (u_1 \wedge \neg c_2) \vee (u_2 \wedge \neg c_1) \tag{4}$$

These rules are integrated implicitly into virtual query program semantics. Each instruction now operates on (p, u) pairs instead of single bitmaps. The `ANDM` instruction loads c from middle bins (fully contained in the query range) and u from boundary bins (leftmost and rightmost), then applies the AND propagation rules. `ORM` operates analogously.

Implementation. Virtual registers now require twice the shared memory: $2 \times n_{\text{reg}} \times nt \times vt$ words per block, where n_{reg} is the number of virtual registers. This doubling arises because the original scheme used one bit to distinguish between definitely excluded and definitely included rows, while our extended scheme adds one more bit to track uncertainty, requiring two bits total per row. Each register stores both p and u bitmaps. The index access phase produces two outputs in r_0: the possible bitmap and the uncertain bitmap.

The column seeking phase (Section 4.2) proceeds as before, using the possible bitmap to identify rows that might qualify. For each qualifying row, we encode whether it is uncertain in the high bit of its position entry. The subsequent query operations receive both the row ID and an "uncertain" flag. When a row is marked uncertain, the operation must verify the predicate against actual column values; when certain, it can skip verification.

Overhead Analysis. This approach is not perfect, but rather a balance between eagerness and laziness: since we track only aggregate uncertainty (not per-column), we must access every predicate column for uncertain rows even if only one column caused the uncertainty, unlike fully deferred lookup. It is a price we pay for operating fully on-chip. The doubled shared memory requirement reduces either the number of virtual registers or vt; Section 6 quantifies this trade-off. The expected fraction of uncertain rows is bounded by $s \cdot 2k/b$, where s is selectivity, k is the number of indexed columns, and b is the bin count per column. For our Sparse layout ($b \approx 6$, $k = 3$), roughly as many rows need verification as pass the filter. In SSB selectivities are 0.02–3.4% and thus the absolute count is small. This is consistent with the 8.8% overhead we observe at SF=140 (Section 8.5).

Section 8.5 shows that worst-case all-unaligned candidate checking is only modestly slow over aligned boundaries: full checking is 8.8% slower than no checking on geometric mean at SF=140 (and 16.4% slower at SF=300 on H800). Even with this overhead included, we still substantially outperform the non-indexed join and previous perfectly-aligned indexed baseline.

6 DISCUSSION

6.1 Shared Memory Budget

The techniques presented in Sections 3–5 share a common resource: shared memory. Here we analyze how to balance competing demands and provides concrete configuration guidelines. With nt threads per block and vt values per thread, virtual registers consume $n_{\text{reg}} \times nt \times vt \times 4$ bytes without candidate checking, or $2 \times n_{\text{reg}} \times nt \times vt \times 4$ bytes with candidate checking (Section 5). On consumer GPUs (all recent generations of RTX series), achieving 100% occupancy limits shared memory to approximately 12 KB per

block. With $nt = 256$ and $n_{\text{reg}} = 3$ (sufficient for complex queries per Section 3.1): without candidate checking, $3 \times 256 \times vt \times 4 \leq 12288$ yields $vt \leq 4$; with candidate checking, $6 \times 256 \times vt \times 4 \leq 12288$ yields $vt \leq 2$.

ILP vs. Method 2 Selectivity. The parameter vt controls instruction level parallelism: larger vt keeps the GPU's execution units busier by providing more independent operations per thread. However, vt also affects Method 2's applicability (Section 4.2). Each block processes $32 \times nt \times vt$ rows; at selectivity s, the position array requires $32 \times nt \times vt \times s \times 4$ bytes. With 12 KB available after freeing virtual registers, $vt = 2$ supports selectivity up to 18.8% and $vt = 4$ up to 9.4%. SSB's maximum selectivity is 3.4%, so both configurations suffice. For workloads with higher selectivity, reducing vt or relying on Method 1 fallback (Section 4.2) may be necessary.

Table 1: Shared memory configuration trade-offs across GPU tiers ($n_{\text{reg}} = 3$, $nt = 256$).

GPU Tier	SMEM	Cand. Chk?	vt	ILP	Max Sel.
Consumer	12 KB	No	4	High	9.4%
Consumer	12 KB	Yes	2	Medium	18.8%
Datacenter	19 KB	No	6	Very High	14.1%
Datacenter	19 KB	Yes	3	High	28.1%
Datacenter*	26 KB	Yes	4	High	37.5%

* At 75% occupancy (acceptable for memory-bound workloads).

For consumer GPUs with 12 KB shared memory, we recommend $n_{\text{reg}} = 3$ and $vt = 4$ without candidate checking, or $vt = 2$ with candidate checking. Three registers express queries of substantial complexity (Section 3.1), $vt = 4$ provides adequate ILP, and the 9.4% selectivity limit covers typical analytical workloads.

On the other hand, at 100% occupancy, datacenter GPUs provide approximately 19 KB shared memory; at 75% occupancy (acceptable for memory-bound workloads), this increases to 26 KB. With 19 KB and 3 registers, without candidate checking $vt \leq 6$ and with checking $vt \leq 3$. The higher vt improves ILP and partially compensates for reduced occupancy. At 26 KB with 75% occupancy, $vt = 4$ becomes feasible even with candidate checking, matching the non-checking consumer configuration. Our implementation fixes $n_{\text{reg}} = 3$ based on the observation that SSB queries (and most analytical queries) do not require deeper nesting. For workloads requiring more registers, reducing vt accommodates additional registers at the cost of ILP.

6.2 Compatibility with Frontend Strategies

Our backend exposes a compact, explicit interface: the frontend is expected to provide (i) a set of bitmap operands (bins), which is common for backends to request; and (ii) an instruction sequence describing how to combine them. This decouples *plan selection* (e.g., operand ordering and binning choices) from *plan execution* (a single fused kernel evaluating the resulting program) and make our backend compatible to varieties of frontend strategies:

Predicate ordering. Since bitmap AND/OR are commutative (and associative within an AND/OR group), the frontend can reorder operands to match classic selectivity-based heuristics [34] and feed *reordered* execution plan directly into our backend. Reordering increases effectiveness of our memory-operand instructions (Section 3.3) that skip bin loads when long sequences of 0 appear in any stage of execution.

Binning strategies. Techniques that adjust bin boundaries for a workload (e.g., to reduce boundary overlap) [28] are complementary to our candidate checking. Boundary tuning reduces the rate of uncertainty by shrinking partially-overlapping bins, but bins can be processed identically afterwards. The number of bins per column presents a clear trade-off: more bins improve filtering precision (fewer rows fall in boundary bins) but increase the number of bitmaps that must be ORed together during index access, consuming more memory bandwidth. Fewer bins reduce memory usage and OR cost but increase boundary-bin rows requiring candidate checking. Our evaluation (Section 8) explores this trade-off empirically with three fixed layouts (Sparse, Medium, Dense) that span the practical range and show small number of bins provide best performance-to-memory ratio.

Other frontend techniques, like runtime profiling based query plan generation [42] and query-aware prefetching [43] are largely orthogonal. Such techniques are applied to the underlying column storage and data movement policies and is unrelated to our index execution model. We now list several similar orthogonal concerns that lie outside the scope of this work but are worth mentioning.

6.3 Orthogonal Concerns

Modularity and Composability. Kernel fusion often cripples modularity. Our implementation addresses modularity at two levels. At compile time, the fused kernel is a C++ template parameterized by launch parameters and device function lambdas composed at compile time (i.e. Crystal / NVIDIA CUB style). At runtime, the downstream function accepts a boolean `chk` parameter controlling whether uncertain rows are verified, so the same kernel handles both aligned and misaligned cases. Current implementation targets star/snowflake style group aggregation; supporting additional operator types would require new device function templates.

Multi-GPU scaling. Bitmap-like indexes are generally stored denormalized [19] and, in this way, inherently parallelizable: each bit corresponds to exactly one fact-table row, so indexes can be trivially partitioned across devices by row ranges. Multi-GPU query execution is primarily a data sharding and communication problem on the query side [9, 18, 25]; our per-device indexing techniques apply unchanged to each partition.

Index updates. We use uncompressed bitmaps (Section 7), which support $O(1)$ single-row updates by flipping the relevant bit. For bulk updates and more sophisticated maintenance strategies (e.g., delta structures, deferred reconstruction), we refer readers to UpBit [4] and Concurrent Updatable Bitmap Indexing [38]. To our knowledge, no prior work addresses GPU-specific update optimizations; this remains an open problem orthogonal to our query-time contributions.

7 DATABASE INTEGRATION

Baseline Optimizations. Given the lack of database integration in previous GPU indexing works and the fusion techniques described in previous sections, we were naturally interested in demonstrating

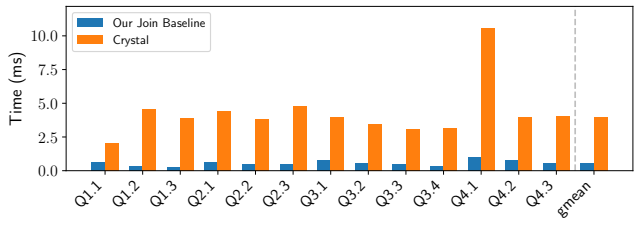

Figure 4: Our join baseline vs. Crystal on SSB SF=20.

usability in a full system. As a proof of concept, we integrated our approach with Crystal [30], a fully fused query backend without indexing support. We optimized as many chokepoints in Crystal as possible. We applied Variable Byte integer storage: columns with values fitting in 8 or 16 bits use `int8` or `int16` types instead of 32-bit integers. Unlike bit-packing schemes [31], native integer types incur no decoding overhead compared to bit packing while retaining most memory traffic reduction capabilities [39]. This allows SF=140 to fit in GPU memory on our platform (32GB VRAM), whereas vanilla Crystal only fits SF=50. We also applied predicated loads [6] and efficient group aggregation techniques [8, 40].

Figure 4 compares our optimized join against vanilla Crystal at SF=20. We achieve 3.31× geometric mean speed, ranging from 1.88× (Q3.1) to 5.58× (Q4.1). The largest gains come from Q4.x queries, which involve more complex joins and aggregations where newly-added techniques shine the most. We implement fused bitmap indexing techniques described in previous technical sections (Section 3.1, 4.2, and 5) upon this optimized baseline. This results in a fused, full-featured GPU query backend supporting a pipeline from index access to final result output in a single kernel, which is the first of its kind to our knowledge.

Backend Interface. We assume that the frontend populates the virtual query program by placing pointers to relevant bins, a straightforward procedural task. The query interface resembles Crystal: both accept a device function that evaluates predicates and returns values for aggregation. Frontend techniques in Section 6.2 are accepted: one can strategically bin columns beforehand, and based on runtime statistics, reorders predicates and produces their results in the form of optimized virtual query program.

Index Creation, Update, and Deletion. Since our prototype uses uncompressed bitmaps, updates are trivial: flip the relevant bits. Deletion reconstructs affected bitmaps or uses delta structures to defer reconstruction [16, 38]. For creation, we fuse materialization of all bins: a single pass over the data populates all indexes simultaneously. This contrasts with compressed schemes that require separate encoding passes. Our approach trades comparable memory usage (within 1% for Sparse layout) for 4.4× construction speed and simpler update operations that avoid decompression overhead. Section 8 compares construction time and memory usage against compressed GPU bitmaps.

8 EXPERIMENTAL EVALUATION

Datasets. We use the Star Schema Benchmark (SSB) [24] at scale factors (SF) 20 and 140, comprising 120M and 840M rows respectively. SF=20 is necessary since baselines (e.g. RTScan) only support SF=20 due to their implementation details. For the server-GPU study on H800 (Section 8.7), we additionally run SSB at SF=300. SSB contains 13 queries across 4 query flights with selectivities ranging from 0.02% to 3.4%. We focus on SSB because Crystal, the fastest GPU query backend (Section 2.3), is limited to star/snowflake schema joins; extending to more complex query patterns (e.g., TPC-H) would require reimplementing the query engine itself, which is orthogonal to our indexing contributions. For controlled experiments, we generate synthetic data with 840M rows (matching SF=140), varying selectivity from 1/256 (0.39%) to 4/5 (80%) and Zipfian skewness from 1.04 to 1.20. The synthetic schema consists of a fact table with two 16-bit or 32-bit attributes and a foreign key to a dimension table; the dimension table has a 32-bit filter attribute and an 8-bit group-by attribute (256 groups). The query evaluates $(\texttt{fa}_1 \in [lo_1, hi_1) \vee \texttt{fa}_2 \in [lo_2, hi_2)) \wedge \texttt{da}_1 \in [lo_3, hi_3)$ with a foreign-key join and group-by aggregation, where `fa` and `da` denote fact and dimension tables respectively.

Bin Configuration. We evaluate three fixed bin layouts, all created offline without query knowledge. Each layout is a superset of the previous, so coarser bins remain accessible for long-spanning predicates. **Sparse** builds one bin per year for date, per continent for city columns and equal-width bins for others, averaging 5–7 bins per column. **Medium** adds finer equi-width bins and country-level city bins (25 countries). Medium layout retains sparse binning on date and supplier city where finer granularity adds little, whereas **Dense** layout further adds those bins.

The three fixed layouts serve as a bin-count experiment: Sparse uses 5–7 bins per column, Medium uses 8–25, and Dense uses up to 80. Sparse and Medium are practical operating points; Dense trades memory for additional speedup. Notably, denser layouts are *built on top of* sparser ones, and thus avoids merging many bitmaps on wide-spanning predicates.

Competitors. We compare against **WAH** [22, 36, 37] (GPU Word-Aligned Hybrid compressed bitmaps; since WAH does not support candidate checking, we give it perfect bin alignment, aka. best case), **RTScan** [19] (ray-tracing-based index; conjunctive selections only, SF=20 only due to memory limits), and **Join** (our optimized non-indexed baseline, Section 7).

Setup. Unless otherwise noted, experiments run on a system with an NVIDIA RTX 5090 D GPU connected to an AMD EPYC 9654 CPU (96 total cores, 24 schedulable in our instance), running on CUDA 13.0. We additionally evaluate on a server GPU platform with an NVIDIA H800 PCIe 80GB GPU (2 TB/s bandwidth) connected to a Xeon Platinum 8458P CPU (44 cores, 20 schedulable cores); we report these results in Section 8.7. All measurements exclude data transfer time (data reside in GPU memory) and report the average of 100 runs after 1 warmup run.

8.1 Overall Performance

Figure 5 compares execution time across all 13 SSB queries at SF=20. We show all three bin layouts (Sparse, Medium, and Dense) for our approach. We also include WAH-compressed bitmaps (perfect alignment), our optimized non-indexed baseline (Join), and RTScan. In WAH, we add the materialization overhead and table lookup time since it only produces intermediate bitmaps. RTScan returns row IDs directly so only table lookup time is added.

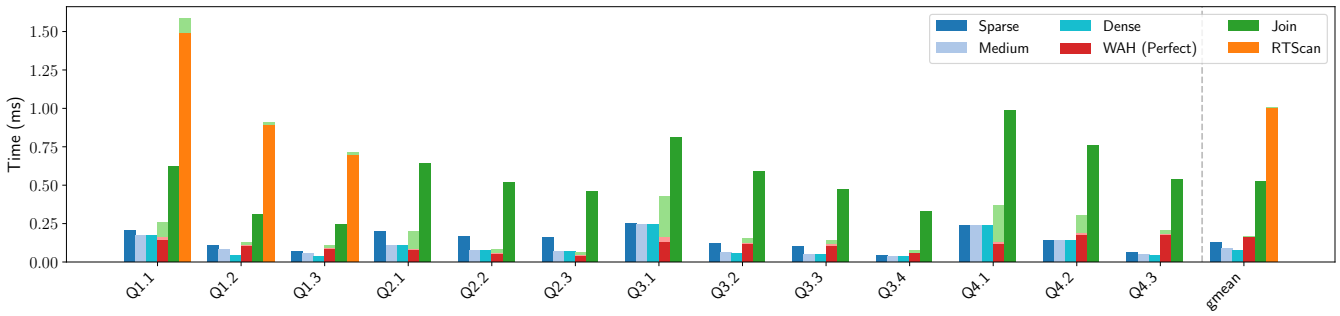

Figure 5: Query execution time on SSB at SF=20. Three bin layouts (Sparse, Medium, Dense) are shown for our approach. For WAH, we add materialization and table lookup time to the index access time since WAH only produces bitmaps. WAH uses perfect bin alignment (best case). RTScan is limited to Q1.x (conjunctive selections).

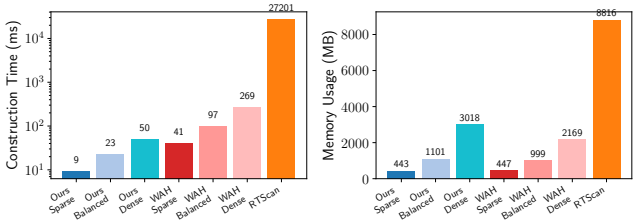

Figure 6: Index construction time and memory usage at SF=20. Three bin layouts shown for both our approach and WAH.

Our Sparse layout achieves a geometric mean of 0.127 ms, outperforming WAH's best-case (0.164 ms, 1.30×) and non indexed join (0.522 ms, 4.11×). Medium and Dense layouts are faster still (0.088 ms and 0.078 ms). Denser binning does improve query performance at the cost of memory. Memory usage details in Section 8.2. RTScan achieves uncompetitive performance on SF=20 Q1.x queries (0.35–0.56 ms) and cannot express queries requiring disjunctions or joins (Q2.x–Q4.x).

8.2 Index Creation and Memory Usage

We compare index construction time and memory footprint at SF=20 (Figure 6) because the temporaries allocated by RTScan will exceed our 32G GPU memory budget when using higher SF.

For the Sparse layout, our uncompressed bitmaps construct with 4.4× speed of WAH (9.3 ms vs 41.0 ms) with comparable memory (443 MB vs 447 MB). The Medium layout (22.6 ms, 1101 MB) and Dense layout (50.4 ms, 3018 MB) trade more construction time and memory for the query-time speedups shown in Section 8.1. Notably, despite using 2× and 7× memory respectively, Medium and Dense binning do not provide 2× speeds compared to Sparse binning. We experimentally conclude that a small number of bins captures most of the filtering benefit, and too fine-grained binning may bring diminishing returns. We therefore recommend memory-saving sparse binning in practical GPU bitmap indexing deployments.

WAH construction scales similarly: 96.7 ms (Medium) and 268.9 ms (Dense). In all configurations, our construction is faster than WAH because we simply scatter bits into pre-allocated arrays while WAH must identify and encode runs, and we create all bins in a single data pass (Section 7). RTScan is orders of magnitude more expensive to build: 27.2 s, roughly 2900× slower than our Sparse approach, and it uses substantially more memory (8.8 GB) due to its ray-based spatial indexing structure storing 3 32b floats per data point.

8.3 Scalability

Figure 7 shows results at SF=140 (840M rows). RTScan is excluded because it exceeds GPU memory. Our Sparse layout achieves a geometric mean of 0.836 ms, 4.31× the speed of unindexed join (3.604 ms) and 1.40× of best-case WAH (1.168 ms). Medium (0.579 ms) and Dense (0.508 ms) provide further speedups. The proportional relationships remain consistent with SF=20: 7× more rows yield approximately 6.6× longer execution. Dense and Medium converge on queries where the Sparse layout already covers most bins (e.g., Q3.1, Q4.1–Q4.2), while the gap is largest on queries with many boundary bins (e.g., Q1.2, Q3.4, Q4.3).

The consistent speedup ordering (Dense > Medium > Sparse) confirms that more bins improve filtering precision, while the modest gaps (Dense is 1.64× the speed of Sparse at SF=140 geometric mean) confirm our conclusion in Section 8.2 that even a small number of bins capturing most of the filtering benefit.

8.4 Ablation: Fusion Progression

Figure 8 shows five execution modes (Sparse layout): (1) **No-fuse**: three separate kernels for index access, materialization, and downstream query. (2) **Naive merge**: all stages in one kernel without VQP or downstream optimizations. It is the elementwise-style fusion tensor frameworks would produce if this pipeline were expressible as tensor programs. (3) **VQP-only**: virtual program for index access, but components remain separate. (4) **Ours**: fully fused with VQP, results in registers throughout. (5) **Dedicated**: hand-written compile-time kernel as performance ceiling.

Across all 13 SSB queries (geometric mean), three separate kernels without virtual program take 2.668 ms. The elementwise naively merged kernel reduces this to 1.988 ms (1.34×). Adding the virtual program to the separate-kernel baseline brings it to 1.322 ms. Our fully fused kernel achieves 0.836 ms, a 3.19× speed over the three-kernel baseline. The gap between naive merging and our approach (2.38×) demonstrates that elementwise-style fusion is insufficient: it

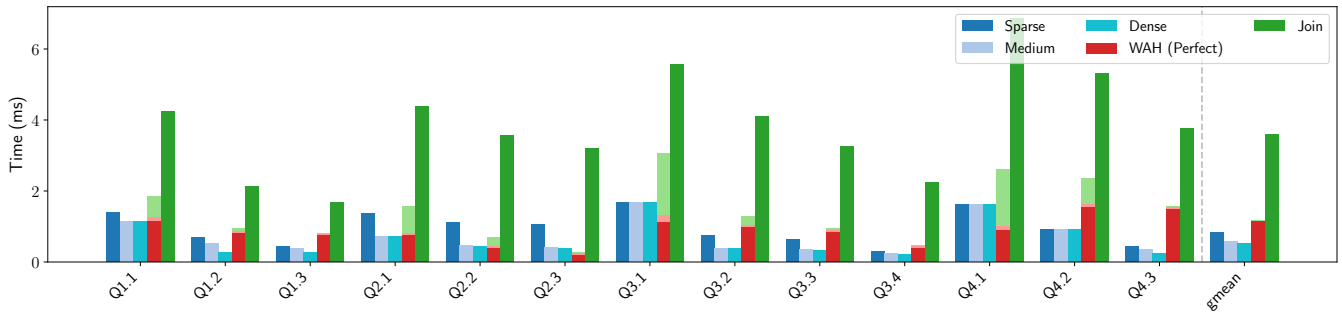

Figure 7: Query execution time on SSB at SF=140. Three bin layouts shown for our approach. WAH uses perfect bin alignment (best case). RTScan exceeds GPU memory at this scale.

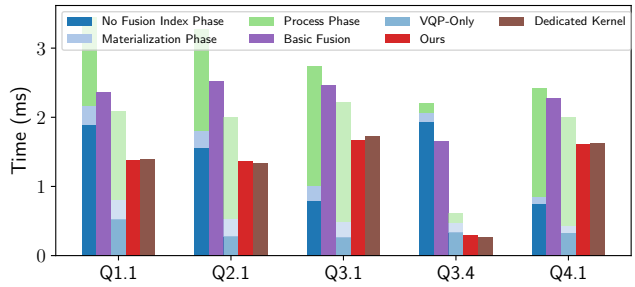

Figure 8: Fusion progression on representative SSB queries (SF=140, Sparse layout). Five execution modes from left to right: (1) no fusion at all (stacked bars showing index, materialize, and process phases); (2) all stages naively merged into one kernel; (3) three separate kernels (stacked) with virtual program; (4) our fully fused kernel; (5) hand-written compile-time kernel as performance ceiling.

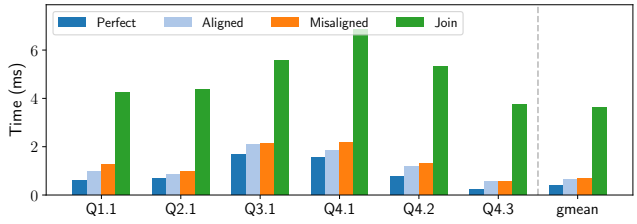

Figure 9: Impact of bin alignment on query execution time at SF=140. Rightmost bar shows geometric mean across all 13 SSB queries.

cannot eliminate the sparse-to-dense materialization step or exploit the register-level data flow that our design enables. The compile-time kernel (0.829 ms) shows our virtual program overhead is only 0.8%, confirming that runtime interpretation adds negligible cost.

8.5 Ablation: Alignment Scenarios

To isolate the impact of candidate checking, we evaluate three controlled alignment scenarios at SF=140 using query-aware fixed-bin configurations. These controlled scenarios quantify misalignment overhead ceiling and complement the fixed-layout results in Sections 8.1–8.3:

- **Perfect**: Bin boundaries exactly match query predicates (best case, no candidate checking).
- **Aligned**: 4 bins per predicate but boundaries align (common case, no candidate checking).
- **Misaligned**: 4 bins with all left and right misaligned (worst case, requires candidate checking).

Figure 9 shows the results. Aligned, the common case where queries span 4 bins with aligned boundaries, achieves 0.624 ms geometric mean. Perfect alignment (0.380 ms) represents an unrealistic best case using exactly one bin per predicate with no OR-ing of multiple bins. It is always faster than Aligned on every single case due to strictly less work done and is 39.1% faster. The key metric is Misaligned overhead over Aligned: only 8.8% slower (0.679 ms vs 0.624 ms), demonstrating that candidate checking adds modest cost even in worst-case misalignment. All scenarios substantially outperform the non-indexed Join baseline (3.604 ms): Aligned is 5.78× its speed and even worst-case Misaligned is 5.31×.

8.6 Synthetic Workloads

To understand sensitivity to workload characteristics independent of misalignment overhead, we use synthetic data (840M rows, same row count as SF=140) with a *controlled* bin layout: a fixed bin count per column, with three alignment scenarios (Perfect, Aligned, Misaligned) as defined in Section 8.5. This same configuration is used across all selectivity and skewness levels.

Selectivity Sensitivity. Figure 10 top-left panel varies selectivity from 1/256 (0.39%) to 4/5 (80%) at fixed skewness 1.12. At low selectivity, Misaligned consistently outperforms WAH's best-case Perfect alignment (0.45× at 1/256, 0.65× at 1/64). As selectivity increases, Misaligned time grows due to a) more rows passing filtering as uncertain bins, thus boosting *index phase intermediate selectivity*, and b) more blocks falling back to Method 1. Join (6.69 ms) outperforms Misaligned (9.61 ms) at selectivity 4/5 skewness 1.16, and is neck-and-neck at selectivity 2/5 skewness 1.12 (top-left Figure 10). Above 1/4 selectivity and 1.12 skewness, Join becomes competitive and eventually overtakes indexed execution, because boundary bins inflate effective bin selectivity beyond the nominal value. Perfect and multi-bin Aligned configurations converges below Join even at

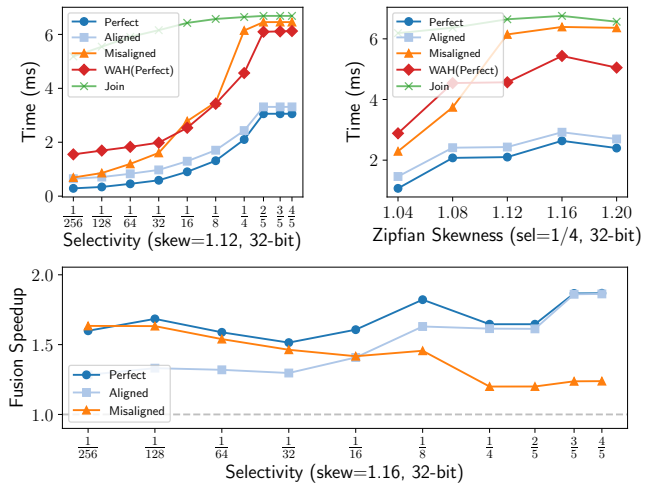

Figure 10: Synthetic workload performance. Top: selectivity (left) and skewness (right) sensitivity. Bottom: after vs. before speed (non-fused / fused) across selectivity levels.

high selectivity as aligned filtering not needing candidate checking remains beneficial.

Skewness Sensitivity. The top-right panel varies Zipfian skewness from 1.04 to 1.20 at fixed selectivity 1/32. Query predicates are drawn uniformly over the value domain, so skew affects how many rows fall in boundary bins rather than the nominal selectivity. At low skew (1.04), Misaligned (2.29 ms) comfortably outperforms WAH (2.89 ms). As skew increases, popular values concentrate in fewer bins, causing Misaligned time to grow sharply (3.87 ms at skew 1.12, 6.36 ms at 1.20) while WAH remains stable around 2.97 ms because its perfect bins are unaffected by value distribution. At skew 1.12 and above, Misaligned exceeds WAH. The skew sensitivity is driven by boundary bins: skewed distributions place more rows in boundary bins, increasing candidate checking workload. Perfect and Aligned configurations, which avoid boundary bins, remain stable across all skewness levels.

Fusion Speedup. Bottom panel shows fusion speedup (non-fused / fused) across selectivity levels. Perfect achieves 1.5–1.7× and Misaligned 1.4–1.7×. At low selectivity, fixed cost of materialization is more prominent, leading to larger fusion benefits. As selectivity increases, more work shifts into candidate checking and column access; fusion speedup correspondingly tapers but remains visible.

8.7 Server GPU (H800) Results

To understand whether our design remains effective across GPU architectures, we repeat the key experiments on an NVIDIA H800 PCIe 80GB (`sm_90`). We expand scale factor to SF=300.

Overall Performance. Figure 11 shows overall performance at SF=300. Our Sparse layout achieves a geometric mean of 2.377 ms, 4.54× the speed of non indexed join (10.794 ms) and 1.21× of best-case WAH (2.872 ms). Medium (1.710 ms) and Dense (1.520 ms) provide further speedups, consistent with the consumer GPU trends.

Fusion Progression. Figure 12 shows the fusion progression on H800. No-fuse to Ours achieves 2.66× (vs 3.19× on RTX 5090 D); naive merge to Ours achieves 1.99× (vs 2.38×). Notably, naive merging achieves nearly identical gains to no fusionn on both platforms (1.33× vs 1.34×). The difference is specific to VQP fusion: the consumer GPU achieves 1.58× fusion speed versus 1.33× on H800. This reflects the H800's tighter register budget (100% occupancy at 32 registers/thread vs 40 on the consumer GPU), which limits the fused kernel's ability to keep execution units on-chip. VQP overhead is 4.6% vs 0.8% on the consumer GPU for the same reason.

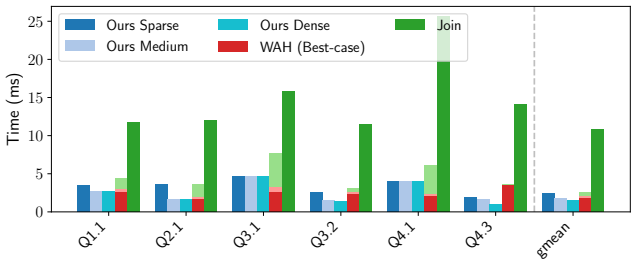

Figure 11: Query execution time on SSB at SF=300 on the H800 (selected queries). Three bin layouts shown for our approach. WAH uses perfect bin alignment (best case).

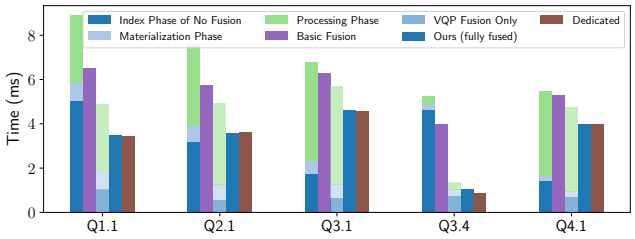

Figure 12: Fusion progression at SF=300 on the H800, following the same structure as Figure 8.

9 CONCLUSION

We present a fused bitmap indexing approach for GPU query engines that supports arbitrary boolean predicates, eliminates pipeline breakers between index access and query execution, and remains robust under misaligned query boundaries via a two-bit candidate checking scheme. On practical bin layouts, we achieve 4.31× speed over an optimized non-indexed baseline and 1.40× over WAH at best-case alignment at SF=140; our VQP-specific fusion reaches 3.19× versus 1.34× for elementwise-style fusion. Future work includes extending to more complex operator types and adaptive binning strategies.

ACKNOWLEDGMENTS

This work was supported by the Special Project of the Key Laboratory of System Software (Chinese Academy of Sciences) (No. ZR202601) and the Beijing Municipal Natural Science Foundation (No. 4252024).

REFERENCES

[1] Daniel J. Abadi. 2007. Column Stores for Wide and Sparse Data. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:3261330

[2] Witold Andrzejewski and Robert Wrembel. 2010. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *International Conference on Database and Expert Systems Applications*. Springer, 315–329.

[3] Witold Andrzejewski and Robert Wrembel. 2011. GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps. *Control and cybernetics* 40, 3 (2011), 627–650.

[4] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. Upbit: Scalable in-memory updatable bitmap indexing. In *Proceedings of the 2016 International Conference on Management of Data*. 1319–1332.

[5] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. 2019. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2019), 1296–1311. https://doi.org/10.1109/TKDE.2018.2825349

[6] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454.

[7] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 355–366. https://doi.org/10.1145/276304.276336

[8] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, Yushi Guan, Christina Giannoula, and Nandita Vijaykumar. 2025. ARC: Warp-level Adaptive Atomic Reduction in GPUs to Accelerate Differentiable Rendering. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 64–83. https://doi.org/10.1145/3669940.3707238

[9] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs.. In *ADMS@ VLDB*. 55–64.

[10] Bala Gurumurthy, David Broneske, Martin Schäler, Thilo Pionteck, and Gunter Saake. 2021. An Investigation of Atomic Synchronization for Sort-Based Group-By Aggregation on GPUs. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. 48–53. https://doi.org/10.1109/ICDEW53142.2021.00016

[11] 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. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2811–2825.

[12] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16, 13 (9 2023), 4268–4281. https://doi.org/10.14778/3625054.3625063

[13] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (March 2023), 81 pages. https://doi.org/10.1145/3570638

[14] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. *ACM SIGPLAN Notices* 48, 4 (2013), 395–406.

[15] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[16] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience* 46, 11 (2016), 1547–1569.

[17] Yihao Liu, Shaoxuan Tang, Yulong Hui, Hangrui Zhou, and Huanchen Zhang. 2025. Selective Late Materialization in Modern Analytical Databases. *Proc. VLDB Endow.* 18, 11 (July 2025), 4616–4628. https://doi.org/10.14778/3749646.3749717

[18] 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

[19] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. 2024. RTScan: Efficient Scan with Ray Tracing Cores. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1460–1472.

[20] Antonio Mallia, Michał Siedlaczek, Torsten Suel, and Mohamed Zahran. 2019. GPU-accelerated decoding of integer lists. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. ACM, Beijing, China, 2193–2196.

[21] Marcelo M. Menezes, Salles Viana Gomes Magalhães, Matheus Aguilar de Oliveira, and Rodrigo Eduardo de Oliveira Bauer Chichorro. 2019. Accelerating the Exact Evaluation of Geometric Predicates with GPUs. https://api.semanticscholar.org/CorpusID:199529950

[22] Mitchell Nelson, Zachary Sorenson, Joseph M Myre, Jason Sawin, and David Chiu. 2019. GPU acceleration of range queries over large data sets. In *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. 11–20.

[23] John Nickolls. 2007. GPU parallel computing architecture and CUDA programming model. In *2007 IEEE Hot Chips 19 Symposium (HCS)*. IEEE, 1–12.

[24] Patrick O'Neil, Betty O'Neil, and Xuedong Chen. 2009. *Star Schema Benchmark*. Technical Report. UMass Boston. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF

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

[26] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[27] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science-Towards Embedded Analytics.. In *CIDR*.

[28] D. Rotem, K. Stockinger, and Kesheng Wu. 2006. Minimizing I/O Costs of Multi-Dimensional Queries with Bitmap Indices. In *18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*. 33–44. https://doi.org/10.1109/SSDBM.2006.33

[29] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th international conference on scientific and statistical database management*. 1–12.

[30] 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*. 1617–1632.

[31] 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*. ACM, Philadelphia, PA, USA, 1390–1403.

[32] Rishi Rakesh Sinha, Marianne Winslett, Kesheng Wu, Kurt Stockinger, and Arie Shoshani. 2008. Adaptive Bitmap Indexes for Space-Constrained Systems. In *2008 IEEE 24th International Conference on Data Engineering*. 1418–1420. https://doi.org/10.1109/ICDE.2008.4497575

[33] 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

[34] Evangelia A. Sitaridi and Kenneth A. Ross. 2016. GPU-accelerated string matching for database applications. *The VLDB Journal* 25, 5 (Oct. 2016), 719–740. https://doi.org/10.1007/s00778-015-0409-y

[35] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. 2004. Evaluation Strategies for Bitmap Indices with Binning. In *Database and Expert Systems Applications: 15th International Conference, DEXA 2004, Zaragoza, Spain, August 30-September 3, 2004. Proceedings*. Springer Berlin Heidelberg.

[36] Brandon Tran, Brennan Schaffner, Joseph M Myre, Jason Sawin, and David Chiu. 2021. Exploring means to enhance the efficiency of GPU bitmap index query processing. *Data Science and Engineering* 6, 2 (2021), 209–228.

[37] Brandon Tran, Brennan Schaffner, Jason Sawin, Joseph M Myre, and David Chiu. 2020. Increasing the efficiency of GPU bitmap index query processing. In *International Conference on Database Systems for Advanced Applications*. Springer, 339–355.

[38] Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing. *Proceedings of the VLDB Endowment* 18, 2 (2024), 399–412.

[39] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 993–1008.

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

[41] Kesheng Wu, Kurt Stockinger, and Arie Shoshani. 2008. Breaking the Curse of Cardinality on Bitmap Indexes. In *Proceedings of the 20th International Conference on Scientific and Statistical Database Management* (Hong Kong, China) *(SSDBM '08)*. Springer-Verlag, Berlin, Heidelberg, 348–365. https://doi.org/10.1007/978-3-540-69497-7_23

[42] Maryann Xue, Yingyi Bu, and Abhishek et al. Somani. 2024. Adaptive and Robust Query Execution for Lakehouses at Scale. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3947–3959. https://doi.org/10.14778/3685800.3685818

[43] Zihang Yao, Rong Chen, Binyu Zang, and Haibo Chen. 2022. Wukong+G: Fast and Concurrent RDF Query Processing Using RDMA-Assisted GPU Graph Exploration. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2022), 1619–1635. https://doi.org/10.1109/TPDS.2021.3121568