RasterDB: Implementing Efficient Index Update Pipeline

Akash Maji, 24212

15th January 2026

Mid-Term MTech Project Report

Abstract

Recent work has demonstrated that GPU rasterization pipelines can be effectively leveraged for database column indexing, achieving significant performance improvements over ray tracing–based approaches by relying on simpler arithmetic comparisons. However, existing rasterization-based indexing designs primarily focus on static index construction and query execution, leaving index maintenance operations largely unexplored.

In this work, we extend the rasterization-based indexing paradigm to support dynamic index updates, including point deletions and insertions. We introduce *CompactScanIndex*, a GPU-accelerated index structure that maintains the computational efficiency of rasterization-based indexing, with slight overheads, while enabling incremental modifications without requiring full index rebuilds.

1 INTRODUCTION

Recent architectural advances have significantly improved the performance of Graphics Processing Units (GPUs), equipping them with specialized hardware for massive parallelism and efficient memory access. These capabilities have made GPUs increasingly attractive for accelerating data-intensive workloads beyond graphics, including column indexing and query processing in database systems [4] [5].

Prior GPU-based indexing approaches primarily focus on index construction and query execution, but do not support in-place updates, requiring full index rebuilds upon data modification and incurring additional data transfer and rebuild overheads. In this work, we demonstrate that GPUs can efficiently support dynamic index updates. Our design adopts a uniform bin-based storage layout with atomic operations to safely enable concurrent updates on the GPU, supporting deletions via linear bin scans with $O(bin\_size)$ complexity and insertions with constant-time $O(1)$ cost per point. We call it *CompactScan* as we can do compaction after deletes or updates to reclaim space, and scale the index structure based on available GPU memory.

2 RELATED WORK

2.1 Ray Tracing for Database Indexing

Ray tracing (RT)–based database indexing adapts graphics rendering pipelines by modeling column values as geometric primitives and executing search predicates as ray–primitive intersection tests. Systems such as RTIndex and RTScan [3] demonstrate the feasibility of RT-based indexing. These approaches rely on a *Bounding Volume Hierarchy* (BVH), a tree-based spatial structure conceptually similar to R-trees, to accelerate ray traversal. While effective for general graphics workloads, RT introduces substantial overheads for database indexing, where data values are regularly structured and query predicates are inherently axis-aligned. They suffer from load imbalance, irregular memory access, and control-flow divergence. Despite several optimizations, including data encoding, primitive substitution, and improved ray decomposition, RT-based indexing remains ill-suited for database workloads due to its reliance on general-purpose geometric processing.

2.2 Rasterization for Database Indexing

Recent work by Doraiswamy and Haritsa demonstrated that GPU rasterization pipelines can be effectively repurposed for database column indexing [1]. Their *RasterScan* approach maps multi-dimensional data points onto a two-dimensional grid structure and leverages the GPU's native rasterization hardware to efficiently construct indexes and execute range queries.

The key insight behind RasterScan is that rasterization relies primarily on simple arithmetic value comparisons, which are significantly cheaper than the geometric intersection tests employed by ray tracing-based methods. As a result, RasterScan achieves order-of-magnitude improvements in both index construction and query execution performance compared to ray tracing counterparts.

3 MOTIVATION

3.1 The Need for Index Updates

Despite its impressive performance benefits, RasterScan is fundamentally designed for static workloads, where the indexed data remains unchanged after construction. However, real-world database systems operate in dynamic environments, where records are continuously inserted, updated, and deleted. Supporting such workloads requires corresponding modifications to the underlying index structures. A naive strategy of rebuilding the entire index after each modification is prohibitively expensive, especially for large-scale datasets. Consequently, efficient mechanisms for incremental index maintenance are essential for making rasterization-based indexing practical for real-world database deployments.

4 IMPLEMENTATION DETAILS

This section describes the GPU-accelerated implementation of `CompactScanIndex`, a 2D-index designed for efficient range queries and in-place updates. The implementation leverages Vulkan compute and graphics pipelines to achieve high throughput on modern GPUs. It can support 2D and 3D columns (we just drop the third dimension z for the calculation of the bin location). For more details and visuals, refer to *RasterScan* paper [1].

4.1 Index Structure

The index partitions the 3D input data space into a grid of $R \times R$ bins, where R is the `INDEX_RESOLUTION` hyperparameter (default 1024). Each point (x, y, z) maps to bin (b_x, b_y) where:

$$b_x = \left\lfloor \frac{(x - x_{min}) \cdot R}{x_{max} - x_{min} + 1} \right\rfloor, \quad b_y = \left\lfloor \frac{(y - y_{min}) \cdot R}{y_{max} - y_{min} + 1} \right\rfloor \tag{1}$$

The index maintains the following GPU buffers:
startAddrBuffer: It stores the starting offset of each bin in the data buffer (R^2 entries).
capacityBuffer: It has maximum capacity allocated for each bin, used for overflow detection (R^2 entries).
extentBuffer: It stores current number of used entries in each bin out of the bin capacity(R^2 entries).
dataBuffer: A contiguous storage for all indexed points, organized by bin. It stores (x, y, z, rowID) as entries.

The `dataBuffer` size is computed as:

$$Size = N \times SCALE\_FACTOR \times 16 \text{ bytes} \tag{2}$$

where N is the initial point count and each entry in *dataBuffer* requires 16 bytes (x, y, z, rowID). The *SCALE_FACTOR* parameter controls capacity. For example, a value of **2** would take **double** the space to store indexed data (x, y, z, rowID) than that required by *RasterScan*. It is to be noted that static workloads would then need *SCALE_FACTOR=1*. A higher value could result in large unused buffer space, while a smaller value could fill the capacity sooner with updates requiring a full index rebuild. So the choice for *SCALE_FACTOR* is crucial based on available GPU memory for indexing.

4.2 Build Phase

Index construction proceeds in four GPU passes (third step is additional in *CompactScan*):
Pass 1: Histogram (Count Pass) A graphics pipeline processes all N input points in parallel. Each vertex shader invocation computes the bin ID (b_x, b_y) for its assigned point (x, y, z) and performs an atomic increment on `extentBuffer[bin]`.

```
uint binX = (x-minVal.x)*resolution/rangeX;
uint binY = (y-minVal.y)*resolution/rangeY;
uint bin = binY*resolution+binX;
atomicAdd(extent[bin], 1);
```

Listing 1: Count shader core logic

Pass 2: Prefix Sum: A single-pass prefix sum (scan) algorithm computes exclusive prefix sums over the histogram, producing starting offsets for each bin. We do scaling in this stage as per $SCALE\_FACTOR$. A single-pass parallel prefix sum computes exclusive prefix sums over the histogram. We employ the StreamScan algorithm [6] achieving $O(N)$ work complexity.
Pass 3: Buffer Setup The prefix sum results are copied to `startAddrBuffer`. The original capacities and counts are preserved in `capacityBuffer` and `extentBuffer` respectively.
Pass 4: Data Insertion (Build Pass) A second graphics pipeline pass inserts all points into `dataBuffer`.

Each vertex shader computes the bin ID for its point. It then atomically increments a per-bin counter to obtain a local position within the bin. Next it computes the `global position` from `local position` and writes the point data to `dataBuffer`.

```
uint localPos = atomicAdd(count[bin], 1);
uint globalPos = startAddr[bin] + localPos;
dataBuffer[globalPos] = uvec4(x,y,z,rid);
```

Listing 2: Build shader core logic

4.3 Query Phase

Range queries follow a two-pass strategy similar to *RasterScan* [1] that first identifies relevant bins, then collects matching points:

Pass 1: Edge Detection: For a range query we first obtain a rectangle $[x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$, then we compute the range of bins that intersect the query. A compute shader iterates over all bins in the query range and emits $[start, end)$ pairs for non-empty bins to an edge buffer. This is extent-based pruning where we don't put empty bins to edge buffer, in contrast to *RasterScan*, resulting in better query performance.

Pass 2: Result Collection: A graphics pipeline processes the edge buffer as instanced draws. Each instance corresponds to one non-empty bin. The vertex shader iterates over entries in the bin, applies the precise query predicate, and writes matching points to the result buffer.

4.4 Update Phase

`CompactScanIndex` supports in-place updates through separate delete and insert operations, avoiding costly full index rebuilds. Only inserts and/or only deletes are thus trivially supported.

4.4.1 Delete Operation

Deletes are implemented as logical invalidation. Each point is identified by its `rowId`. The delete shader first computes the target bin from the 3D point's (x, y) coordinates. Then it scans entries in the bin from startAddr[bin]. On match, atomically sets the most significant bit (MSB) of the `rowId` field using `atomicOr`.

```
uint start = startAddr[bin];
uint end = start + extent[bin];
for (uint i = start; i < end; i++) {
  if ((dataBuffer[i].w & 0x7FFFFFFF) ==
      targetRowId) { // Set MSB
    atomicOr(dataBuffer[i].w, 0x80000000);
    break;
  }
}
```

Listing 3: Delete shader core logic

4.4.2 Insert Operation

Inserts append new points to free capacity space of bins. First we compute the target bin, then atomically increment `extent[bin]` to reserve a slot. Next we check if `localOffset` $<$ `capacity`[bin]. If within capacity, compute global position and write data. Else, we skip writing data. This is a problem and thus we need to resize the capacity or trigger an index rebuild.

```
uint offset = atomicAdd(extent[bin],1);
if (localOffset < capacity[bin]) {
  uint globalPos = startAddr[bin]+offset;
  dataBuffer[globalPos] = uvec4(x,y,z,rid);
}
```

Listing 4: Insert shader core logic

5 EXPERIMENTAL SETUP

Table 1: GPU System Specifications

Spec	High-End	Low-End
GPU	RTX 4090	GTX 1650
vRAM	24 GB	4 GB
CUDA Cores	16,384	896
RT Cores	128	0
Arch.	Ada (SM 8.9)	Turing (SM 7.5)
Bandwidth	1,008 GB/s	128 GB/s

Table 2: Software Environment

Component	Version
OS	Ubuntu 22.04 LTS (Kernel 6.8+)
Graphics API	Vulkan 1.3+
Compiler(Build System)	GCC 11.4+ (CMake 3.22+)

5.1 Evaluation Scope

Find the source code for experiments here. For this proof of concept, we constrain our evaluation as below:

Three Integer Columns: Datasets consist of three 32-bit unsigned integer (`uint32_t`) columns.

Batch Operations: Insertions and deletions are evaluated in batches as single update causes GPU launch overheads. However, it supports single updates, but we recommend batching.

Synthetic and Benchmark Data: Performance is evaluated across uniformly distributed points, normally distributed points, highly skewed Zipfian distributions, and the TPC-C style customer table following the TPC-C specification with `W_ID`, `D_ID`, and `C_ID` columns.

Queries: We generated and tested on **10** queries ($Q1$, $Q2$,...$Q10$) for each dataset with selectivity (10%, 20%, ...100%) respectively. Thus $Q10$ returns all rows.

6 RESULTS

6.1 CompactScan Index Build Times

We evaluated the index build performance of both *RasterScan* and *CompactScan* across multiple index resolutions (1024^2, 2048^2, 4096^2, 8192^2, and 16384^2) under different data distributions at $SF = 1$. Higher resolution would imply lower per bin occupancy.

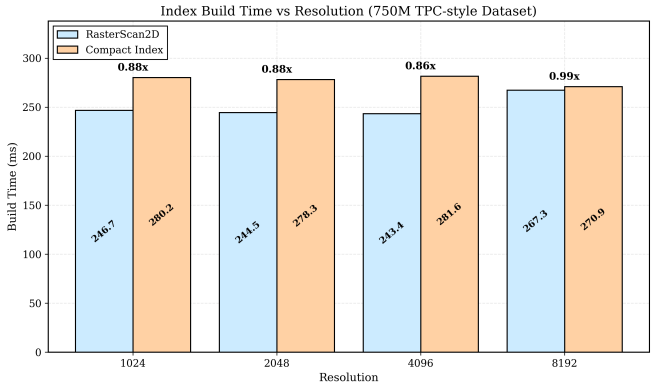

Figure 1: Index Build Times (TPC-C 750M)

For the TPC-C dataset, build times remain relatively stable from 1024^2 to 8192^2 resolutions, with both indexes completing in under 300 ms for 750 M points. CompactScanIndex incurs an overhead of approximately 10–15% compared to RasterScan2D at lower resolutions; however, this gap narrows at 8192^2, where both indexes achieve near parity. At 16384^2 resolution, build times increase sharply by approximately $10\times$, primarily due to memory bandwidth limitations and the overhead associated with managing a significantly larger number of bins causing full memory usage.

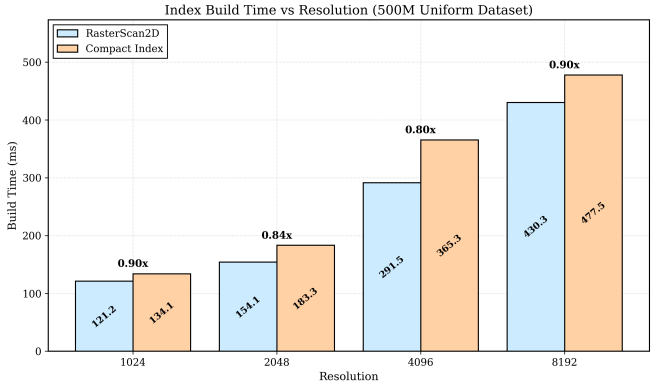

Figure 2: Index Build Times (Uniform 500M)

CompactScan incurs an 11–25% index build overhead for the 500M Uniform dataset.

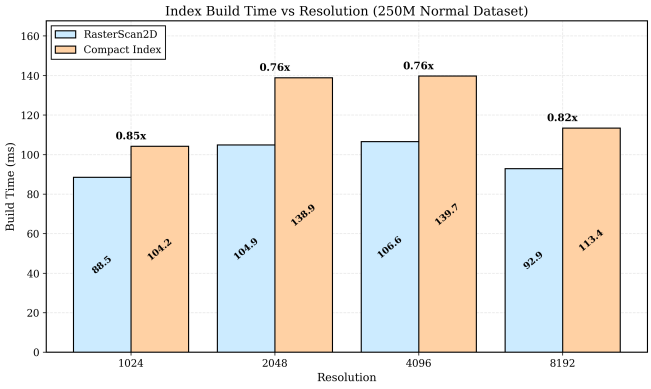

Figure 3: Index Build Times (Normal 250M)

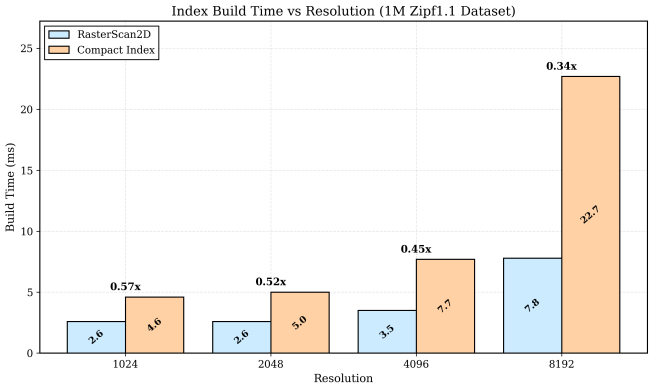

Figure 4: Index Build Times (Zipf1.1 1M)

CompactScanIndex builds are 25-53% slower than RasterScan2D across all distributions. Highly skewed Zipf data reveals the most significant performance gap between the two indexes. The CompactScanIndex overhead increases substantially with resolution (1.77× to 2.91×). This overhead comes from the additional buffer operations required to support updates (*capacityBuffer*, *extentBuffer*, *startAddrBuffer*)

6.2 Impact of SCALE_FACTOR parameter on index build times

Next, we compared build times and average query times for 100M TPC-C and Uniform dataset at (8192^2) resolution. TPC-C builds are consistently 20–40% faster (5) than uniform data (6) at equivalent *dataBufferSize* sizes due to skewed value distributions that concentrate points in fewer bins, reducing atomic contention. In contrast, uniform data spreads points evenly across (8192^2) bins, increasing memory dispersion and cache misses during count and insert passes.

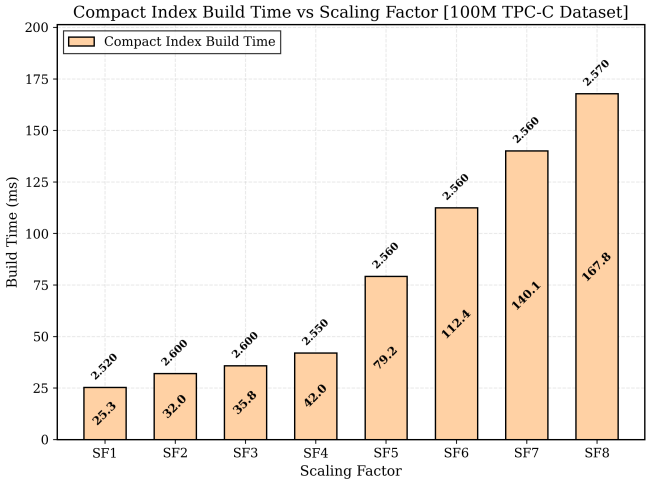

Figure 5: SF Impact on Index Build (TPC-C 100M)

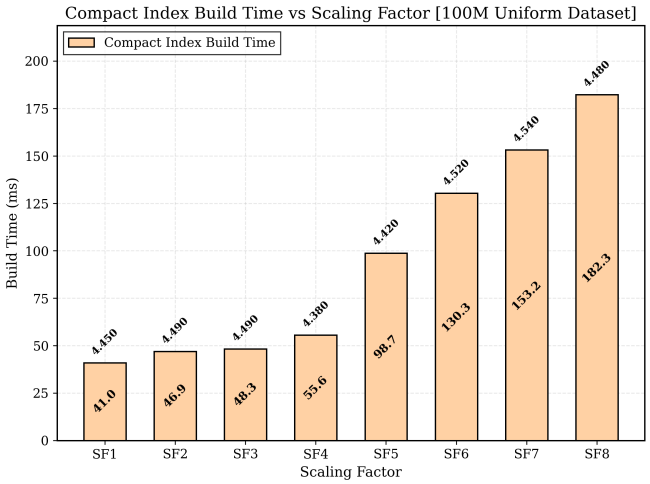

Figure 6: SF Impact on Index Build(Uniform 100M)

TPC-C query times (given on top of bars) are 1.75x faster than uniform (2.55ms vs 4.48ms average). Faster queries stem from its clustered data—range queries touch fewer bins with higher point density, while uniform data spreads results across more bins requiring broader scans.

Takeaway: Although we might expect Uniform to perform better over TPC-C (lightly skewed) with uniform bin sizes and more bins, GPU can perform better with few hot bins and mostly empty bins as a result of extent-pruning.

6.3 Static Query Times

We wanted to show that even at the cost of slightly higher build times, *CompactScan* can outperform *RasterScan* in query performance. For normally distributed dataset, we saw 1.44x average speedup. When both indexes have similar occupancy per bin, *CompactScan* incurs slight overhead of 6% overhead. For TPC-C (real-world skewed workload), *CompactScan* is 1.61x faster. These results show that even in static workload scenarios, we can use *CompactScan* index.

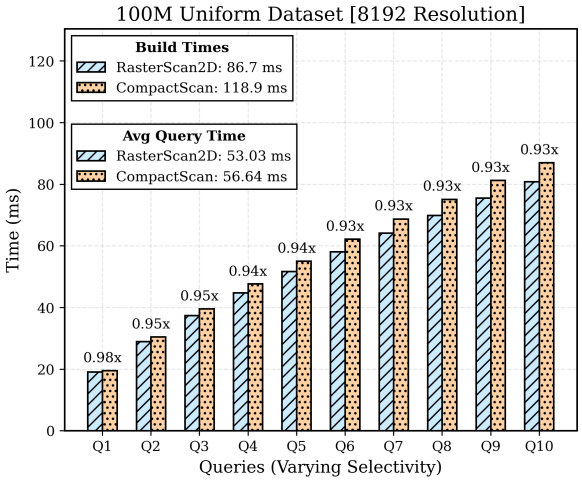

Figure 7: Static Query Times (Uniform)

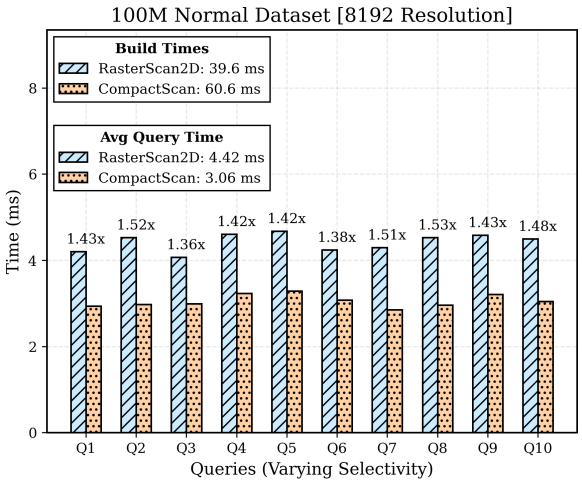

Figure 8: Static Query Times (Normal 100M)

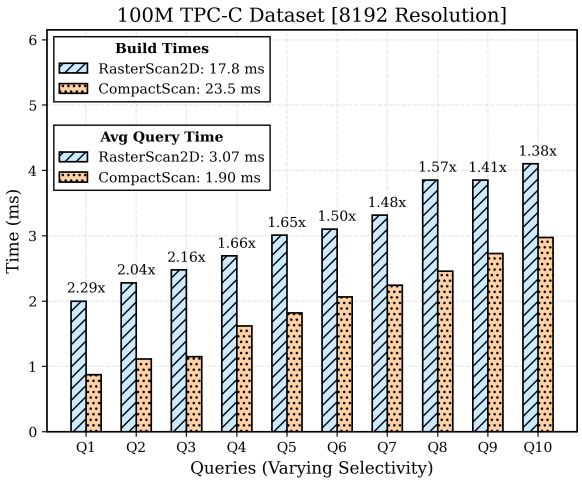

Figure 9: Static Query Times (TPC 100M)

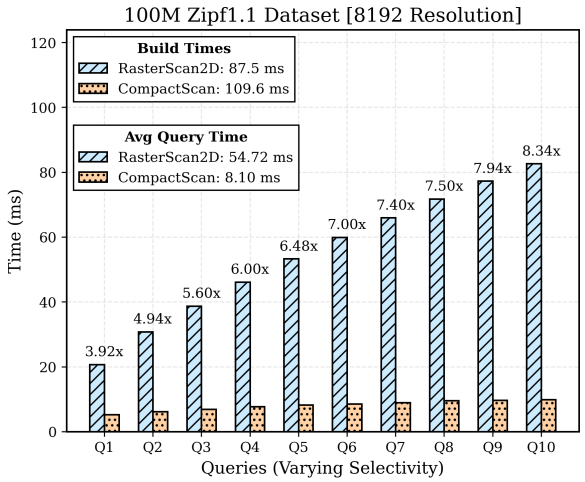

Figure 10: Static Query Times (Zipf 1M)

Figure 10 shows *CompactScan* dramatically outperforms *RasterScan* for skewed distributions. On Zipf1.1 dataset, queries are 6.75x faster on average because our extent-based pruning skips empty bins, while *RasterScan* scans all [st,en) ranges in its *range2D.frag* shader during query processing pipeline.

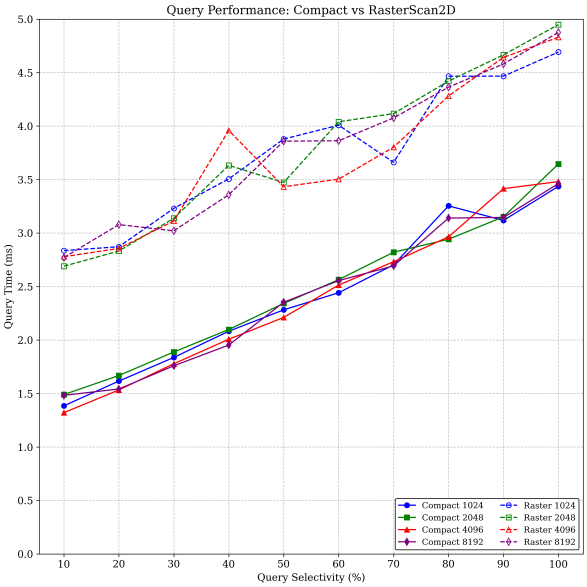

Figure 11: TPC-C Evaluation for Query Times(100M)

Figure 11 shows how *CompactScan* performs wrt to *RasterScan* at various *INDEX RESOLUTION* and at varying query selectivities. We see upto 2x speedups. We see an almost linear rise in query time with query selectivity with little red spikes and dips, possibly due to unsuitable 4096 index resolution, as others remain relatively stable.

6.4 Index Update Performance

We show how index updates (deletes followed by inserts) compare against initial full index build time at varying batch sizes. These batches are randomly formed from the given dataset. We delete and insert the same batch in our experiments. For correctness check, we verified the counts and resulting points against CPU verification code.

Table 3: Index Update Speedup Against Index Build

Batch Size	TPC-C	Normal	Uniform
5	51×	187×	118×
50	42×	154×	101×
500	36×	42×	83×
5,000	31×	16×	90×
50,000	13×	14×	26×

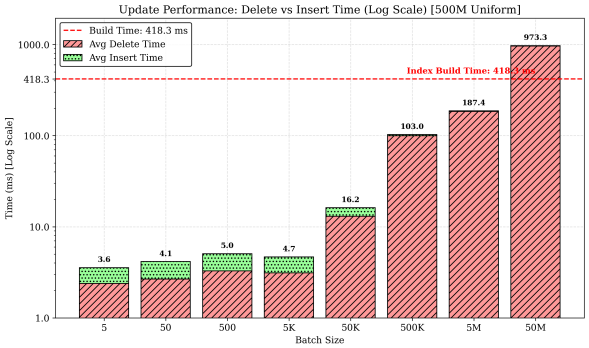

Figure 12: Update Performance (Uniform 500M)

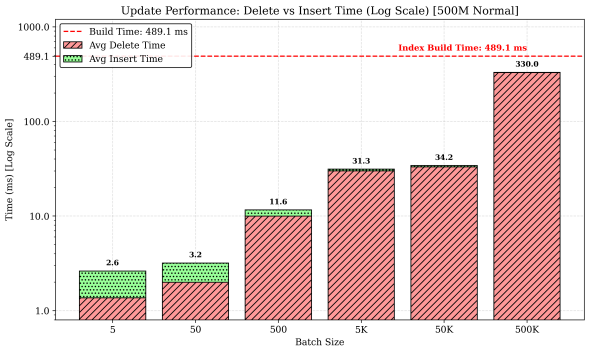

Figure 13: Update Performance (Normal 500M)

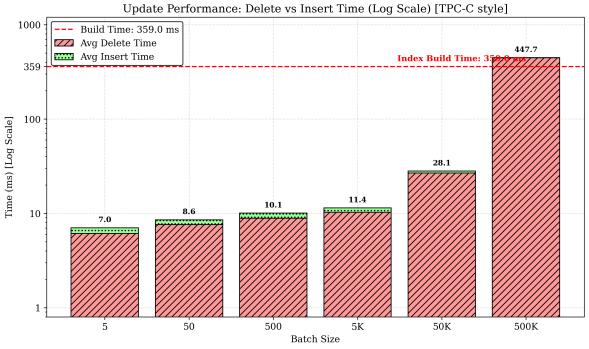

Figure 14: Update Performance (TPC-C 500M)

6.4.1 Observations

Insert costs are nearly constant(1-3ms) regardless of batch size, demonstrating O(1) amortized inserts. The GPU processes inserts in parallel with minimal overhead. Delete cost scales sub-linearly with batch size.

6.5 Query Performance After Updates

We measure query times (Q1–Q10) before and after updating the index with a batch of data representing a fraction of the total points. In most cases, updates increase query latency, though occasional reductions are observed due to GPU warm-up effects. Each update cycle deletes a random batch of m points and reinserts them to preserve an identical dataset across runs, enabling CPU-side correctness verification. As updates are modeled as delete–insert operations, and since inserts are inexpensive while deletes dominate the cost, overall update performance is primarily governed by deletion overhead.

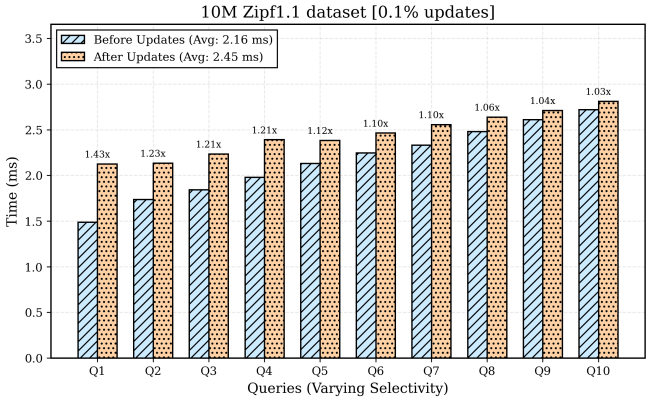

Figure 15: Query Time Post-Update(10M Zipf1.1)

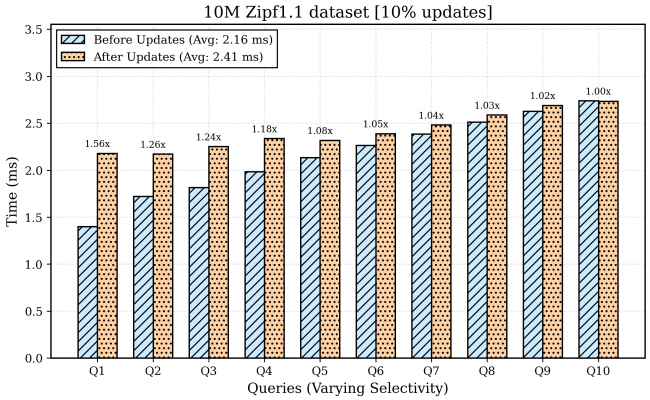

Figure 16: Query Time Post-Update(Zipf 10M)

Zipf distribution shows higher update sensitivity for small queries. Low-selectivity queries (Q1-Q3) experience more degradation after updates, while high-selectivity queries (Q8-Q10) remain nearly unchanged (less than 5% impact). This pattern occurs because Zipf's highly skewed distribution concentrates most points—and therefore most updates—in a few "hot" bins. These hot bins accumulate deleted (invalidated) entries that the query shader must skip during query scans.

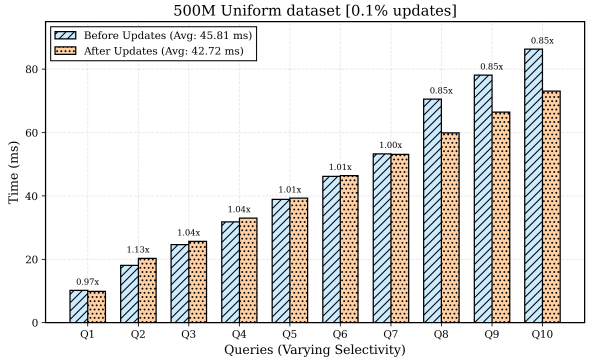

Figure 17: Query Time Post-Update(Uniform 500M)

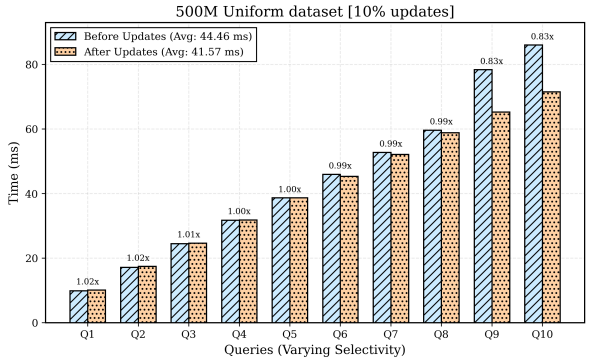

Figure 18: Query Time Post-Update (Uniform 500M)

For uniform, the query times actually decreased by up to 17% after updates, particularly for high-selectivity queries (Q9-Q10). This counterintuitive result is likely due to GPU warm-up effects—the update operations (delete + insert shaders) warm the GPU caches and memory subsystem, benefiting subsequent queries.

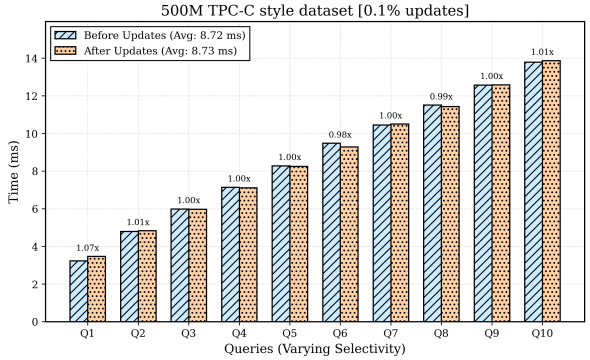

Figure 19: Query Time Post-Update(TPC-C 500M)

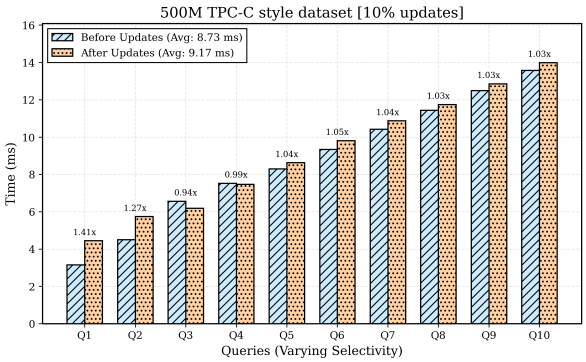

Figure 20: Query Time Post-Update(TPC-C 500M)

In TPC-C, updates cause a modest 3-5% query time increase for most queries, with larger degradation (27-41%) on smaller selectivity queries (Q1-Q2). The hot bins accumulate more invalidated entries, slightly increasing scan overhead. Also GPU warmup may be benefitting later queries.

6.5.1 Conclusion

CompactScan maintains stable query performance even after 10% of data is updated. This is not the upper limit. The observed variations (±5%) in average query times are within acceptable bounds for real-time indexing.

7 CONCLUSION

We demonstrate the feasibility and effectiveness of our approach. Experiments performed at varying configurations show that rasterization-based indexing can be successfully extended to dynamic workloads, with little overheads of index builds, while preserving the query performance advantages of the original static design, making it suitable for update-intensive analytical database scenarios. It even shines when we have append-only data as inserts are much faster over deletes. We also showed that even in static-workloads, we can substitute *CompactScan* for *RasterScan* by setting $SCALE\_FACTOR = 1$ for faster static querying at the cost of slight index build overheads.

8 FUTURE WORK

Our current implementation is a proof of concept for dynamic updates in rasterization-based GPU indexing. It currently follows equi-width histograms with fixed bin boundaries, variable bin sizes. Several extensions can broaden applicability and improve performance:

8.1 Extended Data Type Support:

Extending beyond 32-bit unsigned integers to support 64-bit integers, floating-point types, strings, and heterogeneous column types (e.g., int32, int64, float64).

8.2 Adaptive Binning Strategies:

Replacing uniform fixed-resolution binning with these adaptive schemes to improve load balance and reduce worst-case bin sizes:

Equi-Depth Histogram: Partition data into bins so each bin contains approximately N/B points, ensuring homogenous and bounded bin sizes.

Multi-Resolution Binning: Employ hierarchical bins with finer granularity in dense regions and coarser bins in sparse regions.

References

[1] Harish Doraiswamy and Jayant R. Haritsa. Raster Is Faster: Rethinking Ray Tracing in Database Indexing. In *CIDR*, 2026. https://vldb.org/cidrdb/papers/2026/p18-doraiswamy.pdf

[2] Justus Henneberg and Felix Schuhknecht. RTIndeX: Exploiting Hardware-Accelerated GPU Ray Tracing for Database Indexing. In *PVLDB*, 2023. https://doi.org/10.14778/3625054.3625063

[3] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X. Sean Wang. RTScan: Efficient Scan with RT Cores. In *PVLDB*, 2024. https://dl.acm.org/doi/10.14778/3648160.3648183

[4] Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, and Rubao Lee. RTCUDB: Building Databases with RT Processors. *arXiv preprint arXiv:2412.09337*, 2024. https://arxiv.org/abs/2412.09337

[5] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *GPGPU-3*, 2010. https://dl.acm.org/doi/10.1145/1735688.1735706

[6] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. *SIGPLAN Notices*, 48(8):229–238, Aug 2013. https://code.google.com/archive/p/streamscan

[7] Microsoft Research. RasterScan: GPU Rasterization-Based Indexing. Source code repository, 2025. https://github.com/Microsoft/raster-scan

[8] AntaresAlice. RTScan: Source Code Repository. Source code repository, 2024. https://github.com/AntaresAlice/RTScan