# Raster is Faster: Rethinking Ray Tracing in Database Indexing

Harish Doraiswamy
Microsoft Research
Bengaluru, India
harish.doraiswamy@microsoft.com

Jayant R. Haritsa
Indian Institute of Science
Bengaluru, India
haritsa@iisc.ac.in

## ABSTRACT

Advances in GPU technology have frequently driven innovations in database query processing. The recent incorporation of native support for ray tracing (RT) in modern GPUs has, as expected, spurred interest in its application to relational database workloads – in particular, RT has been advocated for column indexing. We examine this premise here and conversely argue that it is the classical rasterization pipeline, rather than RT, which is to be preferred for such applications. The rationale is that rasterization uses arithmetic value comparisons, instead of the computationally heavy ray-triangle geometric intersections of RT, for data search. To substantiate this claim, we design RasterScan, a purely rasterization-based indexing approach, that adheres to the data model employed by RT-based methods. Our evaluation over a variety of large datasets demonstrates that RasterScan consistently and substantively outperforms its RT counterparts, often achieving order-of-magnitude speedups with regard to both index build and search times.

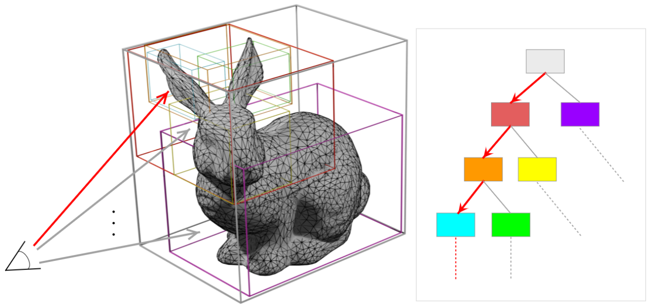

(a) 3D Scene (b) BVH Traversal

**Figure 1: Ray tracing using BVH tree traversal. The nodes of the BVH tree are traversed in the order of the red arrows to identify the set of triangles that intersect the red ray incident on the model.**

## 1 INTRODUCTION

The performance of Graphics Processing Units (GPU) has dramatically improved in recent times due to architectural enhancements. In particular, GPUs now include custom/dedicated processors to handle specific tasks such as *ray tracing* (RT), which physically tracks the rays of light incident on a camera. As compared to the traditional rasterization approach, RT supports creating scenes with highly accurate lighting and shadow effects.

### Ray Tracing in Computer Graphics

Given a ray, RT is accomplished by traversing a Bounding Volume Hierarchy (BVH). This is a tree-based data structure [4], where each node contains a bounding volume enclosing its children – in essence, it is a generalized version of the familiar R-trees [2] used in spatial databases. The BVH is created over the set of *triangles* (or alternative *parametric primitives* such as spheres, cubes, etc.) that make up the graphics scene, and helps identify the indexed objects that intersect with the incident ray.

For example, consider the 3D scene shown in Figure 1(a), representing the triangulated model of a rabbit. The corresponding BVH tree is shown in Figure 1(b). The colored boxes in the model denote the associated sub-volumes encompassed by the (similarly colored) nodes of the BVH tree. Rendering the scene involves shooting multiple rays from the eye (camera) to identify the set of triangles that intersect these rays.

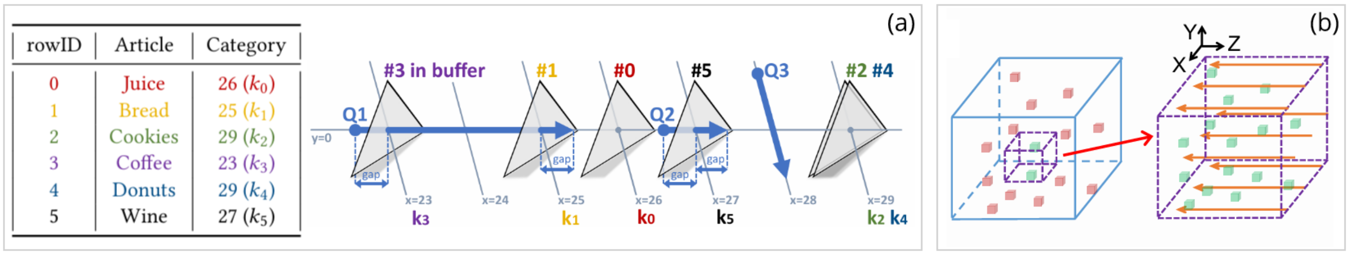
Take, for instance, the red ray incident on the rabbit in Figure 1(a). The path followed by the BVH traversal for this ray is shown using red arrows in Figure 1(b). Once this traversal reaches the leaf node(s) of the BVH tree, a ray-triangle intersection test is performed over the triangles within the corresponding bounding box(es) to identify the set of intersected triangles. Modern GPUs have "RT cores" that natively support this tree traversal.

Although RT cores were designed for graphics computing, during the past few years, researchers have tried to leverage them for database processing as well (e.g., [1, 3, 7, 10]). The use-cases range from query execution to column indexing, and in this work, we focus on the indexing application as an illustrative example to gauge the suitability of RT for relational database workloads.

### Database Indexing using RT

**RTIndex.** The first work to use RT to accelerate indexing of integer columns was RTIndex. It models the values in a column as triangles along the x-axis as shown in Figure 2(a) (row IDs are denoted using '#'). A triangle is placed around each data point, such that the point is symmetrically contained in the triangle. This is accomplished by generating three vertices that offset, by half the grid interval, the point along the different axes.

Given the above data model, column search predicates in the user query are converted into one or more rays (the start and end points of the rays depend on the query constraints). Then, the native RT support provided by the GPU is used to identify all triangles that intersect with the set of query rays – the intersected triangles correspond to the search results. Figure 2(a) shows an example of a range query (Q1) and two point queries (Q2 and Q3). Note that if the data has duplicate values, the above approach can create multiple



| rowID | Article | Category |
|---|---|---|
| 0 | Juice | 26 ($k_0$) |
| 1 | Bread | 25 ($k_1$) |
| 2 | Cookies | 29 ($k_2$) |
| 3 | Coffee | 23 ($k_3$) |
| 4 | Donuts | 29 ($k_4$) |
| 5 | Wine | 27 ($k_5$) |

**Figure 2: RT-based indexing. (a) RTIndex [3] is a single column index that places triangles corresponding to the data points along the $x$-axis such that its center lies on the axis. The Row IDs are denoted using '#'. Q1 corresponds to a range query, while Q2 and Q3 illustrate two ways of realizing point queries (figure sourced from [3]). (b) A 3-column index built using RTScan [7]. Cubes corresponding to each row are placed on a 3D grid, where the coordinates correspond to the values of the 3 columns, respectively. A conjunctive range query is mapped as a cube (inner purple cube), and rays are shot within this cube to compute the query results (figure sourced from [7]).**

coincident triangles, which can result in increasing the intersection workload of a single ray, adversely affecting parallelism.

**RTScan.** A followup RT-based indexing proposal, RTScan [7], improved upon RTIndex through:

(1) An alternative transformation of the query constraint into rays that improves parallelism – instead of shooting one long ray, RTScan splits it into multiple shorter perpendicular rays;
(2) An encoding scheme that uniformly maps $N$ data points uniquely to values in the range $[0, N-1]$. This ensures that duplicates are mapped to different values after the encoding. Taken together with (1) above, it further improves parallelism by reducing the load on a single ray;
(3) Replacing triangles with smaller cubes, thus reducing the number of rays to be shot;
(4) Improving query performance through a Bindex [5]-like data filter; and
(5) Indexing up to three columns using a single BVH structure.

Figure 2(b) illustrates a 3 column index built using RTScan. Cubes are placed on a 3D grid, where the coordinates of the center of a cube correspond to the values of the 3 columns in each relational row, respectively. A conjunctive range query is mapped as a cube (inner purple dashed cube in Figure 2(b)), and rays are shot within this cube to compute the query results.

## Is RT an Overkill?

RT is certainly a generic and powerful graphics technique. However, it is our contention that it represents an *overkill* for database indexing, incurring performance degradation. This is because, in the database context, the triangles (or cubes) hosting the data points are not located randomly, but neatly organized on the grid. Further, and more importantly, the rays representing the search predicates are also aligned with the grid axis, due to the linearity of the column search predicates. But the BVH tree and the ray-triangle intersections do not take advantage of this inherently "well-behaved" nature of database processing. Instead, they incur serious overheads due to: (a) BVH trees being heavy to process on GPUs because of load balancing issues, irregular memory access patterns, and memory divergence during graph traversal [6]; and (b) geometric intersections requiring considerable computational processing.

## Contributions

Given the above, an obvious question is whether a comparatively light-weight technique can be designed for database indexing on GPUs. Our proposal is that, while using a similar data model to RT, indexing can be achieved efficiently via the mature *rasterization pipeline* which is at the core of graphics processing. To evaluate this hypothesis, we present **RasterScan**, a raster-based indexing strategy designed as follows:

(1) Given that the data is already represented as points on a grid, RasterScan simply models it as an **image**.
(2) Search rays are replaced by **lines** that are efficiently drawn through rasterization. A crucial benefit of this modification is that the geometric intersection test is replaced by a fast and simple arithmetic comparison.

We evaluate the performance of RasterScan on a variety of datasets and GPU platforms, including those used in [7] for RTScan. Our results show that RasterScan consistently attains substantive speedups over RTScan, with regard to both index build and index search – the improvements often exceed an order-of-magnitude.

## 2 BACKGROUND: RASTERIZATION PIPELINE

We present here a brief overview of the rasterization pipeline, which is ubiquitously used in interactive graphics applications, including games. A simplified view of this pipeline is shown in Figure 3. It is composed of a series of "shader programs" that are executed in sequence to generate the required "drawing" (or *rendering*). These pipeline stages are summarized below – see [11] for more details.

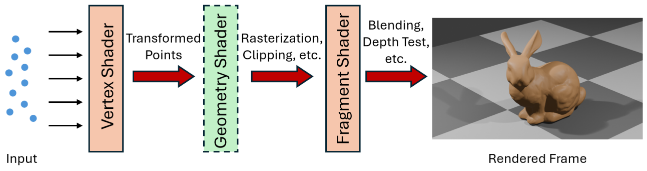
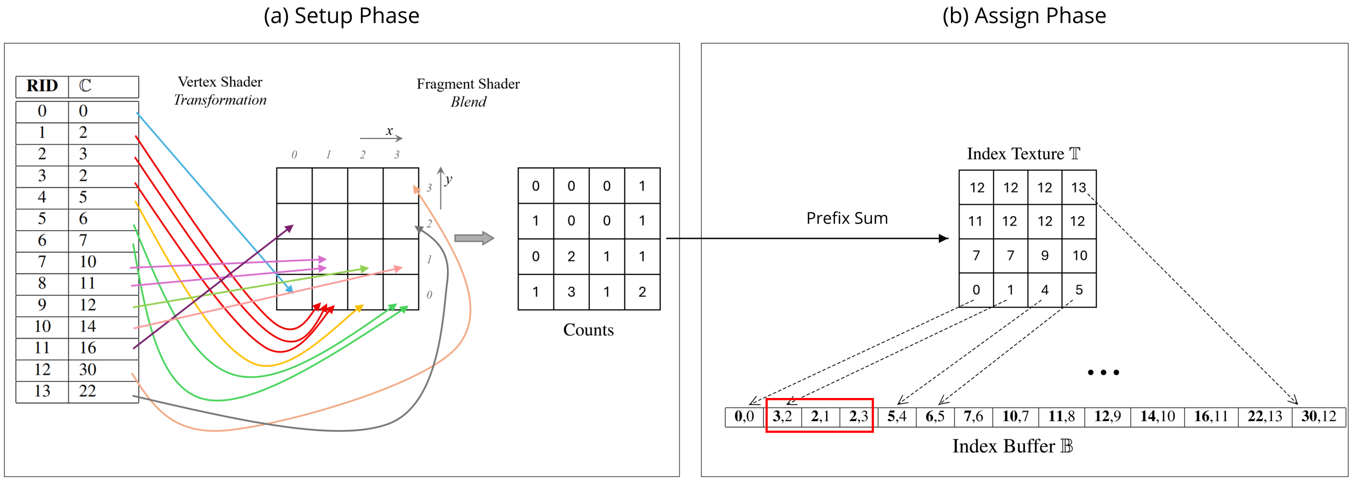


**Figure 3: Rasterization Pipeline.**

**Vertex Shader.** The vertex shader is typically used to perform *model-view-projection*, which transforms the vertices corresponding



| RID | ℂ |
|---|---|
| 0 | 0 |
| 1 | 2 |
| 2 | 3 |
| 3 | 2 |
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 10 |
| 8 | 11 |
| 9 | 12 |
| 10 | 14 |
| 11 | 16 |
| 12 | 30 |
| 13 | 22 |

|  |  |  |  |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 2 | 1 | 1 |
| 1 | 3 | 1 | 2 |

|  |  |  |  |
|---|---|---|---|
| 12 | 12 | 12 | 13 |
| 11 | 12 | 12 | 12 |
| 7 | 7 | 9 | 10 |
| 0 | 1 | 4 | 5 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0**,0 | **3**,2 | **2**,1 | **2**,3 | **5**,4 | **6**,5 | **7**,6 | **10**,7 | **11**,8 | **12**,9 | **14**,10 | **16**,11 | **22**,13 | **30**,12 |

**Figure 4: Index build on a toy column with 14 data values.**

to the objects in a scene into a common *screen space* (the coordinate system wrt the camera). A separate vertex shader instance is instantiated for each point of the input. These instances are executed in a *Single Program Multiple Data* (SPMD) fashion over the vertices. The graphics driver appropriately schedules the different vertex shader instances, and also prefetches the vertex data from the *vertex buffer* into local memory to improve the performance of this stage.

**Geometry Shader.** This shader is used to create custom geometries. Depending on the type of primitives (points, lines, or triangles) being rendered, each geometry shader instance receives one, two, or three of the transformed points processed by the vertex shader. The developer can use these transformed points to generate additional geometries. For example, to save buffer space, one could pass only the diagonal end points of rectangles to be rendered to the vertex shader as a set of lines. The geometry shader can then be used to take these end points, and generate two triangles that form the required (axis-parallel) rectangle.

**Vertex Post Processing.** In the next step, *clipping* and *rasterization* are performed by the GPU's native driver on the transformed primitives output by the geometry shader.

Clipping is the process whereby primitives outside the *viewport* (the region visible from the camera) are removed. Primitives such as lines and triangles that partially intersect the viewport are also cropped during this operation, resulting in a new set of primitives that are fully contained within the viewport. The viewport together with its resolution is set by the developer when initializing the rasterizaton pipeline.

Rasterization is applied on the clipped primitives to generate a set of pixel locations called *fragments.*

**Fragment Shader.** The fragments generated in the above step are processed independently in parallel by multiple fragment shader instances, one for each fragment. In a traditional graphics setting, a fragment shader is used to perform lighting calculations and set the "color" of the processed fragment. However, it can also be used for other purposes such as writing to output buffers.

**Post Fragment Processing.** The colored fragments output from the fragment shader are processed to generate the pixels of the rendered image. Note that multiple fragments can correspond to a given pixel. Such fragments are combined in parallel through an operation called *blending* to finally generate a single pixel.

**Virtual Screen.** Graphics APIs also support rendering images to a "virtual" screen. The rendered image in this case is stored as a *texture*, a specialized buffer for storing images.

## 3 RASTERSCAN: INDEX BUILD

Simply put, rasterization converts a geometric primitive (point, line, or triangle) into a set of pixels in 2D space. RasterScan utilizes this process to both index the data (this section), as well as execute queries over this data (next section) through drawing operations. Since our goal is to demonstrate that rasterization is a significantly better alternative to RT, we focus only on numerical data columns so that we can compare against the RT-based indexing techniques – the handling of non-integer data types is discussed in Section 7. Further, for ease of exposition, we first explain the RasterScan indexing and querying strategy for a single column index. Extensions to index multiple columns are described later in Section 5.

Similar to the RT alternatives, a column of a relational table is modeled as a set of 1D points. The key idea underlying RasterScan is to model the index as a *texture* and assign data points to a pixel location within this texture. In other words, the data points are logically "binned" into pixels of a texture.

Physically, the index is stored as a pair $(\mathbb{T}, \mathbb{B})$, where $\mathbb{T}$ is the texture in which each pixel stores a pointer to its corresponding bin, and $\mathbb{B}$ is an index buffer that stores the actual bins. Each entry in $\mathbb{B}$ stores the data value being indexed along with its row identifier *rid* in the input table. The indexing itself is accomplished in two phases –a *setup phase* and an *assign phase*, as illustrated in Figure 4. Note that the rasterization pipeline designed for these phases only draws a set of *point* primitives and therefore does not utilize the geometry shader in the pipeline.

### 3.1 Setup Phase

The first phase creates a histogram of the bins. That is, it counts the number of input points falling into each pixel of the index texture. This aggregation is accomplished using the rasterization pipeline, which processes the data points independently in parallel. In particular, the vertex and fragment shaders are leveraged for this purpose, as described next.

**Vertex Shader.** The vertex shader is used to transform the data points into the corresponding pixel locations. Specifically, given an index texture $\mathbb{T}$ with resolution $R \times R$, the vertex shader takes a data point $d$ as input, and outputs the coordinates $(x, y) \subseteq [0, R) \times [0, R)$ as its texture location.

The binning strategy is based on the following transformation logic: Let $\mathbb{C} = \{u_1, u_2, \ldots, u_n\}$ be the set of points to be indexed. Let $u_{min} = \min_i(\mathbb{C})$ and $u_{max} = \max_i(\mathbb{C})$. The range of the data, $[u_{min}, u_{max}]$, is then equally divided among the $R^2$ pixels as follows, in row-major order. Consider a pixel $p_i(x, y) \in \mathbb{T}$, where $i = x + y \cdot R$, $x, y \in [0, R)$. The bin corresponding to this pixel stores all data points in the range $[i \cdot du, (i+1) \cdot du)$, where $du = \lceil \frac{u_{max} - u_{min}}{R^2} \rceil$.

In essence, the vertex shader maps points in 1D to 2D using the following two equations. Given a point with value $u$, its bin is computed as

$$i = \lfloor \frac{u}{du} \rfloor \tag{1}$$

This bin $i$ is then transformed into the 2D location

$$(x, y) = (i \bmod r, \lfloor \frac{i}{R} \rfloor) \tag{2}$$

A toy example of the above process is shown in Figure 4(a), where an index is built over an input data column with 14 values, and the count texture is generated after this step. In this example, the resolution of the index texture is $R = 4$, and the range of the column is $u_{max} - u_{min} = 30$. The range of each bin is then computed as $du = \lceil \frac{30}{16} \rceil$. Thus the pixels in row-major order contain points with values in the range $[0, 1], [2, 3], \ldots, [30, 31]$, respectively.

**Fragment Shader.** A separate fragment shader instance is spawned for each transformed point, and used to increment the count of the corresponding pixel. The blend operation of the rasterization pipeline is used to accomplish this increment operation efficiently.

### 3.2 Bin Assignment Phase

This step, shown in Figure 4(b), assigns the input data values to the corresponding bin locations in the index buffer $\mathbb{B}$. First, the *exclusive prefix-sums* of the counts found in the setup phase are computed, and these sums form the index texture $\mathbb{T}$. That is, the value of each pixel in this texture denotes the start location of the corresponding bin in $\mathbb{B}$. The data points are stored in these bins using the following rasterization pipeline:

**Vertex Shader.** The vertex shader operates similarly to the setup phase – it transforms the data points to their corresponding pixels.

**Fragment Shader.** The fragment shader stores the points in the corresponding bins. Prior to running this pipeline, a copy, $\mathbb{T}'$, of the index texture $\mathbb{T}$ is made – this copy is used to keep dynamic track of the next position at which to insert a point, and is discarded once the bin assignment is done.

Each fragment shader instance obtains the location of the corresponding point in the index buffer $\mathbb{B}$ from the texture $\mathbb{T}'$. We need to ensure correctness and consistency when multiple fragment shaders act on the same pixel. For this purpose, we use the `atomicAdd()` function which returns the original value of a location after atomically incrementing its value.

Returning to Figure 4(b), each entry in $\mathbb{B}$ is denoted as a pair of numbers–the first number (in bold) corresponds to the indexed data value, and the second number (in regular text) corresponds to the *rid*. For instance, the third entry (**2**,1) indicates that the value 2 appears in Row 1 of the input table for the indexed column. The bin corresponding to the pixel $(1, 0)$ in $\mathbb{T}$ is highlighted in red.

## 4 RASTERSCAN: INDEX SEARCH

We move on, in this section, to discuss how a search can be performed on the RasterScan index using the rasterization pipeline. Again, we describe this process for the 1D case, and defer higher dimensions to Section 5.

Specifically, we consider range queries of the form $q = [l, r]$, i.e., we are interested in all points $\{u_i \in \mathbb{C} | l \leq u_i \leq r\}$. Equality queries are processed by setting $l = r$, while < and > are modeled by setting $l$ or $r$ to the minimum or maximum of the domain, respectively.

### 4.1 Index Texture Search

The first step in the search process is to identify the bins of interest from the index texture $\mathbb{T}$. This is accomplished using the following rasterization pipeline:

**Vertex Shader.** The vertex shader takes as input a pair of 1D values, $l$ and $r$. It then invokes Equations 1 and 2 (similar to the vertex shader used for index creation) to transform this pair into pixel locations $(x_1, y_1)$ and $(x_2, y_2)$, respectively. These locations are then passed on to the geometry shader for further processing.

For example, consider the query in Figure 5(a) on the index shown in Figure 4. Here, $l = 3$ and $r = 19$, and the corresponding transformed points are $(1, 0)$ and $(1, 2)$, respectively.

**Geometry Shader.** The geometry shader takes the pixel locations as input and creates a geometry in the texture space that covers all bins (pixels) falling into the query range. In Figure 5(a), the pixel locations corresponding to this range are highlighted in blue.

The geometry creation is accomplished by drawing a single rectangle that corresponds to the minimal set of complete rows in $\mathbb{T}$ that cover the required region – highlighted as a red rectangle in Figure 5(a). Note that this means that there may be redundant pixels (white cells) within the red rectangle – they are removed later by the fragment shader. The diagonal of this (axis-parallel) rectangle is defined by the point pair $(0, y_1); (R, y_2 + 1)$. In Figure 5(a), it corresponds to $(0, 0); (4, 3)$.

Given the graphics environment, the red rectangle is modeled as a *pair of triangles*, and the output of the geometry shader is the set of vertices of these two triangles. The rasterization pipeline takes this triangle pair and rasterizes them into the pixels that fill the corresponding logical rectangle. A separate fragment shader is spawned for each of the filled pixels.

**Fragment Shader.** The fragment shader discards two types of pixels: (1) Pixels falling outside the query range (e.g. the three white pixels within the red rectangle at locations $(0, 0)$, $(2, 2)$ and $(3, 2)$ in Figure 5(a)); and (2) Pixels corresponding to empty bins

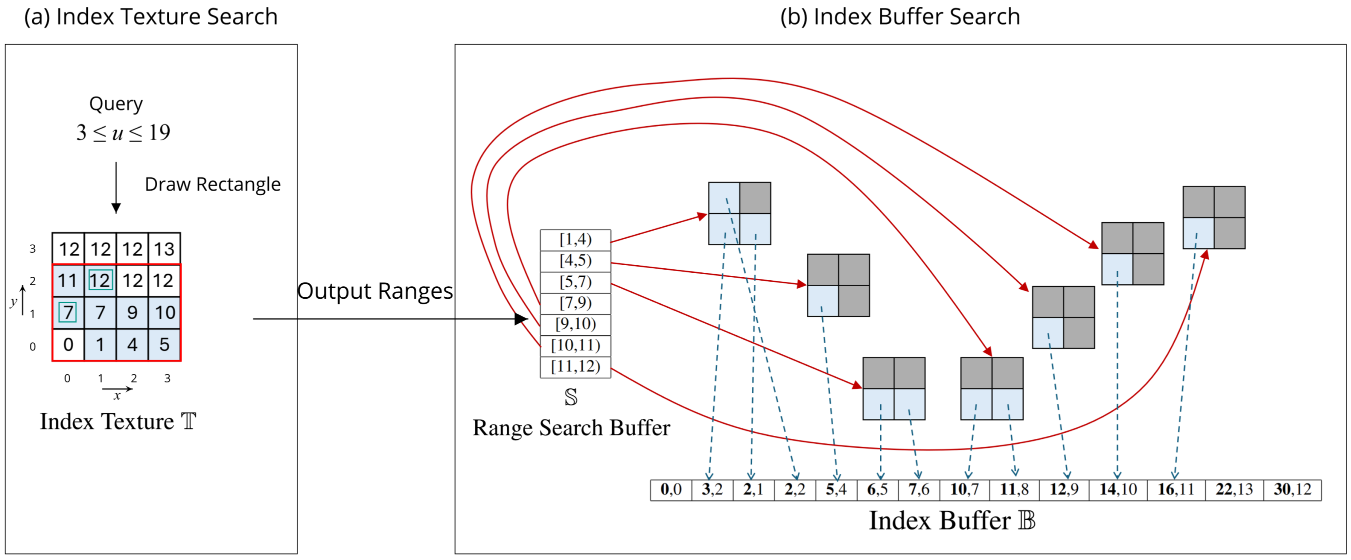


**Figure 5: Range query execution. A rectangle (red) is drawn to identify pixels (bins) of interest in $\mathbb{T}$. The cells colored blue correspond to those having points within the query range. The set of index ranges to search in the index buffer are output to $\mathbb{S}$. The buffer search is then accomplished by drawing a set of rectangles, each corresponding to one of the index ranges in $\mathbb{S}$.**

(e.g. the cells at $(0, 1)$ and $(1, 2)$ in Figure 5(a) highlighted by green internal rectangles).

For a pixel that falls within the query range, we obtain the start and end pointers of the corresponding bin using a texture lookup from $\mathbb{T}$. Let $[st, en)$ denote this range. The simplest way to proceed further would be to test, within the fragment shader itself, whether points in $\mathbb{B}[i], \forall i \in [st, en)$, satisfy the query constraints. Such an approach, however, has a crucial shortcoming: all these points will be processed *linearly* within a single fragment shader thread, and therefore the parallelism provided by the GPU will be lost. Further, in case the distribution of the points is skewed, such a thread will bottleneck the other threads in the warp (typically 32 threads in case of Nvidia GPUs) that are executed together.

Therefore, instead of processing the entire bin in a single fragment shader, we store the range $[st, en)$ as a 2D point in a *Range Search* buffer $\mathbb{S}$. This buffer is processed in a second pass, as described next. In Figure 5(b), there are 7 ranges stored in $\mathbb{S}$ corresponding to the populated pixels falling within the query range.

## 4.2 Index Buffer Search

To improve parallelism, we use a second pipeline to search the different bins of $\mathbb{B}$. Let $d = \max_{(st,en)\in\mathbb{S}}(en - st)$ be the *largest* bin size to be processed from among all bins identified in the previous step, and let $R' = \lceil\sqrt{d}\rceil$. We design the rasterizaton pipeline with a viewport resolution of $R' \times R'$ to efficiently perform this search. For the example in Figure 5(b), $d$ is $4 - 1 = 3$, and $R' = \lceil\sqrt{3}\rceil$. Thus, the viewport is assigned a $2 \times 2$ resolution.

**Vertex Shader.** It simply passes on all the 2D points in $\mathbb{S}$ to the geometry shader.

**Geometry Shader.** Since the viewport has resolution $R' \times R'$, the rasterization pipeline generates fragments corresponding to a $R' \times R'$ virtual image. By definition, this virtual image has at least as many pixels as the maximum bin size to be processed. As was done in index creation, the pixels are organized in row-major order. Let pixel $(0, 0)$ correspond to the value $st$. Then the pixel $(x_{max} = (en - st) \mod R', y_{max} = \lfloor\frac{(en-st)}{R'}\rfloor)$ would correspond to the value $en$. The geometry shader is used to create an axis parallel rectangle (represented as two triangles) whose diagonal end points are $(0, 0)$ and $(R', y_{max})$. Thus, all fragments that fill this rectangle are generated.

**Fragment Shader.** Consider a pixel $(x, y)$ processed by an instance of the fragment shader. This fragment corresponds to the location $i = st + x + y \cdot R'$ in $\mathbb{B}$. If $i \geq en$, this fragment is discarded without further processing. If not, the entry $\mathbb{B}[i]$ is evaluated against the query range to test whether it satisfies the query.

Figure 5(b) illustrates this search process for each entry in $\mathbb{S}$. The grayed out pixels in the $2 \times 2$ viewports correspond to those discarded by the fragment shader. The remaining entries in $\mathbb{B}$ that are considered for processing are indicated by the blue dotted arrows.

Note that our creation of this $\mathbb{S}$-buffer-based search pipeline allows every point in the bins of interest to be processed *independently*, thus maximizing the parallelism.

## 5 EXTENSION TO HIGHER DIMENSIONS

The previous sections addressed the basic case of single-column indexes. We now move on to considering multi-column indexes – specifically, 2D, 3D and nD ($n \geq 4$) indexes, with each class meriting different treatment due to image representation constraints. We begin with the index binning strategy used by the vertex shader during index construction, and then describe how search queries on multiple columns are handled.

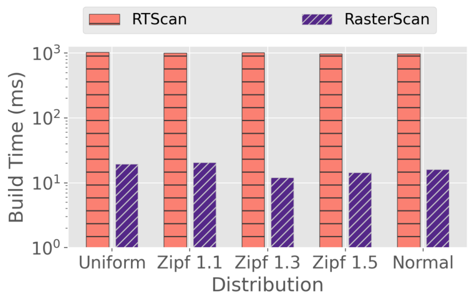


**Figure 6: Index Build Times on RTX 4090**

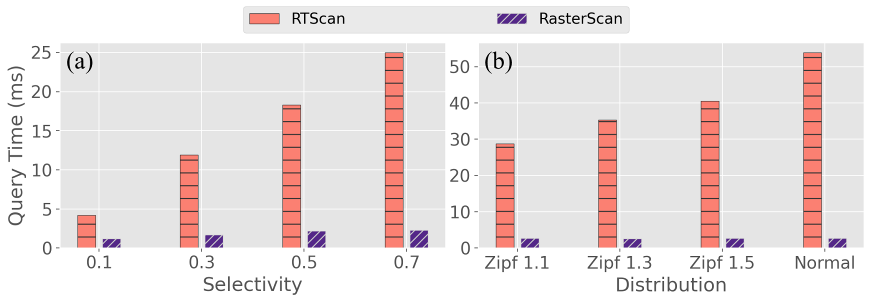


**Figure 7: Index Search Times on RTX 4090**

## 5.1 Index Binning strategy

**2D points:** Let $\mathbb{C}_1 = \{u_i | 0 \leq i < N\}$ and $\mathbb{C}_2 = \{v_i | 0 \leq i < N\}$ be the two jointly-indexed columns. Let $[u_{min}, u_{max}]$ and $[v_{min}, v_{max}]$ be the range of values in $\mathbb{C}_1$ and $\mathbb{C}_2$, respectively. In this case, the range of column $\mathbb{C}_1$ is uniformly divided along the $x$-axis while the range of column $\mathbb{C}_2$ is uniformly divided along the $y$-axis. Specifically, the pixel $p_i(x, y) \in \mathbb{T}$ stores all points $(u, v)$ such that $u \in [x \cdot du, (x+1) \cdot du)$ and $v \in [x \times dv, (x+1) \times dv)$.

Thus, given a row with a 2D value $(u, v)$, the vertex shader transforms this point into the texture space

$$(x, y) = (\lfloor \frac{u}{du} \rfloor, \lfloor \frac{v}{dv} \rfloor) \tag{3}$$

**3D points:** To index 3 columns $\mathbb{C}_1, \mathbb{C}_2,$ and$\mathbb{C}_3$ (i.e., points in 3D), we create a 2D index texture $\mathbb{T}$ using columns $\mathbb{C}_1$ and $\mathbb{C}_2$, as described above. However, while storing the point values in the index buffer $\mathbb{B}$, we store all three dimensions of the corresponding point.

**nD** $(n \geq 4)$ **points:** When more than 3 columns are to be indexed, we fall back on the strategy recommended in [7] – simply create *multiple* raster indexes, each having at most 3 columns. A query can then be answered by individually querying each of these indexes, and then taking the intersection of the results.

## 5.2 Processing 2D and 3D Queries

Vertex and geometry shaders in the texture search pipeline are modified as follows to handle range queries for 2D and 3D indexes.

(1) Let the query ranges for the two columns $\mathbb{C}_1$ and $\mathbb{C}_2$ be $[l_1, r_1]$ and $[l_2, r_2]$, respectively. The vertex shader transforms these values into pixel locations $(x_1, y_1)$ and $(x_2, y_2)$ using Equation 3. This pair of locations is then passed on to the geometry shader.
(2) The geometry shader generates two triangles that make up an axis-parallel rectangle whose diagonal is defined by the line $(x_1, y_1), (x_2, y_2)$ that it receives from the vertex shader.

The fragment shader in the index buffer search pipeline is also modified (according to the dimensionality of the indexed point) to appropriately evaluate the data point with the query range.

## 6 PERFORMANCE EVALUATION

In this section, we first describe our experimental setup, and then present results from our evaluation of RasterScan against RTScan [7], the current state-of-the-art RT-based index. (We do not include RTIndex in our evaluation since it was shown in [7] that RTScan was the better-performing approach.)

## 6.1 Setup

RasterScan is implemented using C++ and Vulkan (which provides support for the graphics pipeline) and is made available at [8]. For RTScan, we used their publicly available code [9]. In order to focus purely on our goal of comparing rasterization against ray-tracing, we *do not* include Bindex-based filtering in our evaluation. However, since the data modeling is similar between RasterScan and RTScan, Bindex-based filtering can be used for RasterScan as well. Further, to have a fair comparison, we ensure that (a) both RasterScan and RTScan use the same input data, encoded as described in [7], and (b) the output structure of RasterScan is identical to RTScan.

**Metrics.** We separately report index build times and index query response times, with the latter representing the time spent on the GPU to execute the queries *after* the indexes have been constructed.

**Testbed.** Baseline comparison experiments were run on a PC having an RTX 4090 GPU with 24 GB video memory (VRAM). We also used an older generation RTX 2080Ti GPU (with 12 GB VRAM), and a low-powered RTX A2000 Laptop GPU (with 4 GB VRAM) to assess the performance of RasterScan on older/weaker hardware.

**Datasets.** We use the same datasets and query workloads that were previously used for evaluation of RTScan in [7]. Specifically, we create a uniformly distributed dataset to test the effect of selectivity, and four skewed datasets, three having a zipf distribution of 1.1, 1.3, and 1.5 respectively, and a fourth skewed dataset with a normal distribution. All these datasets consist of 100 million points and 3 columns that are *jointly* indexed. (Similar performance trends were observed for single-column and two-column indexes.)

## 6.2 Index Build Time

Figure 6 compares the index build performance of the two approaches. We observe that RasterScan builds its index around 50X faster than RTScan. This speedup can be attributed to the BVH tree construction required by RTScan incurring significantly more irregular memory accesses and synchronization on the GPU, as compared to the simple arithmetic and atomic increment operations utilized by RasterScan.

## 6.3 Index Search Time

Figure 7(a) compares the search performance of RasterScan and RTScan on the uniform dataset for different selectivities. We first observe here that RasterScan delivers much faster searches than RTScan. Further, as the selectivity increases, representing queries with more results, the gap between them increases – in fact, at 0.7

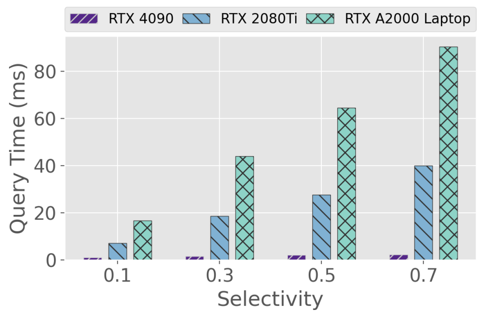


**Figure 8: Different GPUs**

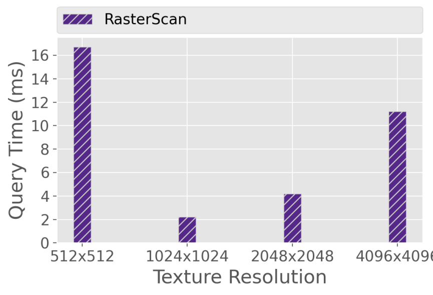


**Figure 9: Resolution Tuning**

selectivity, RasterScan is more than an order-of-magnitude faster than RTScan. Finally, the *slope* of search times for RasterScan is significantly lower than that of RTScan– it mildly increases from 1.2 ms for 0.1 selectivity to 2.3 ms for 0.7 selectivity.

The large performance gap is attributable to the computational overheads incurred by the RT approach – performing BVH traversal for every ray, and performing ray intersection tests with numerous triangles. Whereas RasterScan only performs simple lookup and arithmetic compare operations.

The performance on skewed datasets is shown in Figure 7(b), where all queries have selectivities in the 0.7-0.8 range. We see similar performance trends to the uniform scenario, with RasterScan clearly outperforming RTScan, obtaining well over order-of-magnitude search speedups.

## 6.4 Performance across GPU Architectures

A question that could be asked is whether the above "low-slope" search-time performance of RasterScan could be an artifact of the high-end RTX4090. We therefore evaluated the performance of RasterScan on two other GPUs representing more vanilla hardware: the first is a 4-generation old RTX 2080Ti, and the second a low powered RTX A2000 laptop GPU.

The performance of RasterScan on these three platforms is shown in Figure 8 for the uniform dataset. Unlike the 4090, the impact of selectivity on the vanilla GPUs is much more significant. This can be attributed to two factors: first, these GPUs have a significantly slower memory bandwidth thus increasing the data access / storage times; and second, they have a considerably lower number of compute cores, thus requiring more time to execute the queries.

## 6.5 RasterScan Parameter Tuning

We now consider the effect of the parameter $R$, which is the resolution of the Texture Index $\mathbb{T}$, on the performance of RasterScan. This experiment was conducted on the uniform dataset, and the query selectivity was set to 0.3.

The search time of this query is shown in Figure 9 for four different values of $R$, going from 512x512 to 4096x4096. We observe a "cup-shaped" behavior with the best response time provided by the 1024x1024 configuration. A smaller value of $R$ generates a lower number of search ranges, but each search range is larger since points are now divided among a smaller number of bins. Which means that even though fewer rectangles need to be created for buffer search, all of these rectangles are larger. So, there is a trade-off between the number of rectangles and the sizes of these rectangles – a setting of $R = 1024$ seems to provide a "sweet spot" between these competing factors. We therefore use this resolution for $\mathbb{T}$ in our implementation.

## 6.6 RasterScan Memory Footprint

For all the datasets used in the experiments (100 million rows with 3 columns), our RasterScan index incurred a memory footprint of 1.12 GB. Further, during index build, a tiny amount of additional space (1 MB) was required to store the intermediate data structures (e.g. copy of the index texture). On the other hand, the BVH tree used by the RTScan index takes up 3 GB of memory [7], so clearly the resource overheads of RasterScan are substantially smaller.

Finally, during querying, the RasterScan index consumes a small additional memory (2 MB) to maintain the intermediate structures, apart from the memory required to store the result itself.

## 7 DISCUSSIONS

**Handling other numerical columns.** The current implementation of RasterScan assumes 4-byte integer columns. To support 8-byte integers, single-precision, or double-precision floating-point values, the only required changes are: (1) update the primitive type of the index buffer $\mathbb{B}$; and (2) modify the implementation of the binning strategy in the vertex shader during the index build setup phase to use the appropriate primitive types.

**Handling non-numerical columns.** A straightforward way to extend RasterScan to non-numerical columns is to create a dictionary that maps each unique column value to an integer. If the column type can be ordered (e.g., strings), the set of unique values can be sorted (say lexicographically), and the index of a given string in this sorted list can be assigned as its value.

Queries are then executed by first transforming the predicate using this dictionary before querying the index.

**Index updates.** The index described so far assumed deployment in an OLAP scenario, where the index is created once before analytical queries are executed. This design choice was made for two reasons: (1) We envision GPUs primarily as co-processors to speedup analytical queries; and (2) It allowed for a fair comparison with the RT-based approaches, which also follow the same assumption. Furthermore, even when the index needs to be refreshed, the index creation overhead is low, as seen from the index build times.

However, if necessary, the design of RasterScan can be extended to update the index as follows. The current design stores the index Buffer $\mathbb{B}$ as a contiguous array. Instead, it could be organized as lists of pages with cells in the index texture $\mathbb{T}$ pointing to the corresponding list. An insert operation would then add the row to the appropriate page. In case the page is full, a new page can be appended to the list. A delete would follow a similar process, while an update can be implemented as a delete followed by an insert.

The query algorithm would also need to be modified: during the index texture search stage, the range search buffer $\mathbb{S}$ would be populated with ranges corresponding to all non-empty pages associated with texture cells that satisfy the query range. The index buffer stage of the algorithm remains unchanged.

## 8 CONCLUSIONS

RT hardware is known to be highly useful in computer graphics applications that have arbitrarily oriented triangles and rays. However, both the data and queries in database environments are geometrically well behaved. Therefore, using RT for relational indexing appears to be an overkill due to its inherent computational overheads. Instead, a much simpler rasterization approach, that replaces ray traversals with cheap arithmetic compares, was found to provide substantive build and search speedups of over an order-of-magnitude.

Our focus here was on indexing. However, we expect similar considerations would hold for RT versus rasterization in other database modules, such as query execution [10] and spatial query processing [1], as well. We intend to quantify these assessments in our future work.

## REFERENCES

[1] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2025. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 396–411. doi:10.1145/3710848.3710850

[2] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. doi:10.1145/602259.602266

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

[4] Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (Anaheim, California) *(HPG '13)*. Association for Computing Machinery, New York, NY, USA, 89–99. doi:10.1145/2492045.2492055

[5] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X. Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 909–923. doi:10.1145/3318464.3380563

[6] Yashuai Lü, Hui Guo, Libo Huang, Qi Yu, Li Shen, Nong Xiao, and Zhiying Wang. 2021. GraphPEG: Accelerating Graph Processing on GPUs. *ACM Trans. Archit. Code Optim.* 18, 3, Article 30 (May 2021), 24 pages. doi:10.1145/3450440

[7] 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. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1460–1472. doi:10.14778/3648160.3648183

[8] RasterScan 2025. https://github.com/Microsoft/raster-scan

[9] RTScan 2024. https://github.com/AntaresAlice/RTScan

[10] Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, and Rubao Lee. 2024. RTCUDB: Building Databases with RT Processors. *CoRR* abs/2412.09337 (2024). doi:10.48550/ARXIV.2412.09337

[11] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3* (8th ed.). Addison-Wesley Professional.