RTScan: Efficient Scan with Ray Tracing Cores

Yangming Lv
Fudan University
ymlv21@m.fudan.edu.cn

Kai Zhang*
Fudan University
zhangk@fudan.edu.cn

Ziming Wang
Fudan University
wangzm22@m.fudan.edu.cn

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

Rubao Lee
Freelance Researcher
lee.rubao@ieee.org

Zhenying He
Fudan University
zhenying@fudan.edu.cn

Yinan Jing
Fudan University
jingyn@fudan.edu.cn

X. Sean Wang
Fudan University
xywangCS@fudan.edu.cn

ABSTRACT

Indexing is a core technique for accelerating predicate evaluation in databases. After many years of effort, the indexing performance has reached its peak on the existing hardware infrastructure. We propose to use ray tracing (RT) cores to move the indexing performance and efficiency to another level by addressing the following technical challenges: (1) the lack of an efficient mapping of predicate evaluation to a ray tracing job and (2) the poor performance by the heavy and imbalanced ray load when processing skewed datasets. These challenges set obstacles to effectively exploiting RT cores for predicate evaluation.

In this paper, we propose RTScan, an approach that leverages RT cores to accelerate index scans. RTScan transforms the evaluation of conjunctive predicates into an efficient ray tracing job in a three-dimensional space. A set of techniques are designed in RTScan, i.e., Uniform Encoding, Data Sieving, and Matrix RT Refine, which significantly enhances the parallelism of scans on RT cores while lightening and balancing the ray load. With the proposed techniques, RTScan achieves high performance for datasets with either uniform or skewed distributions and queries with different selectivities. Extensive evaluations demonstrate that RTScan enhances the scan performance on RT cores by five orders of magnitude and outperforms the state-of-the-art approach on CPU by up to 4.6×.

PVLDB Reference Format:
Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X. Sean Wang. RTScan: Efficient Scan with Ray Tracing Cores. PVLDB, 17(6): 1460 - 1472, 2024.
doi:10.14778/3648160.3648183

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

1 INTRODUCTION

Tables are generally stored in a denormalized form in modern data warehouses, where the evaluation of selection predicates dominates the query processing time. Therefore, various indexes and algorithms have been proposed to accelerate scans on CPUs in recent years. Representative scan implementations include Zone Maps [16], Bitweaving [14], and Column Sketches [9]. As the state-of-the-art scan approach, BinDex [13] achieves up to 2.9× higher performance than the runner-up. By limiting the number of memory accesses and trading more memory for higher performance, BinDex leaves little room for further optimizations on the CPU.

Current commodity GPUs adopt ray tracing (RT) cores to accelerate the real-time rendering of three-dimensional scenes. RT cores launch rays into the space to quickly locate intersected objects. With a set of optional shaders, RT cores can perform user-defined actions when a ray intersects an object, which offers flexibility to implement diverse tasks. There is a spectrum of studies that utilize RT cores to accelerate data processing, e.g., K-nearest neighbor search [18, 32], DBSCAN [17], and range minimum queries [15]. When utilizing RT cores for data processing, data are represented by primitives like triangles or spheres that are placed in a three-dimensional space. Queries are transformed into rays that intersect the primitives for fast searching. Since only the data that satisfies the specified conditions needs to be touched, RT cores have demonstrated the potential to accelerate database operations.

A straightforward transformation of a scan into a ray tracing job can be highly inefficient. For instance, RTIndex [8] represents each value in a column as a triangle, where the value defines its X-coordinate in the space. For a predicate $x > a$, RTIndex launches a ray parallel to the X-axis from $(a, 0, 0)$ to $(X_m, 0, 0)$, where X_m is the maximum value in the column. The ray would intersect any triangle whose represented value satisfies the predicate, but its performance is orders of magnitude lower than that of BinDex [13].

Such a straightforward transformation of scan on RT cores fails to utilize the hardware effectively. We have identified the following reasons for the low performance: 1) When evaluating a predicate with a high selectivity on a large dataset, rays have an enormous number of intersections to process, resulting in extremely low performance. 2) For datasets with non-uniform data distributions, the workload of rays is severely imbalanced, where some rays may have to handle much more intersections than others. 3) The simple transformation of the scan to a ray tracing job lacks sufficient parallelism, leaving the RT cores underutilized. Overall, RT cores achieve high performance only when launching a massive number of rays with a light and balanced load, which demands an effective job transformation with an elaborate design for scans.

In this paper, we propose RTScan, an approach that efficiently accelerates scans on RT cores. First, RTScan transforms the evaluation of conjunctive predicates into a ray tracing job in a three-dimensional space. All data records are built as cubes and placed in a three-dimensional space, while the conjunctive predicates are

* represents the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648183

transformed into a set of rays to intersect all the data records in the query area. By evaluating multiple predicates simultaneously, RTScan significantly enhances the scan efficiency by avoiding the multiple data accesses in columns as traditional scans. Moreover, with data records being scattered in a three-dimensional space, more rays can be launched for better parallelism. Second, RTScan proposes a series of techniques to address the inefficiencies of the existing implementation, which makes scans on RT cores extremely fast. The three main techniques are as follows. (1) *Uniform Encoding*: For a dataset with skewed distribution, RTScan matches the data ranges of the attributes in a group and encodes the values to map them uniformly into the space. It balances the load of launched rays for datasets with skewed data distributions. (2) *Data Sieving*: To avoid the large amounts of intersections for predicates with high selectivity, a set of sieving bit vectors is built to store approximate results for predicates. By selecting a vector that contains the approximate results for a predicate, RT cores only need to intersect records in a small region, which significantly alleviates the ray load. (3) *Matrix RT Refine*: RTScan proposes a specific way of casting rays, which adopts cubes as the primitive to reduce the costs for intersection tests and launches short rays with spacing to lighten the traversal costs in the index for primitives. With the above techniques, RTScan notably enhances the scan efficiency.

The contributions of this paper are as follows:

- We propose RTScan, an approach that maps the evaluation of conjunctive predicates to an efficient ray tracing job in a three-dimensional space.
- We propose three techniques to optimize the scan performance, namely Uniform Encoding, Data Sieving, and Matrix RT Refine, which not only enhance the parallelism but also effectively lighten and balance the load of rays.
- We implement the prototype of RTScan and intensively evaluate its performance under diverse workloads and configurations.

Experimental results show that RTScan significantly improves the scan performance with RT cores and achieves up to 4.6× higher performance than the state-of-the-art CPU-based approach.

2 BACKGROUND AND MOTIVATION

2.1 Background of Ray Tracing

Ray tracing is a rendering technique for generating highly realistic lighting effects, which simulates the behavior of rays as they interact with objects[20]. To accelerate the ray tracing performance, the latest commodity off-the-shelf GPUs incorporate RT cores, which are specialized hardware to quickly search primitives and calculate the ray intersections. Besides the mainstream desktop and workstation GPUs from NVIDIA and AMD, e.g., NVIDIA's RTX 40 series and AMD's RX 7000, data center GPUs including NVIDIA A40 and T4 also support ray tracing. Moreover, the Apple M3 and A17 Pro chip have integrated RT cores to support hardware-accelerated ray tracing. The NVIDIA OptiX is a programming framework for accelerating ray tracing algorithms with RT cores. With the abstraction from OptiX, a ray tracing job primarily involves two types of entities: **rays** and **primitives**. Ray tracing renders a three-dimensional scene containing a set of primitives. The primitives include triangles, spheres, and custom primitives defined by programmers. A ray is parameterized by an origin coordinate O and a direction vector D [25]: $R(t) = O + tD$, where t is a range that specifies the ray segment on the line.

In OptiX, primitives are wrapped by bounding volumes, like Axis-Aligned Bounding Boxes (AABBs). AABBs are organized as a Bounding Volume Hierarchy (BVH) tree to accelerate traverse. RT cores traverse the BVH tree and identify ray intersections with primitives, where the *intersection program* in OptiX is called to check if any primitive in a leaf node of the BVH tree is intersected. If an intersection is detected, the user-defined function in the *intersection program* is called to process the intersection information and execute user-defined actions. There are also the *closest hit program* and the *miss program*, which are called when a ray intersects the closest primitive or doesn't intersect any primitive, respectively. These programs offer great flexibility for RT cores to implement various data processing jobs. Therefore, RT cores have been utilized to accelerate data processing, including K-nearest neighbor search [3, 18, 32], DBSCAN [17], and minimum range queries [15].

2.2 Utilizing RT Cores for Scan

RTIndex [8] is a study that uses RT cores for data scans. In RTIndex, values are represented by triangles in the space. For a value k, the vertexes of its represented triangle are $(k - 0.5, 0.5, -0.5)$, $(k - 0.5, -0.5, -0.5)$, and $(k + 0.5, 0, 0.5)$. There are different ways to cast rays for queries in RTIndex. As shown in Figure 1a, for a point query $x = a$, a ray is cast perpendicular to the X-axis. The ray originates at $(a, 0, -\epsilon)$, and the ray length is 2ϵ, where ϵ is a small value so that the origin of the ray is close to the X-axis. Figure 1b demonstrates how a range query is evaluated in RTIndex. For $a \le x \le b$, a ray is cast in parallel to the X-axis. It originates at $(a - \epsilon, 0, 0)$ and ends at $(b + \epsilon, 0, 0)$. When there is an intersection, the Any-Hit Program is called to fetch the record ID and set the result.

For point queries, RTIndex achieves slightly higher performance than B-Tree on GPU [1] and Warpcore [11]. However, RTIndex performs poorly for range queries, where BinDex achieves an average of 22595× higher performance than RTIndex on a uniform dataset with 1×10^8 data. The main reason is that RTIndex cannot effectively exploit the parallelism of RT cores since it only launches one ray for a range query, while the GPU contains 82 RT cores. Consequently, when the selectivity is 0.5, RTIndex has to handle around 5×10^7 intersection tests with only one ray, where parts of the traversal and intersection tests are serial[1].

2.3 RTc1: An Attempt to Improve the Scan Performance

We make our first attempt to enhance scan performance and build RTc1, an implementation that enhances the parallelism of range queries. As shown in Figure 1c, RTc1 still uses triangles as the primitives. The coordinates of a triangle that represents a value k are $(k + s, 0, 0)$, $(k, 0, s)$, and $(k, s, -s)$, where s defines the size of the triangles. The length of the line segment on the X-axis that overlaps with the projection of a triangle on the XZ plane is equal to s. As shown in Figure 1c, for a predicate $a < x < b$, instead of casting a ray in parallel to the X-axis like RTIndex, RTc1 casts $(b - a)/l$ rays

[1] https://forums.developer.nvidia.com/t/does-a-ray-traverses-the-acceleration-structure-in-parallel-when-calling-optixtrace/264585 (last accessed 2024/2/18)

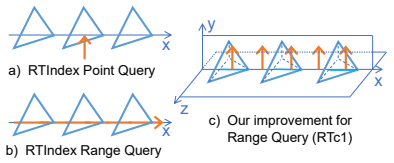

Figure 1: How RTIndex and RTc1 work

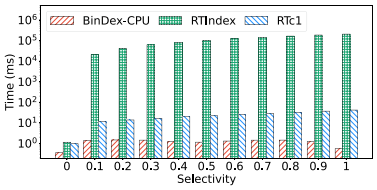

Figure 2: Performance comparison of range scan on one column

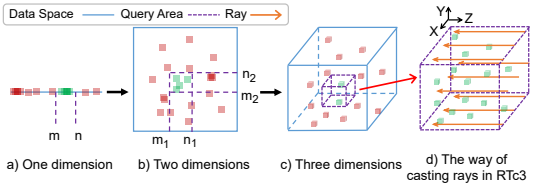

Figure 3: Data distribution in multi-dimensional spaces

perpendicular to the XZ-plane from $(a, 0, 0)$ to $(b, 0, 0)$ in a fixed interval $l (l = s - \epsilon)$. The i-th ray originates from $(a + i \cdot l, 0, 0)$, and its length is equal to l. The interval between two adjacent rays equals l because some primitives will be missed if the interval is larger, while there will be repetitive intersections with a smaller interval. Comparing to RTIndex, RTc1 launches more rays to exploit the parallelism from RT cores.

Figure 2 compares the performance between RTc1, RTIndex, and BinDex on a uniform dataset with 10^8 data records. In the experiments, the CPU is Intel Core i9-10900K, and the GPU is NVIDIA Geforce RTX 3090. RTc1 effectively enhances the RT performance and its optimal configuration achieves an average of 3859.6× performance improvement over RTIndex. Taking selectivity = 0.5 as an example, RTc1 launches 9×10^3 rays. With a total of 5×10^7 intersection tests, each ray handles 5.6×10^3 tests on average, which is four orders of magnitude less than that in RTIndex. However, RTc1 is still, on average, 20.3× slower than BinDex, which is the state-of-the-art implementation on the CPU.

3 RT FOR SCAN WITH CONJUNCTIVE PREDICATES AND THE CHALLENGES

In this section, we make our second attempt by transforming a scan with conjunctive predicates into a ray tracing job in a three-dimensional space and analyzing its performance.

3.1 Opportunities of RT Cores for Conjunctive Predicates and Our Second Attempt: RTc3

In modern data warehouses, tables are stored in denormalized forms, where the evaluation of selection predicates dominates the query processing time [10]. Moreover, decision support queries generally involve many conjunctive predicates, like $p_1 \bigwedge p_2 \bigwedge ... \bigwedge p_n$. Because common subexpression elimination (CSE) is a widely adopted optimization that combines the evaluation of predicates on the same attribute, we consider conjunctive predicates after CSE so that each predicate evaluates a unique column.

RT cores are designed for performing ray operations in a three-dimensional space, perfectly fitting for evaluating conjunctive predicates on three columns. Therefore, we make our second attempt and propose RTc3, which evaluates conjunctive predicates on three columns with only one RT job. In RTc3, cubes are used as the custom primitive, which is more effective in reducing the number of intersection tests (Please refer to Section 4.4 for detailed discussion). For each data record, the values of the three attributes are used as the coordinates of the center point of the corresponding cube in the space. With three conjunctive predicates, a cuboid region in the space needs to be scanned, where each predicate defines one side of the cuboid (Figure 3c). For a query in RTc3, a set of rays perpendicular to the XY-plane is cast into the cuboid (Figure 3d). For example, for a query $(x < a) \wedge (y < b) \wedge (z < c)$, a cuboid from $(x_{min}, y_{min}, z_{min})$ to (a, b, c) needs to be scanned, where x_{min} denotes the smallest value of the column. $a \cdot b/k^2$ rays are cast perpendicular to the surface extending from $(x_{min}, y_{min}, z_{min})$ to (a, b, z_{min}), where k is the ray interval in the X and Y directions. The ray length is set to $c - z_{min}$ to cover the entire cuboid. By setting the cube width to be the ray interval plus ϵ, all the primitives in the query area can be intersected. If a ray intersects a cube, the represented data record is checked to confirm that all three predicates are satisfied.

There are two main advantages of using RT cores for evaluating conjunctive predicates. First, instead of placing primitives only on the X-axis, RTc3 scatters primitives in the three-dimensional space. Therefore, more rays can be launched in the cuboid to enhance the parallelism. Second, RTc3 reduces the amount of data to scan. There are a spectrum of studies that optimizes the evaluation of conjunctive predicates [12, 19, 23, 31]. Assuming the selectivity for the kth predicate is S_k. If three conjunctive predicates are evaluated with logical-and (&), the overall number of data to scan is $(S_1 + S_2 + S_3) \cdot N$, where N is the total number of data records. For approaches with branching-and(&&), the evaluation starts from the column with the lowest selectivity, and the amount of scanned data is $(S_1 + S_1 \cdot S_2 + S_1 \cdot S_2 \cdot S_3) \cdot N$. Instead, RTScan touches a much smaller number of data records. As shown in Figure 3b, if two conjunctive predicates are mapped to a two-dimensional space, the number of data records to scan becomes $S_1 \cdot S_2 \cdot N$. Therefore, the number of data to scan in RTc3 is $S_1 \cdot S_2 \cdot S_3 \cdot N$ (Figure 3c), which is much smaller than the previous approaches.

3.2 Evaluation and Analysis of Conjunctive Predicates on RT Cores

We evaluate and analyze the performance of RTc3, RTc1, and BinDex on various workloads. In evaluating BinDex and RTc1, we perform scans on columns separately and merge their results with *bitwise AND* operations. Through experiments and analyses, we find three main challenges to building efficient scans on RT cores.

Challenge 1: Poor Performance with Large Amounts of Intersection Tests. Figure 4 compares the performance of RTc3, RTc1, and BinDex on a dataset with a uniform data distribution. The *selectivity* on the X-axis denotes the selectivity of each predicate. The performance of an RT job is closely related to the number of intersection tests involved, which depends on the dataset volume and the predicate selectivity. RTc3 achieves up to 23.8× performance

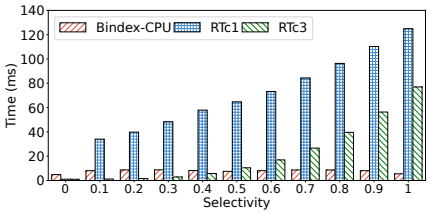

Figure 4: Performance of RTc3 on a dataset with uniform data distribution

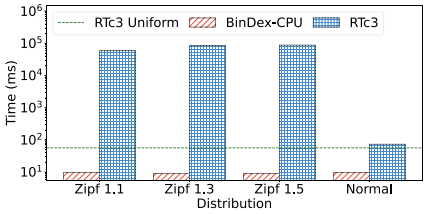

Figure 5: Performance of RTc3 on a dataset with skewed data distribution

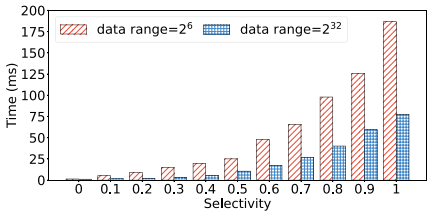

Figure 6: Performance of RTc3 on a dataset with small data ranges

improvement over RTc1. Taking selectivity = 0.2 as an example, RTc1 casts 1.0×10^4 rays and has 6.0×10^7 intersection tests, which means each ray handles around 5574 intersection tests on average. Instead, RTc3 casts 6.4×10^3 rays and has 8.7×10^5 intersection tests, where each ray only has to handle 136 intersection tests on average. Compared to BinDex, RTc3 is 3.0× faster for predicates with a selectivity of less than or equal to 0.4. However, for selectivities larger than 0.4, BinDex achieves much higher performance than RTc3. This is because a higher selectivity leads to a larger query area, where the number of rays and intersections increase sharply. For example, 1.3×10^5 rays are launched with 7.9×10^7 intersection tests when the selectivity is 0.9. Consequently, handling two orders of magnitude more intersections breaks the advantage of RT cores.

Challenge 2: Imbalanced Load with Skewed Data Distribution. Figure 5 demonstrates the performance with skewed workloads. There are three datasets following Zipf distribution with $\alpha = 1.1/1.3/1.5$ and a dataset following normal distribution with $\mu = 2^{31}$ and $\sigma = 1$. The selectivity of each predicate in the evaluation is 0.9. The green dotted line shows the processing time of RTc3 on a uniform dataset with the same selectivity, which is around 1037× lower than that with skewed distributions on average. In skewed data distribution, a large amount of primitives cluster and overlap with each other in a few small regions. As a result, only a few rays intersect with primitives, and the rays need to handle a large number of intersections. Taking the Zipf distribution with $\alpha = 1.3$ as an example, 90% records are aggregated in $[0, 1242]$ while only 10% are scattered in $[1242, 2^{32}]$. To optimize the average performance for predicates with varying selectivities, the ray interval is set as $2^{32}/400$. As a result, when the selectivity equals 0.9, only one ray is cast in the range of $[0, 1242]$, and it needs to handle all the 9.8×10^7 intersection tests. Therefore, a skewed dataset results in an imbalanced ray load and, thus, low performance.

Challenge 3: Poor Parallelism for Attributes with Small Data Ranges. An attribute's data range equals the difference between the maximum and the minimum value. When there are many data records with a much smaller data range, there would be repetitive values in a column. As a result, only a limited amount of rays can be launched for the small data range, while each ray has to make a large number of intersections. Figure 6 compares the performance of RTc3 on datasets that contain 10^8 data records but with different data ranges, i.e., $[0, 2^6)$ and $[0, 2^{32})$. When we say the data range is $[0, 2^6)$, the data ranges of all the three attributes are from 0 to 2^6. As shown in the figure, because of the limited parallelism in a smaller space, RTc3 with the data range of $[0, 2^6)$ shows 22% - 3.7× lower performance than that with the data range of $[0, 2^{32})$.

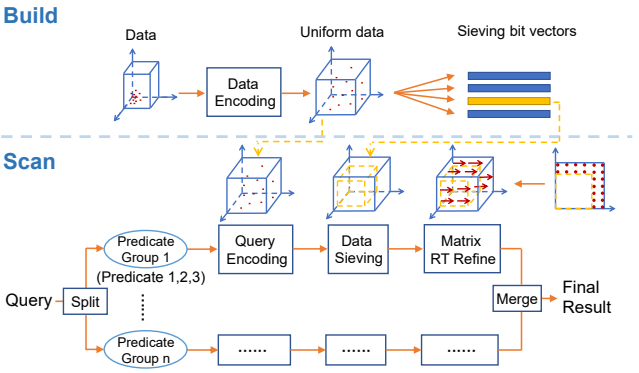

Figure 7: The workflow of RTScan

Overall, the above challenges need to be addressed to fully exploit the capability of RT cores.

4 RTSCAN DESIGN AND IMPLEMENTATION

4.0.1 The Approach of RTScan. After developing and evaluating RTIndex, RTc1, and RTc3, we analyze the inefficiencies of these implementations and propose RTScan, which proposes the following three main techniques to address the challenges.

- *Uniform Encoding* uniformly distributes data records in the space to reduce the number of ray intersections and balance the ray load. Meanwhile, the data range is adjusted to enable sufficient parallelism.
- With the encoded base data, *Data Sieving* generates an approximate result to reduce the number of intersected data records dramatically. With Data Sieving, only a small region in the space needs to be detected by rays for queries with any selectivities.
- *Matrix RT Refine* uses cubes as the primitive and proposes a way of launching rays with a specific interval and spacing to enhance the parallelism while reducing repetitive ray intersections and the BVH traversal overheads.

4.1 Overview

4.1.1 The Workflow of RTScan. The workflow of RTScan is shown in Figure 7. In index-building, each attribute is encoded with Uniform Encoding by CPU to map the data uniformly in an appropriate range, then the encoded data are transferred into the GPU device memory. The encoded attribute are composed as attribute groups,

where each attribute group consists of three attributes. A BVH (Bounding Volume Hierarchy) tree is built for each attribute group to map the data in a three-dimensional space with customized primitives. After that, sieving bit vectors are built for each encoded attribute and stored in the GPU memory. The main index building procedures, i.e., building the BVH tree and the sieving bit vectors, are executed within the GPU memory. The constructed BVH trees for different attribute groups can be cached in the GPU memory and loaded to perform ray tracing jobs when needed[2].

Conjunctive predicates are firstly divided into several predicate groups according to the pre-built BVH trees. When the number of conjunctive predicates is not divisible by three, the last predicate group may have one or two predicates. For each predicate group, its BVH tree in the GPU memory is set for RT cores to traverse. The procedure of evaluating each predicate group consists of three steps. First, each predicate is mapped to the space of the uniformly encoded data. Second, sieving bit vectors for the involved attributes are selected to get the approximate results, which filter a large portion of the data. Third, Matrix RT Refine launches rays to intersect cubes in the remaining areas to refine the approximate results by Data Sieving. With the evaluation results from all predicate groups, their results are merged by CUDA cores as the final result with *bitwise AND* operations. With the techniques, RTScan achieves high efficiency for conjunctive predicates on diverse workloads.

4.2 Uniform Encoding

Uniform Encoding encodes the values of each attribute to address challenges 2 and 3 by launching more rays with a balanced load. Uniform Encoding aims at 1) mapping each attribute in a predicate group to have an appropriate data range and 2) mapping data in each attribute to be uniformly distributed in the data range. When achieving the two goals, Uniform Encoding guarantees that the encoded data still preserves the relative orders in each attribute. The order-preserving property is to ensure that a query on the encoded data can still get correct results.

The procedure of Uniform Encoding is as follows. First, the data range of each column is set the same as N, where N is the total number of data records in the denormalized table. Second, the original data in each column are sorted to get their orders in the column. For example, a column with data $\{5, 9, 1, 6\}$ is sorted as $\{1, 5, 6, 9\}$. According to the sorted order, the original data is encoded as $\{2, 4, 1, 3\}$ with a data range of 4. '5' is encoded as '2' because it is the second smallest number after sorting. A mapping table is built as $\{(1, 1), (5, 2), (6, 3), (9, 4)\}$ to translate the original data to the encoded data. For a predicate $x > 5$ on the original data, a binary search is conducted on the mapping table to locate $(5, 2)$, and the query is encoded as $x > 2$ for evaluation.

For a dataset with a skewed distribution, a value may appear multiple times in a column. Uniform Encoding uniformly maps all the values in the space. Thus, repeated values will be encoded as different values. Figure 8 demonstrates a dataset with nine values consisting two repetitive '3's and six '5's. The values are first sorted with their order being recorded. According to their sorting order, the six '5's are encoded to the interval [4,9], and the mapping table

[2]https://forums.developer.nvidia.com/t/access-multiple-bvh-parallel/260142 (last accessed 2024/2/18)

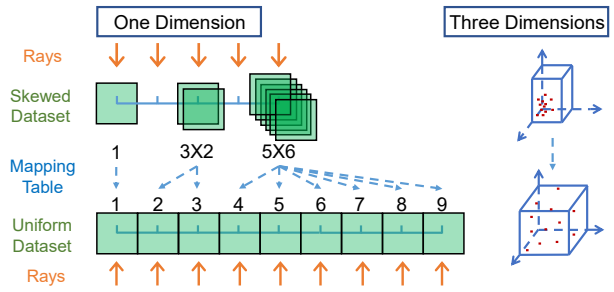

Figure 8: Uniform encoding

is constructed as $\{(1, 1), (3, [2, 3]), (5, [4, 9])\}$. With the mapping scheme, the predicate $x < 5$ is mapped to $x < 4$, which is the lower bound of the interval. Instead, the predicate $x > 5$ is mapped to $x > 9$, which takes the upper bound of the interval. For the query $x = 5$, it is transformed to a BETWEEN operator, i.e., $4 \leq x \leq 9$.

Figure 8 illustrates the effects of uniformly distributing data in a broader range with Uniform Encoding. With a ray interval of 1, there are five rays launched, with the last ray having six intersections and the third ray having two intersections. Instead, with Uniform Encoding, nine rays are launched with one intersection per ray. After performing the Uniform Encoding for each attribute, data records are distributed sparsely in the space. The technique not only enhances the parallelism by launching more rays but also balances the load of the rays for higher performance.

4.3 Data Sieving

With a high selectivity or a large dataset, RT cores need to intersect a large volume of data, leading to low performance. To achieve high performance for various workloads, we propose Data Sieving, which stores approximate results in a set of bit vectors to significantly reduce the number of data records to be scanned.

4.3.1 Bit Vectors for Approximate Results. RTScan uses binning to generate a set of bit vectors F for each column, denoted as $F = \{F_1, F_2, ..., F_k\}$. The bit vector F_i pre-stores an order-preserving array consisting of N bits, where N is the number of data records in the table. Each bit in a bit vector F_i indicates whether the corresponding value in the column matches the predicate $x < X_i$ or not, where X_i marks the represented range of F_i. For datasets with uniform distributions, the data range is uniformly partitioned to build vectors. For example, if the data range is $[1, N]$ and the number of bit vectors is k, the difference between the range of each bit vector is $I = N/k$. The sieving bit vectors will be $F = \{F_1 : x < (1 + I), F_2 : x < (1 + 2 \times I), ..., F_k : x < (1 + k \times I)\}$.

Figure 9 illustrates how the vectors are built for a column of data. Because 41 (the fifth value) in the raw data is greater than 40, the fifth bit in F_1 is set to 0, while the corresponding bits in the rest vectors are set to 1. With these bit vectors, the approximate results of a predicate on the column can be obtained. For example, for a predicate $x < 80$, the bit vector $F_2 : x < 70$ can be chosen as the approximate result. However, the bits representing values in $[70, 80)$ are incorrect as the result, which needs to be corrected with ray tracing. Other operators are also supported with the same set of vectors. For example, for a predicate $x > 80$, the bits in $F_2 : x < 70$

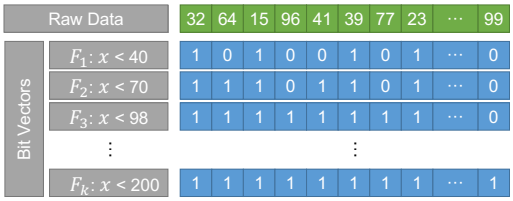

Figure 9: Sieving bit vectors for one attribute

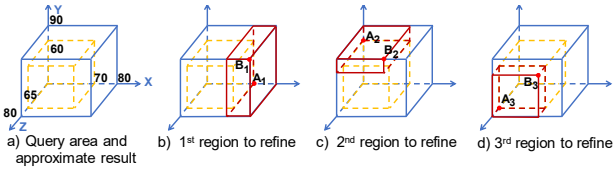

Figure 10: RT refining after Data Sieving with three '<'s

are flipped to represent $x >= 70$; therefore, only values in $[70, 80]$ need to be scanned to refine the results.

The idea of using bit vectors to filter data is also adopted in BinDex. Differently, due to the encoded data in RTScan, we build bit vectors with the information from the previous Uniform Encoding stage. For the originally skewed datasets, although the data distribution of the encoded data is uniform, a uniform partitioning would make repetitive numbers be represented in two or more vectors. For instance, if the encoded data in Figure 8 is built with two vectors $F_1 : x < 5$ and $F_2 : x < 10$, the six '5's in the original data are contained in both the vectors. As a result, for a query $x >= 5$, both vectors still need to be refined to get the final results. To address this issue, RTScan uses the mapping table to pass the data distribution information for building the vectors. After a uniform partitioning of the data range, the borders of each vector are checked through the mapping table, and RTScan adjusts the vectors to keep repetitive values in the same vector.

4.3.2 RT Refine after Data Sieving. After Data Sieving is performed on the three conjunctive predicates, the region to be scanned in the space dramatically shrinks. Figure 10a demonstrates the evaluation of three conjunctive predicates with the '<' operator. The blue cuboid represents the query area of the predicate group, and the approximate result is the yellow cuboid with dotted lines. For example, if the predicates in a predicate group are $x < 80, y < 90, z < 80$, the coordinates of the diagonal line of the blue cuboid are from (0, 0, 0) to (80, 90, 80). With the selected approximate bit vectors for the three columns as $F_x : x < 70, F_y : y < 60, F_z : z < 65$, the diagonal line of the yellow cuboid is from (0, 0, 0) to (70, 60, 65). With the approximate results, RTScan only needs to refine the results in other areas of the blue cuboid.

With Data Sieving, the original query area of the predicate group is transformed into three much smaller regions, i.e., 1) from $A_1(70, 0, 0)$ to $B_1(80, 90, 80)$ in Figure 10b, 2) from $A_2(0, 60, 0)$ to $B_2(70, 90, 80)$ in Figure 10c, 3) from $A_3(0, 0, 65)$ to $B_3(70, 60, 80)$ in Figure 10d. With rays being launched to detect values in the three red cuboids, it significantly reduces the amount of data to be scanned. As a result, with fewer intersections to handle, the overall performance is notably improved. Please note that for other operators such as '>' or *BETWEEN*, the refine areas can be different from that shown in the figure, where three *BETWEEN* predicates will lead to six small regions for refining.

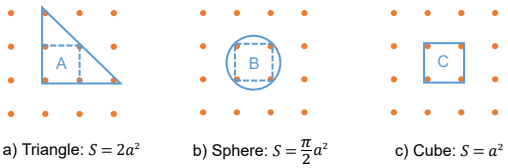

Figure 11: Projection of primitives on a 2D plane

4.3.3 Performance Optimization. **Positive and negative selection:** After predicate encoding, for a query $x < C$, two vectors $F_i : x < X_i$ and $F_{i+1} : x < X_{i+1}$ can be chosen as its approximate results, where $X_i < C < X_{i+1}$. RTScan improves the performance by choosing the vector that leads to fewer ray intersections. Since the encoded data is uniformly distributed, the number of values to refine can be estimated and compared by calculating the data ranges. If $C - X_i < X_{i+1} - C$, F_i is chosen as the result, we call it a *Positive Selection*. In this case, RT cores must refine the bits in $[X_i, C)$. Instead, if $C - X_i \geq X_{i+1} - C$, F_{i+1} is chosen, the values in the range $[C, X_{i+1})$ need to be scanned. This is called *Negative Selection* in RTScan. RTScan adaptively selects vectors to minimize the number of intersections to handle.

The number of sieving vectors: Generally, the more sieving bit vectors are built, the finer granularity the data are partitioned. Correspondingly, with less data to be refined for a query, a higher performance can be achieved. Specifically, with K sieving vectors on N values, the mathematical expectation of the number of records to be refined is $N/4K$. Since more vectors occupy more memory while a GPU has limited memory space, tradeoffs should be made between memory consumption and performance.

4.4 Matrix RT Refine

For refining the rest regions after Data Sieving, Matrix RT Refine is designed to increase the parallelism for RT cores while further reducing the number of intersections.

4.4.1 Optimization Goal. Through experiments and analysis, we find that there are two main costs in the ray tracing process: 1) *Intersection Cost* that comes from intersection tests of primitives in the leaf nodes of the BVH tree, where the Intersection Shader is called; 2) *Traversal Cost* that comes from the rays that traverse the non-leaf nodes in a BVH tree, which is performed by the RT cores. Therefore, our optimization goal in Matrix RT Refine is to reduce the number of intersection tests while reducing the traversal costs. The key to reducing the number of intersection tests is to avoid repeated intersections of a primitive by different rays, which lies in the primitive selection and the ray interval. Differently, the key to reducing the traversal costs lies in reducing the total ray length. It is because a longer ray will touch more areas in the space, causing more traversals in the BVH tree. Therefore, the design of Matrix RT Refine is mainly based on the two optimization goals.

4.4.2 Primitive. To reduce the number of intersections, RTScan uses a cube as the primitive for ray tracing. Among the commonly

Algorithm 1 Pseudo-code of Intersection Shader

Input: predicates P_a, P_b, P_c, result bit vector V
Output: result bit vector V

```
1: primIdx ← get_primitive_index()
2: [a, b, c] ← get_column_value(primIdx)
3: if satisfy(a, P_a) and satisfy(b, P_b) and satisfy(c, P_c) then
4:     set_bit_atomic(V, primIdx)
5: end if
```

used primitives, Figure 11 shows the projection of a triangle, a sphere, and a cube on a 2D plane. The orange points represent a matrix of rays perpendicular to the projection. Rays have the same fixed intervals to intersect all primitives in a querying area. Assume the ray launching interval is a. To guarantee that a primitive can be intersected by at least one ray, the minimum projection area of a triangle is $2a^2$, while that of a sphere and a cube are $\frac{\pi}{2}a^2$ and a^2, respectively. Since primitives can be anywhere in space, a larger area increases the possibility of being intersected by multiple rays. Therefore, since a cube has the lowest projection area, RTScan adopts a cube as the primitive to alleviate the load of RT cores for intersections. For each data record, the encoded values of the three attributes are used as the coordinates of the center of its cube.

When the Intersection Shader is called, a primitive is tested to determine whether the associated data record satisfies the predicates with user-defined conditions. Algorithm 1 shows the pseudo-code of the Intersection Shader. If the values of the data record match all the corresponding predicates, the result bit vector from Data Sieving is refined atomically with the record's ID.

4.4.3 Ray Interval and Ray Spacing. With a smaller query area after Data Sieving, the number of rays that can be launched also becomes less. To minimize the intersection costs while ensuring that any cube in the query area can be intersected by at least one ray, the cube width w should equal the ray interval I plus ϵ. This is because cubes that satisfy the predicates may be missed with $I > w$ while a cube may be repeatedly hit by more than one ray with $I < w$. Therefore, a way of launching more rays is to have a smaller cube width and, thus, a smaller ray interval. However, simply launching more rays with a smaller cube width may lead to low performance. This is because each ray will traverse the BVH tree as it travels through the query area, which incurs higher BVH traversal costs.

To address the issue, Matrix RT Refine proposes a new way to cast rays, which increases the ray number to enhance parallelism while alleviating the traversal overhead. Figure 12 shows the Matrix RT Refine adopted in RTScan. From the view of the projection on the XY-plane, RTScan launches a matrix of rays with a fixed *interval*, similar to that in RTc3. Differently, RTScan further optimizes the performance by launching fewer rays from the view of the XY-plane but many shorter rays in the ray direction with a fixed *spacing*. First, Matrix RT Refine enhances parallelism as long rays are divided into short rays. Second, the BVH tree traversal overhead is alleviated by reducing the overall ray length with spacing. The added spacing equals the cube width minus ϵ, which avoids missing any primitive. Figure 13a shows how to launch rays in RTc3, where one ray has three intersections for three cubes. Figure 13b launches three connected rays, but there are a total of five intersection tests

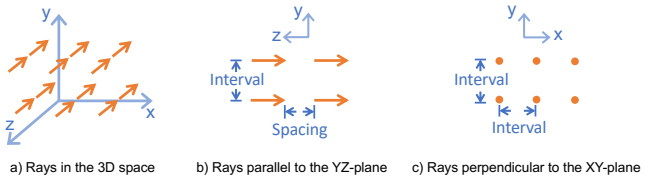

Figure 12: Ray interval and spacing

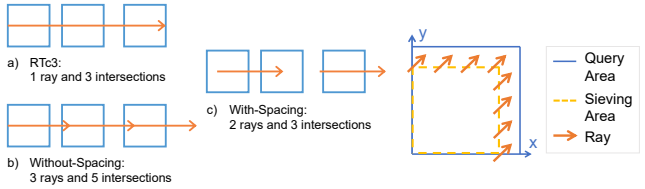

Figure 13: The effect of adding spacing to rays

Figure 14: RTScan on two conj. predicates

since the second and third primitive are intersected by two rays. Figure 13c demonstrates how RTScan launches rays, where two rays make three intersections but a smaller total length than Figure 13a. Overall, with a matrix of rays cast in specific intervals and spacing, all cubes in the querying area can be detected with a lighter and more balanced load.

4.4.4 Analysis of the Precision Loss. OptiX only supports single-precision floating-point numbers as the coordinates. However, due to the encoding scheme of float, there would be precision loss when transforming an integer larger than 2^{23} into a float. For a data range of $(0, 2^K)$ where $K > 23$, we calculate the maximum precision loss as follows. For an integer $(1.xxx...x)_2 \times 2^S$ which can be represented by float, the nearest integer that can be represented is $(1.xxx...x-0.000...1)_2 \times 2^S$. So the precision loss for the two integers is $(1.xxx...x)_2 \times 2^S$ - $(1.xxx...x - 0.000...1)_2 \times 2^S$ = $(0.000...1)_2 \times 2^S$ = 2^{S-23}. Because the gap 2^{S-23} is monotonically increasing, the maximum precision loss for an integer in the range $(2^{23}, 2^K]$ is 2^K - $(1.111...)_2 \times 2^{K-1}$ = 2^{K-24} after being transformed into a float. Considering all the potential precision losses for rays and AABB vertexes, RTScan sets the ϵ as 2 times the maximum precision loss, which is 2^{K-23}. For instance, with $K = 2^{27}$, the maximum precision loss is 8, and ϵ equals 16.

4.4.5 RTScan for One or Two Predicates. RTScan divides conjunctive predicates of a query into multiple predicate groups, and the last group may contain one or two predicates. When scanning one column with RTScan, cubes are placed along the X-axis like the one-dimension case in Figure 8, where the X-coordinate of the cube center is used to represent a data record. After Data Sieving, only cubes on a line segment on the X-axis need to be refined. Like RTc1, a series of short rays are launched perpendicular to the line segment, with a fixed interval equaling the cube width minus ϵ.

When evaluating two conjunctive predicates, a data record is represented by the X-coordinate and the Y-coordinate of its cube center. As shown in Figure 14, the results for the cubes in the orange rectangle are obtained after Data Sieving. Therefore, only the area between the blue rectangle and the orange rectangle needs to be

scanned. Rays are launched perpendicular to the XY-plane, with a fixed interval equaling the cube width minus ϵ. The ray length equals the cube width to intersect all cubes in the area.

5 EXPERIMENTAL ANALYSIS

5.1 Experiment Setup

Hardware and Software We run experiments on a server equipped with an Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz, 32GB DDR4 DRAM, and an NVIDIA GeForce RTX 3090 with 82 RT cores, 10496 CUDA cores, and 24GB VRAM. The operating system is 64-bit Ubuntu Server 20.04 with Linux Kernel 5.15.0-72-generic. The GPU programming interface uses CUDA 12.1 and OptiX 7.5.

Workloads We generate uniform and skewed datasets for evaluation in the experiments, each with 1×10^8 data records. For the uniform workload, the attributes are integers uniformly distributed in $[0, 2^{32})$. Besides, we generate four datasets for evaluating skewed workloads, including three with Zipf distributions and one with normal distribution. The attributes of the four datasets are integers distributed in $[0, 2^{32})$. The probability density for the Zipf distribution is $p(k) = \frac{k^{-\alpha}}{\zeta(\alpha)}$ for integers $k \geq 1$, where ζ is the Riemann Zeta function, and α is the distribution parameter. In the evaluation, the parameters of the Zipf distributions are set as $\alpha = 1.1/1.3/1.5$, and the parameters for the normal distribution are $\mu = 2^{31}$ and $\sigma = 1$. The NumPy package is used to generate the datasets. Specifically, *numpy.random.zipf(α, size=(m, n))* is used to generate datasets with Zipf distribution, where α is the skewness, and size=(m, n) indicates the generation of m data records and n attributes.

Ray Tracing Configurations Through experiments on RTX 3090, we find that, for a uniform dataset with 1×10^8 records, the approximate optimal configuration for the ray length is 1.1×10^6, while that for the ray interval and ray space is 8.3×10^4. To ensure an intersection, the cube width equals the ray interval plus ϵ. ϵ is set for a dataset according to the formula in Section 4.4.4. We adopt the above configurations for RTScan in the evaluation.

Baselines We compare RTScan against RTIndex, BinDex on CPU, BinDex on CUDA, and GPU B-Tree [1]. RTIndex[3] is the state-of-the-art scan implementation on RT cores. Since the performance of RTIndex is orders of magnitude lower than that of CPU-based implementations, we use BinDex for performance comparison as it is the state-of-the-art scan approach on the CPU. To effectively compare the scan performance on RT and CUDA cores, we port the BinDex implementation to CUDA cores, referred to as BinDex-CUDA. BinDex-CUDA uses CUDA cores to accelerate the scan process in BinDex, including copying filter vectors and refining results, which achieves notable performance improvement. Both the BinDex implementations and RTScan use 128 bit vectors for a fair comparison. We also take an open-source GPU B-Tree[4] for performance comparison on CUDA cores.

In evaluating conjunctive predicates, approaches like BinDex, RTIndex, RTc1, and GPU B-Tree perform scans on the columns separately and then merge their results to get the final result. There are studies optimize the order of conjunctive predicates to improve the performance, where the predicate with the lowest selectivity is evaluated first to minimize the amount of accessed data. As the state-of-the-art implementation, BinDex's performance is not sensitive to the predicate selectivity, thus it does not need to reorder the predicates to optimize its performance. Therefore, the BinDex implementations on CPU and CUDA cores represent the state-of-the-art performance on the hardware.

[3]https://gitlab.rlp.net/juhenneb/rtindex (last accessed 2024/2/18)
[4]https://github.com/owensgroup/MVGpuBTree (last accessed 2024/2/18)

5.2 Performance Improvement of RTScan

A performance comparison between RTIndex, RTc1, RTc3, and RTScan on a uniform dataset is shown in Figure 15. The performance of RTScan is up to 5.4×10^5 higher than that of RTIndex and up to 106× higher than RTc1. Compared with RTc3, the naive implementation of evaluating conjunctive predicates in a three-dimensional space, RTScan also has up to 65× performance improvements. We can see that the performance of RTScan stays relatively stable when the selectivity varies, with a scan time ranging from 1.05 ms to 2 ms. This is because, even with high selectivity, RTScan only needs to scan a small volume of data records with Data Sieving. The difference in the execution time of RTScan comes from the selection of sieving bit vectors, where a query that needs fewer bits to refine is faster. The time cost of other scan methods increases when the selectivity increases. Thus, RTScan can achieve higher performance improvement with a high selectivity. Performance comparison on skewed datasets is shown in Figure 16, where a query with a selectivity of 0.9 is evaluated. RTScan has a 1.2×10^5 -2×10^5 times improvement over RTIndex and a 102 -1.7×10^5 times improvement over RTc1. Compared with RTc3, RTScan has a 40 -5.4×10^4 times improvement. Taking $\alpha = 1.3$ as an example, only one ray hits cubes in RTc3, which has to handle 9.8×10^7 intersection tests. However, in RTScan, 8.9×10^4 rays hit cubes, and the number of intersection tests is 1.3×10^5. Therefore, the average number of cubes intersected by each active ray is only 1.5. This brings RTScan a huge advantage over RTc3. Besides, the more skewed the dataset is, the higher the improvement RTScan can achieve over the competitors, which shows the effectiveness of the proposed techniques, i.e., Uniform Encoding.

The proportion of time spent per stage in RTScan is shown in Figure 17. The predicates used in the evaluation have the same selectivity of 0.9. Data Sieving and the merge of sieving bit vectors account for 24.0% and 14.5%, respectively. The main overhead of Data Sieving comes from the copying of selected bit vectors, which is performed by CUDA cores. Query encoding only accounts for 1.2%, which is fast to perform the mapping. Matrix RT Refine has the highest proportion and accounts for 60.4% of the total time on average. With a smaller selectivity, Matrix RT Refine would take a smaller proportion for fewer intersections while the other stages stay relatively stable.

5.3 Performance Comparison with CPU and CUDA Cores

To compare the performance of scan on CPU, CUDA cores, and RT cores, we compare the performance of RTScan against BinDex-CPU, BinDex-CUDA, and GPU B-Tree for a comprehensive evaluation in Figure 19. For a CPU-based database, the scan results may need to be transferred to the host memory after GPU acceleration. Therefore, we include the data transfer time in the evaluation, which is marked as the slash part of each bar. Instead, since BinDex-CPU

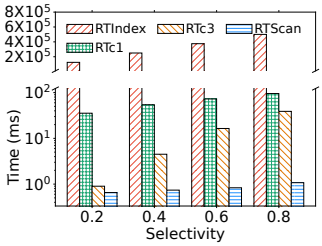

Figure 15: Performance improvement of RTScan on a uniform dataset

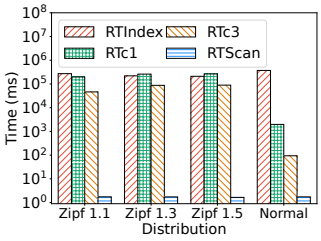

Figure 16: Performance improvement of RTScan on a skewed dataset

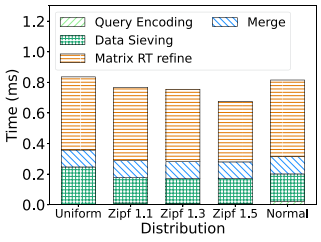

Figure 17: The proportion of different parts' time of RTScan

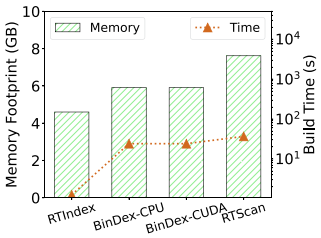

Figure 18: Index construction overhead

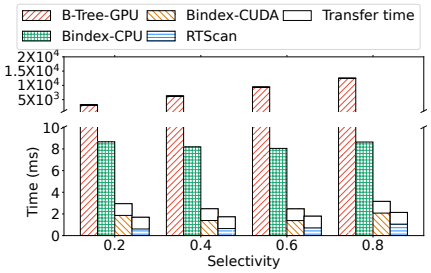

Figure 19: Performance comparison on uniform dataset

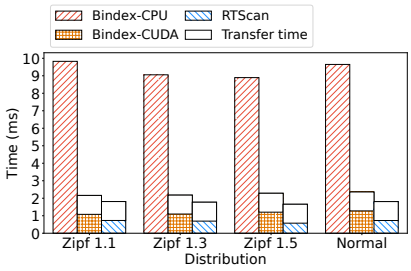

Figure 20: Performance comparison on skewed dataset

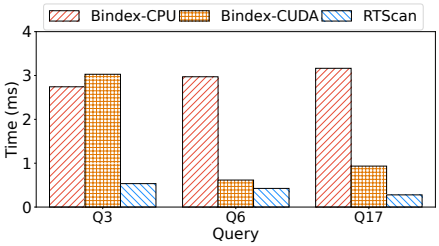

Figure 21: Performance of RTScan with TPC-H dataset

runs on the CPU, it does not need to transfer data. As shown in the figure, RTScan achieves an average of 3.7× performance improvement over BinDex-CPU, up to 1.3×10^4 times improvement over GPU B-Tree, and an average of 48% improvement over BinDex-CUDA. For a GPU database with no need to transfer the results back, RTScan has an average of 17.2× performance advantage over BinDex-CPU, up to 7.4×10^4 times over B-Tree, and an average of 90% over BinDex-CUDA. Although BinDex-CUDA also demonstrates much higher performance than the CPU, its performance advantage mainly comes from the 10496 CUDA cores. Instead, with only 82 RT cores, RTScan can also be 90% faster than BinDex-CUDA, which demonstrates the efficiency of RT cores for scans.

In Figure 20, we evaluate the performance of RTScan on four datasets with skewed distributions. For highly skewed datasets like Zipf with $\alpha = 1.5$, most of the data tend to be some specific values, and we use queries with a selectivity of 0.9 in the evaluation. GPU B-Tree is not evaluated for skewed distributions because it only supports scanning datasets with unique keys. As shown in the figure, when the results are transferred to the host memory, RTScan has a 4.2−4.6× improvement over BinDex-CPU and a 23.5%-37.9% improvement over BinDex-CUDA. In a GPU database without data transferring, RTScan has a 11.3× performance advantage over BinDex-CPU and a 69.7% improvement over BinDex-CUDA on average. As stated in section 5.2, with Uniform Encoding, RTScan can achieve higher improvements for highly skewed datasets. Specifically, for Zipf distribution with $\alpha = 1.5$, RTScan achieves up to 11.8× and 90.3% performance improvements over BinDex-CPU and BinDex-CUDA, respectively. These results demonstrate that, with the proposed techniques in RTScan, RT cores can achieve a much higher efficiency in handling skewed workloads than CPU and CUDA cores.

5.4 Data and Query Transformation Overhead

In Figure 18, we compare the memory consumption and the index construction time between RTScan, BinDex-CPU, BinDex-CUDA, and RTIndex. The dataset is a table containing 1×10^8 data records with three attributes, which follows a Zipf distribution with $\alpha = 1.5$. For the specified dataset, RTScan takes 36.8 seconds to construct the index, which takes 52.3% more time than the BinDex implementations and 28.1× more time than RTIndex. In RTScan, building the mapping table for Uniform Encoding and the sieving bit vectors take 48.2% and 49.5% of the overall index construction time, respectively. The data transformation time, i.e., building the BVH tree with customized primitives, accounts for 2.3% of the overall time. In comparison, RTIndex only needs to build the BVH tree, whose time is similar to that in RTScan.

For the dataset, the index of RTScan consumes 7.6 GB of memory space, which is 28.8% more than BinDex-CPU, 28.8% more than BinDex-CUDA, and 65.6% more than RTIndex. The memory allocation in RTScan mainly includes the mapping table for Uniform Encoding, the sieving bit vectors, and the BVH tree. The mapping table, which is stored in the host memory, accounts for 1.9% of the memory space. The sieving bit vectors and the BVH tree are stored in the device memory, occupying 58.6% and 39.5% of the memory, respectively. Reducing the sieving bit vectors in RTScan can reduce its memory usage while preserving its performance advantage. For instance, when the number of bit vectors is reduced from 128 to 64, RTScan can achieve an average of 33.7% higher performance than

BinDex-CUDA with 128 vectors, while the memory consumption is 9.8% lower than that of BinDex-CUDA.

In RTScan, conjunctive predicates need to be transformed into rays in the three-dimensional space. The query transformation consists of two main parts: identifying the regions to refine after Data Sieving and setting up the parameters of rays in these regions, including ray length, origin, and direction. For a range predicate, the query transformation time only takes an average of 0.8% of the overall query execution time. Even for a point query (x = a), which touches only a small amount of data, the query transformation in RTScan takes 1.1% of the query time on average.

5.5 Performance Improvement for TPC-H

We evaluate the performance of RTScan with TPC-H 3.0.1[5]. Since RTScan primarily focuses on scanning three columns, we select Q3, Q6, and Q17 from the TPC-H queries that fit RTScan's intended scenario. The predicates in the queries are as follows, which contain '<', '>', '=', and BETWEEN operators. (1) Q3: l_shipdate > "1995-03-15", o_orderdate < "1995-03-15", c_mktsegment = "BUILDING"; (2) Q6: l_shipdate between "1994-01-01" and "1995-01-01", l_discount between 0.05 and 0.07, l_quantity < 24; (3) Q17: p_brand = "Brand#23", p_container = "MED BOX", l_quantity < 6. We evaluate the average evaluation time for each query. As shown in Figure 21, RTScan has a $4.1-10.4\times$ improvement over BinDex-CPU and $44.8\%-4.7\times$ over BinDex-CUDA. Specifically, RTScan has a maximum of $10.4\times$ improvement over BinDex-CPU in Q17 and $4.7\times$ over BinDex-CUDA for predicates in Q3. For predicates in Q6, RTScan only achieves minor improvement over Bindex-CUDA. We analyze the reason and find that RTScan and BinDex-CUDA only have to refine an extremely small amount of data in Q6, which is about 3×10^3 data records for RTScan and 1.5×10^4 for BinDex-CUDA. In other queries, the amount of data to scan is larger; thus, RTScan has a higher improvement. For example, in Q3, 1.5×10^4 data records are intersected in RTScan, while 6.1×10^4 values are accessed for scanning in BinDex-CUDA. On average, RTScan has a $6.8\times$ performance improvement over BinDex-CPU and $2.5\times$ improvement over BinDex-CUDA, which proves the effectiveness of RTScan for workloads in production systems.

5.6 Performance of Other Number of Predicates

Figure 22, Figure 23 and Figure 24 show the performance of BinDex-CPU, BinDex-CUDA, and RTScan on one, two, and four predicates under a uniform dataset with 1×10^8 records. The execution time includes the data transfer between host and GPU memory. When evaluating one predicate, RTScan achieves 9.7% and 22.0% higher performance than BinDex-CPU and BinDex-CUDA, respectively. For two predicates, RTScan is 32.0% faster than BinDex-CUDA and $2.1\times$ faster than BinDex-CPU. This demonstrates that RTScan's performance is still competitive with less than three predicates.

RTScan divides conjunctive predicates into predicate groups, where the last group may contain one or two predicates. We take four conjunctive predicates as an example, which have to be divided into two predicate groups. There are two different combinations: either a combination of three columns and one column (3+1) or each group contains two columns (2+2). We implemented both approaches and evaluated their performance in Figure 24. As shown in the figure, RTScan(2+2) and RTScan(3+1) achieve similar performance for different selectivities, where RTScan(2+2) is only 3.2% faster than RTScan(3+1) on average. Compared with other processors, RTScan(2+2) demonstrates $5\times$ and 40.0% higher performance than BinDex-CPU and BinDex-CUDA, respectively. Because a query with any number of conjunctive predicates can be divided and processed by combinations with one, two, or three columns, RTScan can outperform other approaches for all cases.

5.7 Performance Impact of Encoding

The performance comparison of RTScan with and without Uniform Encoding on skewed datasets is shown in Figure 25. RTScan with Uniform Encoding (w/ EC) delivers $44.6\times$, $1152\times$, $16015\times$, and $1.3\times$ speedup than RTScan without encoding (w/o EC) for the four skewed datasets, respectively. Under $\alpha = 1.1/1.3/1.5$, the performance of RTScan without encoding is abysmal. For the skewed data distributions, most primitives cluster in some specific locations. As a result, among the rays cast uniformly into space, only a few can intersect cubes, while other rays would only incur BVH traversal overheads. However, there are tens of thousands of cubes for these hit rays to scan, leading to a significant decrease in parallelism. Taking $\alpha = 1.3$ as an example, only 6 rays hit the cubes, but the total number of intersection tests is 1.3×10^5, which means each ray needs to handle 2.1×10^4 intersection tests on average. Since certain parts of traversals and intersections are handled serially, there lacks sufficient parallelism to utilize the RT cores, resulting in poor performance. However, when Uniform Encoding is used to scatter the skewed data into the space, 8.9×10^4 rays hit cubes for the Zipf distribution with $\alpha = 1.3$, and the average number of cubes intersected by a ray is only 1.48. Therefore, Uniform Encoding balances the workload of rays and improves the parallelism.

We also evaluate the effectiveness of Uniform Encoding on attributes with small data ranges. The dataset has 1×10^8 data records, and the data range of each attribute is from 0 to 2^6. To prevent the result from being directly obtained through Data Sieving for the small range, we use 16 sieving bit vectors in this evaluation. We compare the performance with the one after encoding, where all columns have a data range from 0 to 1×10^8. As shown in Figure 26, RTScan has an average of $1.2\times$ performance enhancement over the one without encoding. The space size formed by the 2^6 data range is 2^{18} ($(2^6)^3$ in the three-dimensional space), smaller than the number of data records, i.e., 1×10^8. Consequently, around 381 (calculated as $1 \times 10^8/2^{18}$) cubes congregate at each coordinate in the space. This limits the parallelism and makes it hard to launch more rays to improve performance. With a selectivity of 0.8, 2.4×10^5 rays are launched for the 2^6 data range, and there are a total of 3.1×10^6 intersection tests and more than 13 intersection tests for each ray. After encoding, 3.8×10^6 rays are launched. With a total of 3.3×10^6 intersection tests, each ray only makes an average of 0.9 intersection tests. Therefore, with more rays being launched in a larger space, Uniform Encoding achieves higher parallelism and greatly lightens the burden of RT cores.

[5]https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf (last accessed 2024/2/18)

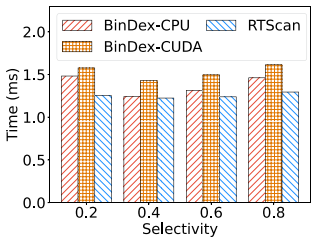

Figure 22: Scan on one column

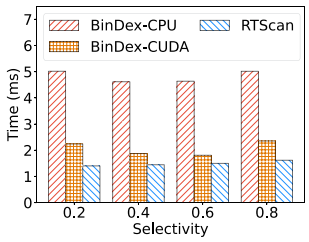

Figure 23: Scan on two columns

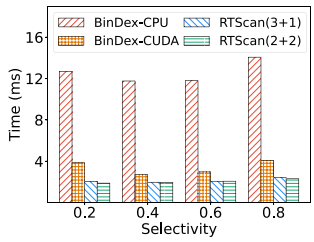

Figure 24: Scan on four columns

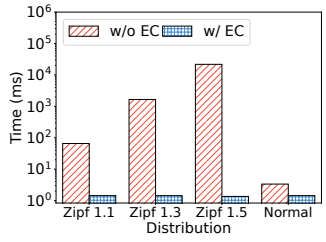

Figure 25: Effect of encoding on skewed datasets

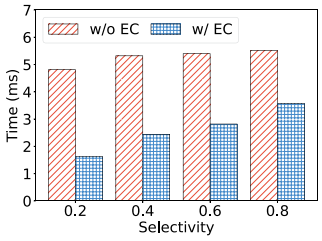

Figure 26: Performance impact of encoding on a dataset with small data ranges

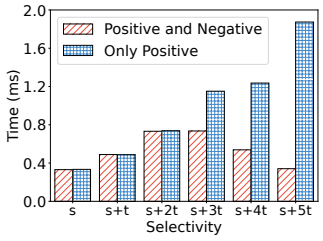

Figure 27: Positive and negative selection (s = 93.0%, t = 0.14%)

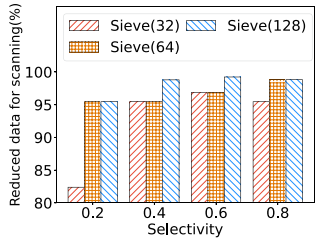

Figure 28: The reduced data for scanning with different number of bit vectors

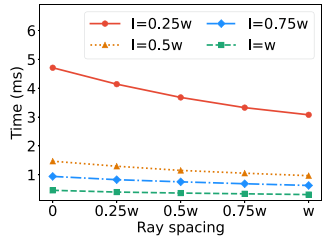

Figure 29: Scan performance by varying ray interval (I) and ray spacing (w is cube width)

5.8 Evaluation of Positive and Negative Selection

As described in Section 4.3.3 , for a query $x < C$ ($X_i < C < X_{i+1}$), both $F_i : x < X_i$ (Positive Selection) and $F_{i+1} : x < X_{i+1}$ (Negative Selection) can be selected as the approximate results. RTScan optimizes the performance by choosing the vector that requires refining a smaller amount of data. Figure 27 shows a performance comparison between the way using only Positive Selection and the adaptive way in RTScan that chooses the optimal vector. We take two sieving bit vectors F_{110} and F_{111} from the first column of data as an example, where F_{110} covers 93% of the entire dataset and F_{111} covers 93.7% on a dataset with uniform distribution. Predicate C is in the range of $[X_{110}, X_{111})$, and we divide the interval [93%, 93.7%] into five parts with an interval t = 0.14% (s = 93%). As shown in Figure 27, in the first half of the range, i.e., $[s, s + 2t]$, where query $C < (X_{111} - X_{110})/2$, both methods choose Positive Selection and use F_{110} as the vector for refining. Therefore, the scan time increases with the data that needs refining, i.e., in the range $[X_{110}, C - X_{110})$. In the second half of the range, i.e., $[s + 3t, s + 5t]$, where query $C > (X_{111} - X_{110})/2$, RTScan chooses Negative Selection and uses F_{111} as the vector for refining. In this case, the amount of data that needs refining is the range of $[X_{111} - C, X_{111}]$, which decreases as C increases. In contrast, Positive Selection keeps using F_{110} as the vector for refining in the second half of the range, resulting in an increasing amount of data that needs refining (i.e., $C - X_{110}$) and an increasing scan time as C increases. When selectivity comes to s+5t (93.7%), RTScan has a 1.8× improvement over the one that only uses Positive Selection. This figure takes F_{110} and F_{111} as examples of the effect, while the selectivities between other adjacent vectors also demonstrate the same pattern.

5.9 Reduced Data for Scanning by Different Numbers of Vectors

The number of sieving bit vectors in RTScan is the main factor influencing the performance since it determines the number of data records to refine. Figure 28 demonstrates the portion of reduced data for scanning in RTScan with different numbers of bit vectors on a uniform dataset with 10^8 data records. Sieve(K) denotes the Data Sieving with K bit vectors, and reduced data for scanning represents the proportion of data records avoided to scan with Data Sieving. As demonstrated in the figure, because different numbers of sieving bit vectors have different partitions on the data range, the reduced amount fluctuates across different selectivities. For instance, for N data records with 32 sieving bit vectors, the ith vectors represent $3.125\% \cdot i \cdot N$ data records. When F_6 is chosen for a predicate with 20% selectivity, it filters $18.75\%N$ values in the column. The total portion of data records that are avoided for scanning reaches $\frac{(18.75)^3}{(20)^3} = 82.4\%$ with three sieving bit vectors applied for the conjunctive predicates. On the other hand, F_{13} will be chosen for predicates with 40% selectivity, and the reduced number of data records for scanning rises to $\frac{(40)^3}{(40.625)^3} = 95.5\%$.

The reduced amount increases as the number of sieving bit vectors increases. This is because the data range is partitioned in a much finer way with more vectors, leading to fewer data records to be refined with a selected vector. For instance, when the number of sieving bit vectors increases to 64, adjacent vectors only have around $1.56\%N$ different represented data records. The F_{13} is chosen with a predicate of 20% selectivity, leading to a 95.5% reduction of the scanned data record, which is 13.1% higher than that of Sieve(32) (82.4%). Because more bit vectors lead to more

memory consumption, users can choose fewer vectors with a minor influence on the performance.

5.10 Performance Impact of Ray Interval and Ray Spacing

Figure 29 presents a comparative analysis of scan performance while varying the ray interval (I) and ray spacing (S). The value of ray interval and ray spacing in the figure is its proportion to the cube width (w), where $I = 0.5w$ means the ray interval equals half of the cube width. With a fixed ray interval, when the ray spacing increases from 0 to w, the performance of RTScan improves. This is because an increased ray spacing leads to a lower BVH traversal overhead and fewer intersection tests. Specifically, when the ray interval is equal to the cube width ($I = w$), the performance of $S = w$ has a 47% performance improvement over that of $S = 0$, and for $S = w$, there are 2.56×10^5 rays and 2.29×10^5 intersection tests, whereas for $S = 0$, there are 2.76×10^5 rays and 2.45×10^5 intersection tests. With a fixed ray spacing, RTScan delivers improved performance as the ray interval increases. This is because an increased ray interval means fewer rays to be launched, which leads to a lower BVH traversal overhead. Moreover, with the same cube width, fewer rays lead to fewer intersections. Specifically, when the ray spacing is equal to the cube width w, RTScan with $I = w$ outperforms that with $I = 0.25w$ by 70%. There are 3.84×10^6 rays and 3.44×10^6 intersection tests for $I = 0.25w$, and that for $I = w$ is described above.

6 RELATED WORK

Accelerating scan operations Scan operations generally include sequential scans and index scans. Various techniques have been investigated to accelerate sequential scans, such as compression [2, 27], and scan sharing [6, 21, 22]. Traditional secondary indexes, such as bitmaps, B-trees, and their variations, are commonly employed for categorical data with low cardinality or numerical data with a low selectivity[24, 30]. For a predicate with a high selectivity, lightweight indexes [16, 28, 29] adopt statistic information to skip data in sequential scan, but they can be less effective when the data are uniformly distributed. The early pruning techniques, including Bitweaving [14] and ByteSlice [4], and Column Sketches [9], use either bit-level storage or data compression to reduce the amount of touched memory in sequential scan. BinDex [13] is an index that uses bitmaps as the filters to avoid accessing most data for a predicate. It outperforms existing approaches for predicates with all selectivities.

Conjunctive predicates Several commercial systems, including Vectorwise [26, 33], a prominent column store designed for analytical workloads, prioritize predicates based on ascending selectivity without consideration of their costs. There are some other optimization techniques for conjunctive predicates, with a focus on determining the optimal evaluation order of selection predicates with cost models or selectivity estimation [7, 12, 19, 23]. In addition, Wang et al. [31] propose an order-oblivious execution scheme to optimize the evaluation of conjunctive predicates. Note that, because RTScan evaluates three predicates at the same time, it achieves high performance for a predicate group without considering their evaluation orders.

7 DISCUSSION

The performance advantage of RTScan: The performance advantage of RTScan comes from two aspects. 1) As discussed in Section 3.1 for RTc3, RTScan evaluates three conjunctive predicates in one RT job and only needs to scan $S_1 \cdot S_2 \cdot S_3 \cdot N$ data records, where S_k denotes the selectivity of the k-th predicate. With Data Sieving, the amount of touched data is further reduced. 2) RTScan efficiently transforms data scans as an RT job and leverages RT cores for hardware acceleration. RTScan can benefit from the fast GPU advancement. While the NVIDIA RTX 3090 has 82 RT cores, the RTX 4090 GPU has 128 third-generation RT cores with two to three times performance improvement.

The integration of RTScan: The integration of RTScan in a database system depends on the underlying execution model. For a database system with a materialization model, like MonetDB and VoltDB, RTScan can be integrated by using it as the selection operator. Most GPU databases adopt a materialization model and can directly adopt RTScan as the selection operator. Besides, RTScan can further improve the performance and GPU hardware utilization by utilizing both CUDA and RT cores for query processing.

When data is updated, the BVH tree should be rebuilt, while the mapping table and the sieving bit vectors need to be updated. As shown in Section 5.4, it takes a non-trivial cost to build the index of RTScan. Therefore, RTScan is currently only fit for handling workloads with infrequent updates.

Handling disjunctive predicates: The paper only discusses conjunctive predicates, while the evaluation of disjunctive predicates needs further studies. There are two ways to support disjunctive predicates. One way is transforming disjunctive predicates into conjunctive predicates [5]. For example, predicates $(A \wedge B \vee C)$ can be transformed into $!(!(A \wedge B) \wedge !C)$. However, this scheme cannot leverage the advantage of RT cores because only $(A \wedge B)$ can be evaluated on RT cores, while other operations like *bitwise NOT* and the evaluation of C has to be performed on CUDA cores. The second way is to identify the involved query areas in the three-dimensional space and launch rays to intersect the primitives that satisfy the predicates. For predicates like $(A \wedge B \vee C)$, there would be several separate query areas scattered in the space, especially after Data Sieving is applied. It requires a more complex query transformation mechanism to cover all areas of the predicates. Moreover, it may need to reevaluate the efficiency of RT cores for such predicates. We take it as a future work.

8 CONCLUSION

Through intensive evaluation and design space exploration, we analyze and identify the performance bottlenecks when utilizing RT cores to accelerate scans. Based on the analyses of workload characteristics and their impact on performance, we propose RTScan, a scan approach that efficiently maps the evaluation of conjunctive predicates into a ray tracing job. RTScan is designed with three key techniques, i.e., Uniform Encoding, Data Sieving, and Matrix RT Refine. These techniques significantly accelerate RTScan on various workloads by enhancing the parallelism and alleviating and balancing the ray load. RTScan enhances the scan performance on RT cores by five orders of magnitude and outperforms state-of-the-art implementations on CPUs and CUDA cores.

REFERENCES

[1] Muhammad A Awad, Serban D Porumbescu, and John D Owens. 2022. A GPU Multiversion B-Tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 481–493.

[2] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 283–296.

[3] I Evangelou, G Papaioannou, K Vardis, and AA Vasilakis. 2021. Fast radius search exploiting ray-tracing frameworks. *Journal of Computer Graphics Techniques Vol* 10, 1 (2021).

[4] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*. ACM, 31–46.

[5] Vijay K Garg. 2002. *Elements of distributed computing*. John Wiley & Sons, 150–151.

[6] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared workload optimization. *Proceedings of the VLDB Endowment* 7, 6 (2014), 429–440.

[7] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate migration: optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93*.

[8] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *arXiv preprint arXiv:2303.01139* (2023).

[9] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*. 857–872.

[10] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-Wise Parallel Predicate Evaluation. *Proc. VLDB Endow.* 1, 1 (aug 2008), 622–634.

[11] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. Warpcore: A library for fast hash tables on gpus. In *2020 IEEE 27th international conference on high performance computing, data, and analytics (HiPC)*. IEEE, 11–20.

[12] Fisnik Kastrati and Guido Moerkotte. 2016. Optimization of conjunctive predicates for main memory column stores. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1125–1136.

[13] 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*. 909–923.

[14] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300.

[15] Enzo Meneses, Cristóbal A. Navarro, Héctor Ferrada, and Felipe A. Quezada. 2023. Accelerating Range Minimum Queries with Ray Tracing Cores. arXiv:2306.03282 [cs.DC]

[16] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proc. VLDB Endow.* 476–487.

[17] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. arXiv:2303.09655 [cs.DC]

[18] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-KNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 289–300.

[19] Thomas Neumann, Sven Helmer, and Guido Moerkotte. 2005. On the optimal ordering of maps and selections under factorization. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 490–501.

[20] NVIDIA. 2018. NVIDIA Turing GPU architecture. (2018), 25–29, 30–32. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[21] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing Data and Work across Concurrent Analytical Queries. *Proc. VLDB Endow.* 6, 9 (2013), 637–648.

[22] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J Haas, and Guy M Lohman. 2008. Main-memory scan sharing for multi-core CPUs. *Proceedings of the VLDB Endowment* 1, 1 (2008), 610–621.

[23] Kenneth A Ross. 2002. Conjunctive selection conditions in main memory. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 109–120.

[24] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[25] Peter Shirley, Ingo Wald, Tomas Akenine-Möller, and Eric Haines. 2019. *What is a Ray?* Apress, Berkeley, CA, 15–19. https://doi.org/10.1007/978-1-4842-4427-2_2

[26] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[27] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[28] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1115–1126.

[29] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[30] 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.

[31] Zeke Wang, Xue Liu, Kai Zhang, Haihang Zhou, and Bingsheng He. 2019. Understanding and Optimizing Conjunctive Predicates Under Memory-Efficient Storage Layouts. *IEEE Transactions on Knowledge and Data Engineering* 33, 6 (2019), 2803–2817.

[32] Yuhao Zhu. 2022. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 76–89.

[33] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.