RayDB: Building Databases with Ray Tracing Cores

Xuri Shi
Fudan University
xrshi23@m.fudan.edu.cn

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

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

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

Rubao Lee
Freelance Researcher
lee.rubao@ieee.org

ABSTRACT

Ray tracing (RT) cores are a new type of hardware that have been actively integrated into modern GPUs. Recent studies have demonstrated that RT cores can deliver much higher performance than GPU CUDA cores and CPUs for general-purpose data processing tasks like table scan. In this paper, we propose RayDB, the first query engine that leverages RT cores to accelerate database query processing. RayDB transforms the core part of the query execution that involves multiple operators into one single ray tracing job. With a set of proposed encoding and ray launching techniques, RayDB effectively exploits RT cores to accelerate diverse workloads. Experimental results show that RayDB outperforms the state-of-the-art GPU-based query engine by up to 18.3×.

PVLDB Reference Format:
Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, and Rubao Lee. RayDB: Building Databases with Ray Tracing Cores. PVLDB, 19(1): 43 - 55, 2025.
doi:10.14778/3772181.3772185

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LonelySlim/myOptixDB/tree/fin.

1 INTRODUCTION

With the ever-increasing data volume from applications, modern processors have been intensively studied to enhance the performance of database engines. Representative processors for which database systems have been designed include GPU CUDA cores [4, 13, 32], Tensor cores [12, 15], and FPGAs [26, 33]. Specifically, GPUs were originally designed to accelerate computer graphics. After they were found useful in general-purpose computing for the massive number of cores, the programming model evolved from OpenGL to CUDA/OpenCL, and GPUs are used to accelerate a broad class of data processing tasks. Crystal [32] is the state-of-the-art implementation of GPU databases, which has made a notable advancement in utilizing CUDA cores. Experimental results show that Crystal is 16× faster than the GPU-based HeavyDB and 61× faster than the CPU-based MonetDB. However, Crystal saturates more than 97% of the GPU memory bandwidth for queries in the Star Schema Benchmark (SSB). Since the approach has already tried to minimize the amount of data accessed, it has become difficult to further improve the query performance on CUDA cores.

Recently, commodity GPUs have incorporated ray tracing (RT) cores to boost the real-time rendering of 3D scenes. As an important type of computing resource, RT cores have been adopted in mobile, desktop, and workstation processors, which are under fast development. RT cores efficiently trace rays through a 3D space to identify intersected objects. With user-defined functions (a.k.a., shaders), RT cores can perform customizable operations upon ray-object intersections, providing versatility for various tasks. RT cores have been used to accelerate data processing tasks such as K-nearest neighbor search [25, 38], scan operator [14, 21], and range minimum queries [22]. Specifically, existing work like RTScan [21] has shown that RT cores can bring up to 4.6× higher performance than CUDA cores and CPU for database operators. Therefore, we believe that RT cores have the potential to become another critical computing resource for general-purpose data processing tasks.

Unlike database implementations on CUDA cores, accelerating a data processing program with RT cores requires the program to be transformed into an efficient RT job. In an RT job, data records are transformed into primitives, such as triangles or spheres, positioned in a three-dimensional space with a bounding volume hierarchy (BVH), while a query is converted into rays in a specified region. If the task does not fit such a job transformation, or the mapping is inefficient (e.g., lack of parallelism with a limited number of rays), it may result in even lower performance than CPUs and CUDA cores [14, 21]. Due to the above reason, exploring RT cores to accelerate database queries is quite challenging because an operator like `Join` or `GroupBy` is hard to transform into an independent RT job. Moreover, since the execution of an operator depends on the output of its previous operator in the query plan, the BVHs for the following operators have to be built during query execution, which is very time-consuming. Therefore, simply implementing independent RT-based operators like CUDA-based databases cannot exploit the performance advantage of RT cores.

In this paper, we propose RayDB, a query engine that utilizes ray tracing cores to achieve unprecedented performance. Instead of implementing an RT job for each operator, the main idea of RayDB is to map the core query execution containing multiple operators into a single RT job. To be specific, RayDB maps and encodes the data attributes involved in the core operators, i.e., `Aggregation`, `GroupBy`, and `Scan`, to the coordinates x, y, z in the 3D space, respectively. RayDB pre-builds a set of BVHs for query execution to select from. When building a BVH, the attributes

*Dr. Kai Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772185

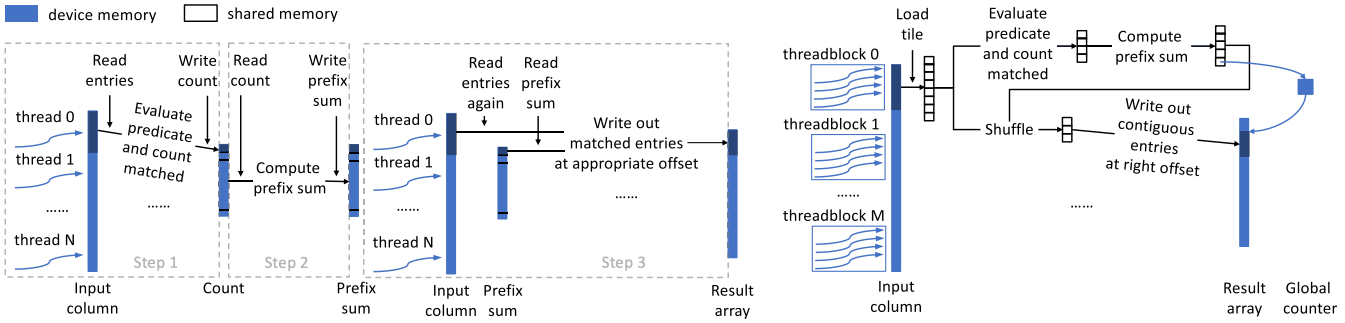


Figure 1: Running Scan on CUDA-based database systems

involved are extracted and encoded as the primitive coordinates of each data record. With a selected BVH for a query, rays in RayDB are launched in the region specified by `Scan` so that they intersect only with primitives that satisfy the predicates, significantly reducing the amount of data accessed. For each primitive, the data attributes involved in `Aggregation` and `GroupBy` are encoded in their 3D coordinates, which can be accessed directly in query processing. Therefore, another main advantage of RayDB is that it can retrieve all the attributes of a data record needed for the three operators with only one memory access, i.e., access to the coordinates of a primitive, dramatically reducing the number of random memory accesses. Moreover, the entire process is accelerated by ray tracing cores, which are designed to perform tasks like this efficiently. After RT processing, RayDB uses CUDA cores to execute other operators like `Having` and `OrderBy` if any. In addition, RayDB natively supports parallel execution on multiple GPUs by partitioning the 3D space into independent regions. It enables flexible scaling out to support workloads of any size.

The contributions of this paper are as follows.

- We propose RayDB, a query engine that leverages RT cores for acceleration by effectively transforming the core part of a query into a ray tracing job.
- We propose a set of encoding and ray-launching mechanisms to enable efficient query execution across diverse workloads.
- We implement the prototype of RayDB, evaluate its performance, and quantitatively analyze the advantages of RT cores.

Experimental results show that RayDB can significantly enhance query performance. Compared with the state-of-the-art CUDA-based method, RayDB improves the query performance by up to 18.3×. In addition to the substantial speedup, the usage of the memory bandwidth drops from 97.4% to only 36.7% on average, which proves that RayDB has broken the limitation of memory bandwidth in GPU-based query execution. To our knowledge, it is the first work that demonstrates that RT cores can be effectively used in building a database engine with unprecedented performance.

2 BACKGROUND AND MOTIVATION

2.1 An Analysis of GPU-based Databases

As a general-purpose computing device, GPU CUDA cores have been intensively studied to build high-performance query engines, such as Crystal [32], HeavyDB [13], BlazingSQL [4], etc. Among them, Crystal is the state-of-the-art. Figure 2 compares the query runtime of Crystal with HeavyDB and MonetDB [5] on SSB flat. The GPU used in the comparison is NVIDIA GeForce RTX 4090, and the CPU used is Intel Xeon Gold 5318Y. The results show that Crystal is, on average, 16× faster than HeavyDB and 61× faster than MonetDB.

The key to Crystal's high performance is that its *tile-based execution model* aims at efficiently utilizing the GPU shared memory, which has an order of magnitude higher bandwidth than the device memory (36618 GB/sec versus 1008 GB/sec in the RTX 4090). Taking the `Scan` operator as an example, the traditional CUDA-based database systems execute the operator in three steps, as shown in Figure 1a. First, multiple CUDA threads are launched to scan their assigned entries and count the matches, storing the results in the *count* array. Then, prefix sums are computed over the *count* array to produce the *prefix sum* array, which provides write offsets. Finally, the same number of threads are launched to read the allocated entries again and write matched entries to the *result* array using the offsets. There are three main performance issues with this approach, including 1) reading the input column twice from the device memory, 2) reading and writing intermediate structures like *count* array and *prefix sum* array in the device memory, 3) each thread writes to a different location in the *result* array, resulting in uncoalesced memory accesses.

Figure 1b demonstrates how Crystal works. Crystal uses a thread block as the basic execution unit, with each block processing a tile of entries. It first loads the tile from device memory into shared memory. Threads then evaluate the predicate in parallel and record match counts of each thread in the *count* array in shared memory. A prefix sum is computed over the *count* array, and a global counter is atomically updated to get the block's output offset. Matched entries are shuffled to form a contiguous array in shared memory, enabling coalesced writes to the *result* array in device memory. By leveraging shared memory, Crystal 1) avoids repeatedly reading input columns from the device memory, 2) eliminates accesses to intermediate structures in device memory, and 3) enables coalesced memory access when writing the *result* array back to the device memory, leading to significant performance improvements.

It is worth noting that existing CUDA-based database systems generally do not use traditional indexes. This design choice is driven

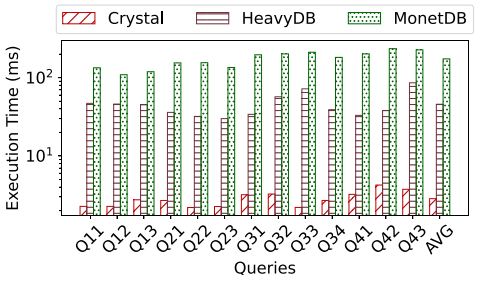


Figure 2: Comparison of query runtime for Crystal, MonetDB, and HeavyDB on SSB flat

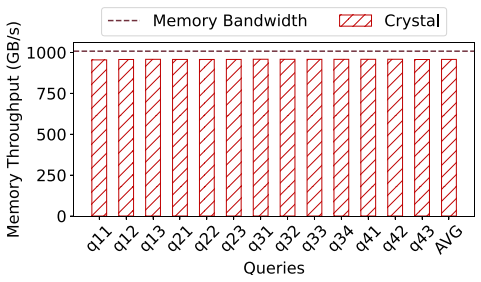


Figure 3: Query memory throughput of Crystal on SSB flat

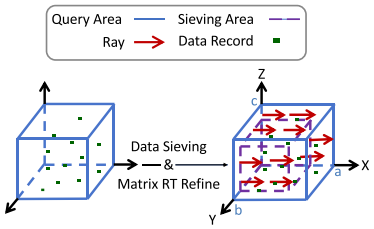


Figure 4: The execution of RTScan

by the architectural and performance characteristics of GPUs: 1) Avoid thread divergence. On GPUs, the SIMT (Single Instruction, Multiple Threads) architecture requires all threads in a warp (typically 32) to execute the same instruction each cycle. When threads diverge due to conditional branches (e.g., if), execution becomes serialized, and inactive threads idle, reducing efficiency [3, 6, 31]. Traditional index structures like B+ trees cause severe divergence because each thread follows a different search path based on its key. As a result, threads within a warp execute different code paths, making indexes poorly suited for efficient execution on GPUs. 2) Avoid uncoalesced memory accesses. To amortize memory access latency and improve bandwidth utilization, GPU hardware employs memory access coalescing, which combines memory accesses from multiple threads within a warp into one or a few large memory transactions when the accessed addresses are contiguous. However, in traditional index structures (such as B+ trees and hash indexes), data and index nodes are typically scattered across memory. Each thread accesses different paths and nodes depending on its query key, resulting in highly irregular and non-contiguous memory accesses across threads. In such scenarios, memory access coalescing fails, leading to a large number of small and fragmented memory transactions. This causes severe memory bandwidth underutilization and a significant drop in performance [6, 31]. As a result, full table scans are generally preferred over indexes in CUDA-based database systems.

Figure 3 shows the memory throughput of Crystal on the 13 queries of SSB flat. It can be observed that its memory throughput is very close (an average of 97.4%) to the GPU device memory bandwidth. As Crystal already makes highly efficient use of memory bandwidth, the potential for further optimization is minimal.

2.2 Background of Ray Tracing

Ray tracing is a rendering technique used in computer graphics to simulate the way rays interact with objects in a scene. It works by tracing the paths of rays as they travel through a 3D space. Objects in the three-dimensional space are represented as primitives, whose types include triangles, spheres, and even custom primitives. All primitives in the space are wrapped by bounding volumes, which are usually Axis-Aligned Bounding Boxes (AABBs). AABBs are then organized hierarchically as a tree known as the Bounding Volume Hierarchy (BVH). A ray tracing job utilizes a BVH to traverse the space and find intersected primitives with the rays.

Although BVH avoids a large number of potential ray-primitive intersection tests, the BVH traversal and intersection tests are still time-consuming. Since the Turing architecture [27], NVIDIA GPUs are equipped with dedicated hardware, i.e., RT cores, to speed up BVH traversal and intersection tests. Taking the classic Turing architecture as an example, each Streaming Multiprocessor (SM) integrates one RT core, which operates alongside CUDA cores and Tensor cores [27]. The RT core essentially adds a dedicated pipeline (ASIC) to the SM. It can access the BVH and configure some L0 buffers to reduce the delay of BVH and triangle data access. The request is made by SM. The instruction is issued, and the result is returned to the SM's local register. The interleaved instruction and other arithmetic or memory I/O instructions can be concurrent. Because it is an ASIC-specific circuit logic, $performance/mm^2$ can be increased by an order of magnitude compared to the use of shader code[1]. Besides the mainstream desktop and workstation GPUs like NVIDIAs RTX 40 series and AMDs RX 7000, data center GPUs like NVIDIA A40 and T4 also support ray tracing. Specifically, the NVIDIA RTX 4090 GPU integrates 128 RT cores. OptiX programming model [29] is an application framework for building ray tracing jobs. In OptiX, each ray is mapped to a CUDA thread. CUDA threads generate rays with the specified ray origins and directions. Then, the control is transferred to RT cores, which accelerate BVH traversal and ray-triangle intersection tests. For NVIDIA GPUs, triangles are the built-in primitives, which allows RT cores to accelerate intersection tests when the primitives are triangles. When other types of primitives are used, control is transferred to CUDA threads to perform intersection tests defined in the Intersection Shader. Overall, OptiX leverages both CUDA threads and RT cores to achieve efficient ray tracing jobs. RT cores specialize in accelerating BVH traversal/ray-triangle intersections, while CUDA threads provide flexibility for custom intersection logic through the Intersection Shader. This enables extensibility for complex ray tracing scenarios.

2.3 Expedite Data Processing with RT Cores

RT cores have been utilized to accelerate various data processing tasks like K-nearest neighbor search [25, 38] and range minimum

[1]https://www.reddit.com/r/nvidia/comments/97qogl/former_nvidia_gpu_architect_what_is_rt_core_in (last accessed 2025/10/6)

queries [22]. RTScan [21] and RTIndex [14] are pioneering implementations that leverage RT cores to accelerate the Scan operator. Specifically, RTScan [21] achieves significant performance improvement by mapping the evaluation of entire conjunctive predicates into a ray tracing process. Experiments show that RTScan achieves up to 4.6× higher performance than BinDex [20], which is the state-of-the-art scan approach on CPUs. Figure 4 demonstrates the approach of RTScan when evaluating a query with three predicates. For each data record, the three data attributes involved in the predicates are used as the coordinates of the corresponding primitive. Assuming that the conjunctive predicates are $(0 \leq x \leq a \wedge 0 \leq y \leq b \wedge 0 \leq z \leq c)$, then the query area is a cuboid with the origin as a vertex and three edges of length a, b, and c as shown in the figure. The data records satisfying the conjunctive predicates are all in the query area. To reduce the computation overheads on intersection tests, RTScan adopts Data Sieving, which uses pre-stored results to filter most data records (the dashed area). Then, RTScan launches rays in the remaining region to intersect primitives with enhanced parallelism. These techniques add up to form the performance improvement over CUDA cores and CPUs. In addition, RTScan adopts Uniform Encoding to evenly distribute data records in space, enabling it to achieve good performance even under skewed data distributions.

After analyzing RTScan and other RT-based implementations, we summarize three key aspects of efficiently mapping a data processing task to a high-performance RT job. 1) *Evaluating multiple operators in one job:* RTScan utilizes the 3D feature to evaluate three predicates simultaneously within one RT job, whose execution time is even lower than that of evaluating one predicate. 2) *Reducing the amount of data accesses:* The evaluation of conjunctive predicates and the Data Sieving technique dramatically shrink the querying region, leading to a significantly smaller number of primitives for intersections. 3) *Enhancing the parallelism:* RTScan segments a long ray into several small rays with spacing, which aims at maximizing the utilization of RT cores while balancing their load.

2.4 Challenges of Accelerating Query Processing with RT Cores

For GPU acceleration with CUDA cores, operators are generally implemented as separate CUDA kernels and executed sequentially. RTScan demonstrates a standalone implementation of accelerating Scan with RT cores. However, accelerating database queries with each operator implemented as a separate RT job faces several critical issues that are hard to address.

Difficulties in the three-dimensional mapping of operators: In order to improve performance, a database operator needs to be effectively converted into a ray tracing job, or the performance advantage of RT hardware cannot be effectively exploited. For instance, a naive scan implementation on RT cores can be 2.3×10^4 times slower than the state-of-the-art CPU-based implementation [21]. However, the natures of some database operators make it hard to map the data in the 3D space and hard to convert operations into ray intersections. Therefore, except Scan, other operators like Join and GroupBy have not demonstrated superior performance on RT cores so far.

Inefficiency of the execution of multiple operators: A ray tracing job uses a BVH as its index, but building a BVH is a time-consuming process that takes orders of magnitude longer time than the RT job itself. For a dataset with 120 million tuples, the average time to build a BVH is 227.84 ms, while the average time to launch rays is only 0.75 ms. When the operators of a query are executed sequentially on RT cores, the BVHs for the second to the last operator can only be built online, resulting in an ultra-high query processing latency.

```
A: SELECT AVG(Math)
   FROM Score
   WHERE English >= 60
   GROUP BY Hometown;
```

We take the query A as an example, which executes the operators in the following order: Scan → GroupBy → Aggregation. For Scan, the BVH it traverses can be pre-built from the English column directly. However, for GroupBy, the BVH it traversed is built from the Hometown column filtered by the execution results of Scan. Therefore, it can only start building the BVH after the execution of Scan is complete and the results are obtained. The above process is shown in Figure 5a. As a result, the BVH building process for GroupBy has to be taken as part of the query execution, which results in ultra-low performance. An alternative scheme is shown in Figure 5b, where the BVHs for all operators have been pre-built from their corresponding attributes. In this case, GroupBy cannot get the filtered results from Scan and has to group all data in the column. Moreover, the results of GroupBy have to be further filtered according to the results of Scan. Consequently, this scheme completely negates the performance benefit of RT cores and is severely inefficient.

To conclude, due to the aforementioned issues, accelerating database queries with RT cores to outperform CUDA cores and CPUs is particularly challenging.

3 THE DESIGN OF RAYDB

3.1 Overview of RayDB

We propose RayDB, a high-performance query engine accelerated by RT cores. Specifically, RayDB leverages RT cores to accelerate three core operatorsAggregation, GroupBy, and Scanwhile delegating the remaining parts of the query to CUDA cores. Instead of implementing each operator as a separate RT job, RayDB maps the execution of these three operators to one ray tracing job. The basic idea is to use the attributes involved in these three operators as the coordinates for a data record, i.e., the attributes in Aggregation, GroupBy, and Scan are used as the X-coordinate, Y-coordinate, and Z-coordinate, respectively.

The workflow of RayDB is illustrated in Figure 7. As denormalization is widely adopted in data warehouses and favored by many technology companies [1, 2, 11], RayDB is designed to accelerate data warehouse queries on denormalized tables. During the offline phase, it performs denormalization in initialization by joining all relevant tables into a single flat table, thereby eliminating the need for Join during query execution. Next, a set of BVHs is built offline based on different combinations of attributes in the wide table. For a given query, RayDB parses the attribute composition of its three core operators and selects a pre-built BVH from the BVH

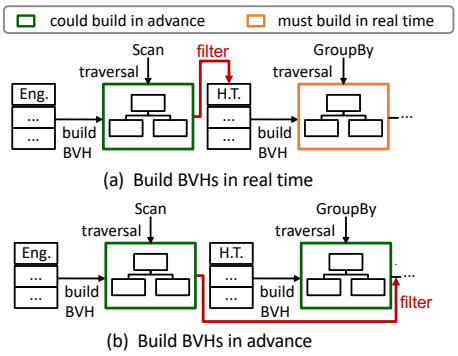


Figure 5: Two strategies of query execution with RT cores

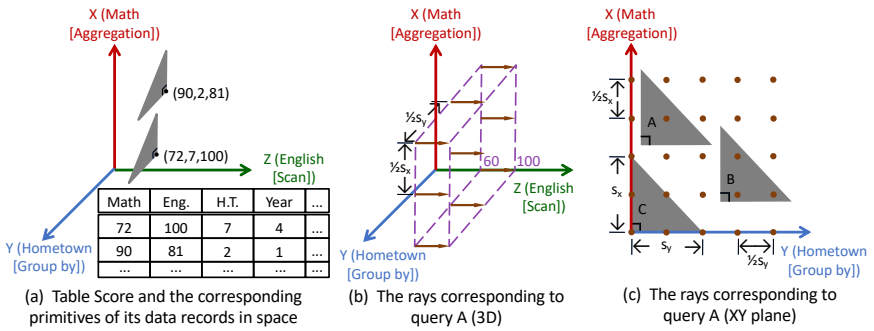


Figure 6: The design of RayDB

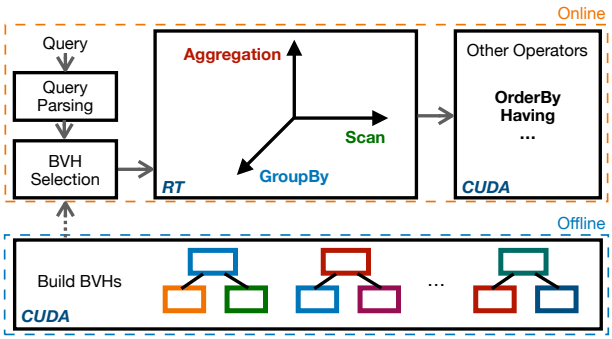


Figure 7: The workflow of RayDB

set that covers all the required attributes. It is worth noting that when a query involves excessive number of attributes such that no BVH can fully cover all of them, RayDB can utilize a BVH that includes only part of the required attributes to enable the query to still benefit from the acceleration provided by RT cores (introduced in Section 4.5). With the selected BVH, the RT cores are responsible for accelerating the execution of `Aggregation`, `GroupBy`, and `Scan`, which are fused into a single RT job. The design is based on the observation that these three operators appear consecutively in the query plan after denormalization. It is worth noting that this approach offers sufficient flexibility to remain effective even when certain core operators are absent from the query. For example, in queries without `GroupBy`, all records can be treated as belonging to the same group. RayDB determines the corresponding query area in the space based on `Scan` and launches a set of rays to intersect all primitives in the area. By only accessing data records in the query area defined by `Scan`, there is a significant reduction in the amount of data accessed for a query. Based on the coordinates of an intersected primitive, RayDB can obtain its data attributes for `GroupBy` and `Aggregation` and then perform the corresponding operations. For each data record, the data attributes involved in the three operators are stored together as coordinates, which can be retrieved by a single memory access. It dramatically reduces the number of random accesses to the device memory. Once the RT cores complete execution, the remaining operatorssuch as `OrderBy` and `Havingare` handled by CUDA cores in a manner similar to existing CUDA-based database systems.

Instead of building a BVH for every query, RayDB maintains a few BVHs to support a wide range of queries. The encoding scheme allows multiple attributes to be compressed into a single coordinate (introduced in Section 4), where a query may use one or an arbitrary combination of the encoded attributes, enabling each BVH to support more queries. Furthermore, based on our observation of real-world queries, we find that certain columns tend to be used with specific types of operators. For example, in TPC-H, `l_shipdate` is predominantly used in `Scan`, while `l_extendedprice` is typically used in `Aggregation`. By pre-constructing BVHs for semantically meaningful combinations of attributes, RayDB is able to cover common OLAP queries using a few BVHs. For queries with subqueries, RayDB can accelerate them if the optimizer rewrites them into equivalent queries without subqueries through techniques like unnesting or decorrelation. If such rewriting isnt possible, RayDB currently does not support their execution.

3.2 The Mechanism of RayDB: An Example

In this subsection, we demonstrate how a BVH is constructed and how an RT job is executed. Suppose there is a denormalized table named *Score*, as shown in Figure 6a, that stores information about students' scores. Each row of the table corresponds to a student, and the table has many attributes, among which the *Math*, *English*, *Hometown*, and *Year* are used to store students' math scores, English scores, hometowns, and academic years, respectively. The course scores are in the range $[0, 100]$ and are integers. For each column, there must exist a data range that can be known in advance (e.g., maintained as metadata). We use the execution of query A (Section 2.4) as an example. The query obtains the average math score of students in each hometown who have passed English.

Building the BVH: First, it should be clarified that the BVH constructed in this example is the most suitable one from the pre-built BVH set for executing Query A. The key step in building a BVH lies in mapping data records to primitives. Based on the idea of representing data attributes involved in `Aggregation`, `GroupBy`, and `Scan` by the coordinates in three-dimensional space, RayDB makes the X-axis, Y-axis, and Z-axis represent the data attributes involved in `Aggregation`, `GroupBy`, and `Scan`, respectively. In this way, each

data record in the table corresponds to a primitive in space. For example, row 0 of the table corresponds to $(72, 7, 100)$ in space. Then, using the point as its vertex, RayDB creates a right triangle as the primitive. The reason why we choose triangles as primitives is that only the ray-triangle intersection test is hardware-supported by RT cores, while the intersection tests for other types of primitives are software-based and offloaded to CUDA cores. Therefore, the use of triangles allows RayDB to enhance performance by exploiting hardware acceleration from RT cores. Specifically, if the coordinate of a data record is (a, b, c), then the three vertex coordinates of the right triangle we create are (a, b, c), $(a + S_x, b, c)$, and $(a, b + S_y, c)$, where S_x and S_y are the two leg lengths of the right triangle. Therefore, the projections of a primitive on the XZ-plane and YZ-plane are a line segment S_x and a line segment S_y, respectively. In this case, each data record in the table is mapped to a triangle in three-dimensional space, as shown in Figure 6a. Once all primitives in the space are determined, the BVH can be built.

Executing the RT job: With the selected BVH, RayDB initiates the execution of the RT job by launching a set of rays. For query A, RayDB launches a set of parallel rays starting from the $Z = 60$ plane to the $Z = 100$ plane, along the positive direction of the Z-axis, as shown in Figure 6b. The rays launched should be dense enough to intersect all triangles in the region. Rays are launched as a two-dimensional array from the view of the XY-plane ($Z = 60$ plane), which have an interval of $\frac{1}{2}S_x$ along the X-axis and an interval of $\frac{1}{2}S_y$ along the Y-axis. Recall that when mapping data to primitives, the two legs of right triangles have lengths S_x and S_y, respectively. The design guarantees that a triangle can be intersected by at least one ray. As shown in Figure 6c, primitives may intersect one ray (triangle A) or three rays (triangle B), and in the limiting case primitives intersect at most three rays (triangle C)[2]. If the interval grows, there may be triangles that fail to intersect any ray. In turn, if the interval gets smaller, it increases the probability that a triangle is intersected by more than one ray, which degrades the performance. Rays entirely cover the query area $60 \leq Z \leq 100$, and triangles in the query area are bound to intersect rays, while triangles not in the query area are bound not to intersect any ray.

For students who pass the English examination, their triangles are in the query area $60 \leq Z \leq 100$. Thus, the set of triangles intersecting a ray is the set of students that satisfy the predicate of `Scan`. For each triangle that is intersected by a ray, the Y-coordinate of its right-angle vertex is used to find the group to which it belongs, while the X-coordinate is read to compute the aggregate function, respectively. The aggregate function is *AVG* in Query A, so we maintain two arrays in the Any Hit Shader function, whose pseudo-code is shown in Algorithm 1. The *sum* array is to store the sum of *Math* for all students in each group, and the *count* array is to store the number of students in each group. Indexing by the Y-axis coordinate, RayDB appends the X-coordinate to the corresponding element of the *sum* array and increments the corresponding element of the *count* array by 1. The Any Hit Shader, which is executed on the SM as part of the shader pipeline, will be called each time a ray finds an intersection with a triangle. Therefore, the flag bit array V_{flag} (line 3) is used to ensure that triangles are not double-counted, and the atomic operation *atomic_add* (lines 5 and 6) is used to avoid synchronization issues when tracing multiple rays in parallel. After the BVH traversal is complete, the sum of the *Math* (V_{sum}) is divided by the number of students (V_{count}) to obtain the average score of *Math* in each group. Since aggregate functions share the common characteristic of operating on a group of values to return a single scalar result, their implementations are generally similar. As a result, RayDB supports all commonly used aggregate functions, including `AVG`, `SUM`, `COUNT`, `MAX`, and `MIN`.

[2]https://forums.developer.nvidia.com/t/what-is-the-limiting-case-of-ray-triangle-intersection/309730/2 (last accessed 2025/10/6)

Algorithm 1 Pseudo-code of Any Hit Shader

Input: flag bit array V_{flag}, result arrays V_{sum}, V_{count}
Output: result arrays V_{sum}, V_{count}

```
1: primIdx ← get_prim_index()
2: [a, b, c] ← get_prim_right_vertex_coord(primIdx)
3: flag ← atomic_bit_exch(V_flag[primIdx], 1)
4: if flag = 0 then
5:     atomic_add(V_sum[b], a)
6:     atomic_add(V_count[b], 1)
7: end if
```

4 ENCODING OF MULTIPLE DATA ATTRIBUTES

4.1 Challenges from Processing Multiple Attributes

Query A is a simple query where each operator involves only one data attribute. However, in real-world queries, it is common for an operator to involve multiple data attributes. When mapping Query A in the space, a primitive directly uses the value in the corresponding attribute as the coordinate on an axis. For instance, *English* is used as the coordinate on the Z-axis. However, when multiple data attributes are evaluated by one operator, the coordinate on one dimension needs to represent all data attributes involved. For instance, with the Where clause `WHERE English ≥ 60 and Math ≥ 60`, both English score and Math score should be represented by the Z-axis. To address this issue, in RayDB, we propose to encode multiple data attributes as the coordinate on each axis. In addition, the encoding scheme brings an extra benefit: it compresses multiple attributes into a single coordinate, allowing a query to utilize one or an arbitrary combination of the encoded attributes. This enables a single BVH to support a wider range of queries, thereby reducing the number of pre-built BVHs required. However, since different operators have different functionalities, an appropriate encoding scheme needs to be chosen for each one. In this section, we study how to encode attributes for `Scan`, `GroupBy`, and `Aggregation`.

4.2 Scan with Conjunctive Predicates

The encoding scheme for `Scan` needs to maintain the relative order of the encoded data and specify the ray launching area to ensure correct execution. A query generally contains multiple conjunctive predicates, like $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, and the attributes involved in the predicates can be encoded in the same attribute.

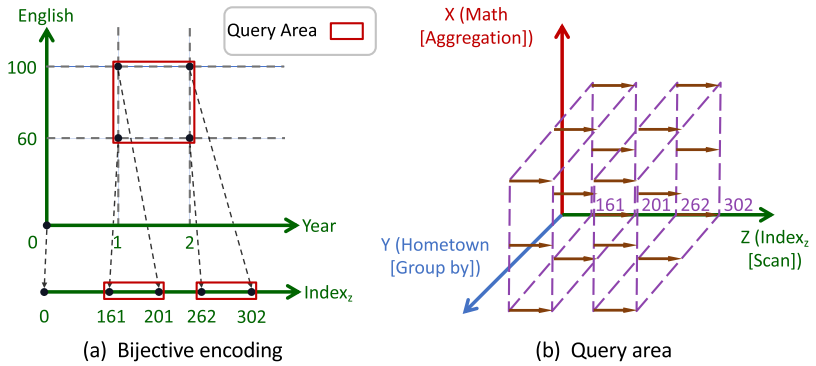


Figure 8: Processing for conjunctive predicates

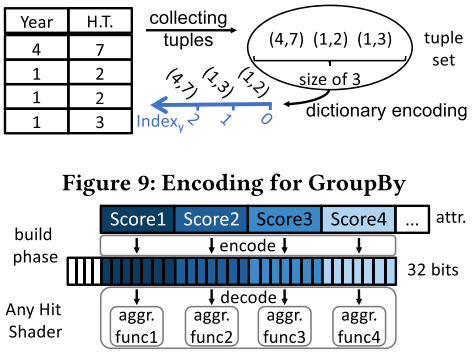


Figure 9: Encoding for GroupBy

Figure 10: Encoding for Aggregation

Data Encoding: RayDB adopts Bijection Encoding to encode multiple attributes in Scan as a coordinate. It treats all data attributes involved in conjunctive predicates as n-tuples, where each n-tuple is uniquely mapped to a natural number, and each natural number uniquely corresponds to an n-tuple. Assume that conjunctive predicates involves n attributes $A_1, A_2, \cdots, A_n$, where $A_i \in [0, k_i)$ and $A_i \in \mathbb{Z}\ (i \in [0, n])$, then the encoding rule is as follows:

$$Index_z = \sum_{i=1}^{n} \left[\left(\prod_{j=i+1}^{n} k_j \right) A_i \right]$$

If the data type of an attribute is not INTEGER, for fixed-point data types like DECIMAL(p, s) & NUMERIC(p, s), where p is the total number of digits and s is the number of fractional digits, RayDB uses Fixed-Point Encoding, which multiplies the decimal value by a fixed scaling factortypically $10^s$to convert it into an integer; for data types such as DATE, TIME, CHAR, VARCHAR, and ENUM, RayDB adopts dictionary encoding, where each distinct value is placed into a sorted dictionary and assigned an integer identifier representing its position in the sorted set of values. Therefore, it supports a wide range of data types. If the data range of an attribute is not of the form $[0, k)$, a dictionary encoding is used to convert it to the above form. Taking the query B as an example:

```
B: SELECT AVG(Math)
   FROM Score
   WHERE Year between 1 and 2 AND English >= 60
   GROUP BY Hometown;
```

For simplicity, it is assumed that the attributes involved in Scan are already encoded. Query B contains conjunctive predicates as its filtering conditions, which not only require students to have a passing score in English but also require students to be from either the first or second grade. RayDB assigns each $(Year, English)$ tuple with a natural number by the following rule:

$$Index_z = 101 \cdot Year + English$$

Here, 101 represents the range of $English$. Figure 8a depicts the above process.

Ray Launching: After Bijective Encoding, the Z-axis represents $Index_z$ generated by Bijective Encoding as an alternative. As a result, the query area changes as well. In the example, the query area is split into two parts, $161 \le Index_z \le 201$ and $262 \le Index_z \le 302$, as shown in Figure 8a. Correspondingly, as shown in Figure 8b, RayDB launches parallel rays from the $Z = 161$ plane to the $Z = 201$ plane and from the $Z = 262$ plane to the $Z = 302$ plane along the positive direction of the Z-axis to cover the entire query area.

4.3 GroupBy with Multiple Attributes

For data attributes involved in GroupBy, RayDB treats the n data attributes as n-tuples, where each distinct n-tuple represents a different group. Take query C as an example, which groups students according to two data attributes $Year$ and $Hometown$.

```
C: SELECT AVG(Math)
   FROM Score
   WHERE English >= 60
   GROUP BY Year, Hometown;
```

For all distinct n-tuples appearing in the table, RayDB maps them to natural numbers $Index_y$ via dictionary encoding. The correspondence between $Index_y$ and n-tuples is maintained by a mapping table that allows efficient lookup. After dictionary encoding, each natural number in $Index_y$ represents a group, which serves as the Y-coordinate. Taking query C as an example, table $Score$ is searched to find all distinct $(Year, Hometown)$ tuples. Figure 9 illustrates the above process. For simplicity of presentation, there are only 4 data records in the table. In the table, there are three different $(Year, Hometown)$ tuples $(4, 7)$, $(1, 2)$, and $(1, 3)$, making up a group set of size $K = 3$. Then, RayDB performs dictionary encoding to the group set, producing a coordinate $Index_y$ ranging from 0 to $K - 1$. As shown in the figure, $(1, 2)$ is assigned to 0, $(1, 3)$ is assigned to 1, and $(4, 7)$ is assigned to 2. This encoding scheme facilitates the implementation of aggregation because the Y-coordinate can be directly used as the index to the array that stores aggregate results for each group.

4.4 Aggregation with Multiple Attributes

For Aggregation, we choose to pack the bits of the attribute values in a 32-bit float coordinate so that they can be directly used for the aggregate function. For example, assume that there is a query containing two aggregate functions AVG(Math) and SUM(English). Considering that the data range of course scores is $[0, 100]$, we store $Math$ and $English$ with 7 bits each in the X-coordinate. In

Any Hit Shader, we obtain *Math* and *English* by decoding the X-coordinate and computing the two aggregate functions separately. Given that the coordinates have 32 bits and 7 bits are sufficient to store a single score, the X-coordinate can hold up to 4 scores. Therefore, the encoding can support queries with up to 4 aggregate functions, as shown in Figure 10.

For cases where an aggregate function contains multiple data attributes, e.g., `SUM(Math + English)`, the calculated result *Math + English* is directly encoded in the coordinate if the result does not exceed the expression range of a float. Multiple queries in SSB have such forms of aggregations. This scheme may help make further compressions. In this example, the range of *Math + English* is $[0, 200]$, which only needs 8 bits instead of 14 bits when being separately stored. RayDB adopts this optimization when possible to store more attributes in a coordinate.

4.5 Breaking through the Encoding Limitation

RayDB converts all data types to `INTEGER` for encoding, while BVH coordinates can only be represented using `Float`, which can accurately represent integers only within the range of $[-2^{24}, 2^{24}]$. Under this limitation, the maximum number of attributes that can be encoded into a single coordinate depends on the value range of each attribute involved in the encoding. Taking attributes from the SSB as an example, RayDB can encode up to four `LO_QUANTITY` (with a value range of 150) or six `LO_DISCOUNT` (with a value range of 010) into a single coordinate.

Although a coordinate can only encode a limited number of attributes, RayDB can support queries with more attributes. From a high-level perspective, when a BVH that covers all attributes in the query is unavailable in the pre-constructed BVH set, RayDB can utilize a BVH that encodes only part of the attributes to enable the query to still benefit from the acceleration provided by the RT cores. RayDB performs the RT job on the BVH, and upon detecting a ray-primitive intersection, the Any-Hit Shader that is called uses the primitive ID (i.e., row ID) to access the remaining uncoded attributes from the denormalized table in the GPU memory and perform the corresponding operations. Since reading attributes from the GPU memory incurs extra memory accesses, RayDB selects the optimal BVH by maximizing the attribute coveragei.e., choosing the BVH that encodes the largest number of attributes involved in the query. In particular, attributes in `Scan` are given higher weight, as the presence of `Scan` can significantly reduce memory access overhead. This selection process is formalized as $\max\{x + y + \alpha \cdot z\}$, where x, y, and z represent the number of the matched attributes in `Aggregation`, `GroupBy`, and `Scan`, respectively, and α is the weight assigned to `Scan`.

5 SCALING OUT WITH MULTIPLE GPUS

In RayDB, we store multiple BVHs in the device memory to support various queries. When BVHs exceed the device memory size of a single GPU, RayDB can only discard some BVHs. This results in more queries failing to find a BVH that encodes all of their attributes and instead choosing a BVH that covers only a part of the attributes, as described in Section 4.5. This incurs additional memory accesses, leading to performance degradation.

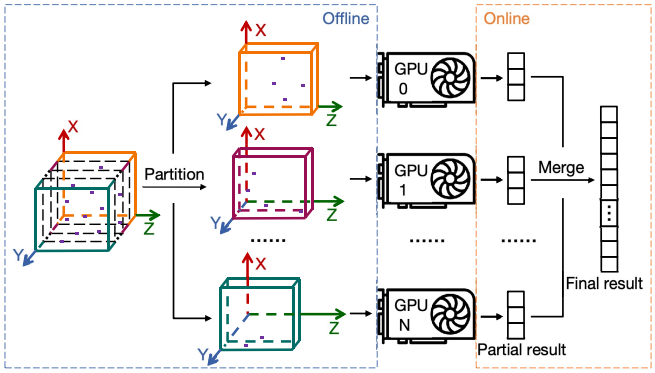


Figure 11: The workflow of parallel execution on multiple GPUs

To cope with this dilemma, we design a parallel execution scheme for RayDB on multiple GPUs, which can not only utilize the device memory of multiple GPUs to accommodate larger data sizes but also utilize the RT cores of multiple GPUs to improve query performance. The key idea is that the 3D space in RayDB can be naturally partitioned into any number of independent regions for parallel execution. First, all primitives can be divided into several sets according to the X-axis (`Aggregation`), Y-axis (`GroupBy`), or Z-axis (`Scan`) coordinates. After that, each set can be assigned to a GPU and used to build a specific BVH, which can be viewed as a sub-BVH. In this way, each GPU is supposed to intersect primitives to get partial results for the query. After all GPUs have completed processing, the final query result can be obtained by merging the results from all GPUs.

Figure 11 shows the parallel execution workflow when partitioning the data space along the Y-axis (`GroupBy`). We evenly divide the range of Y-axis coordinates into N segments, and the primitives located in the same segment are divided into the same group and assigned to the same GPU to build the BVH. Spatially, the entire data space is partitioned into a number of equal-sized subspaces along the Y-axis, and each subspace is assigned to a GPU, and each GPU launches rays in this subspace according to the same principles as a single GPU. Since we are partitioning the space along the Y-axis, the result obtained by each GPU is the query result for the groups involved within the corresponding segment. The final merge process is as simple as concatenating the results of each GPU in order to obtain the final query result. Since each primitive belongs to only one subspace, it is neither repeatedly accessed by multiple GPUs nor omitted, which ensures the correctness of the parallel execution result. In this way, all GPUs can be executed in parallel without any dependency. This approach can not only solve the problem of insufficient device memory but also accelerate the execution of queries.

The obvious benefit of partitioning the data space along the Y-axis is the simplification of merging. If we partition the space along the X-axis (`Aggregation`) according to a similar principle and the aggregate function is `SUM` as an example, the result obtained by each GPU is not the sum of some groups but the partial sum of all groups in the subspace. During the merge process, we need to sum

the result arrays of all GPUs together to get the query result, which is more time-consuming. In addition, if the data space is partitioned along the Z-axis (Scan), the query region may only be in a small number of subspaces, which will cause a serious load imbalance. In summary, we believe that partitioning along the Y-axis is a good choice for most cases.

The denormalized table can also be partitionedsplit by attributes and distributed across multiple GPUs. Unlike BVHs, each GPU may need to access attributes stored on other GPUs during query execution. With technologies like NVLink and RDMA enabling direct memory access between GPUs, RayDB can still support queries involving an arbitrary number of attributes in a multi-GPU parallel execution environment.

6 EXPERIMENTAL ANALYSIS

6.1 Experiment Setup

Hardware and Software We run most experiments on a machine equipped with two Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, 128GB DDR4 DRAM, and an NVIDIA GeForce RTX 4090 with 128 RT cores, 16384 CUDA cores, and 24GB VRAM. The operating system is 64-bit Ubuntu Server 20.04 with Linux Kernel 5.4.0-42-generic. The GPU programming interface uses CUDA 10.1 and OptiX 7.1.

In Section 6.3, we also compare the performance between RayDB and Crystal on NVIDIA TITAN X (PASCAL), where OptiX 5.1 is used for programming. The GPU was launched in 2015 and does not have RT cores. Therefore, ray tracing jobs are only executed on CUDA cores. The experiment aims to analyze the performance benefits from RT core acceleration. In Section 6.8, to evaluate the scalability of RayDB on multiple GPUs, the experiment was run on a machine with four NVIDIA GeForce RTX 4080 SUPER.

Workloads Throughout the experiments, we adopt the Star Schema Benchmark (SSB) [28]. There are a total of 13 queries in the benchmark, divided into 4 query flights. The industry usually flattens SSB into a wide table model (SSB flat) to test the performance of query engines. In our experiments, we run the SSB flat with a scale factor of 1, 10, and 20 to evaluate the performance with different data set sizes. When the scale factor equals 20, it will generate a flat table with 120 million tuples.

Baseline We compare RayDB with Crystal [32]. Crystal is a recently proposed state-of-the-art GPU database system that delivers superior query execution performance compared to other systems. It currently supports only queries from the Star Schema Benchmark (SSB). Specifically, Crystal implements a custom operator-invocation program for each query in the SSB. For a fair comparison, we modify these programs by removing all `Join`. Besides, since Cyrstal does not implement the `OrderBy`, we also remove it from all queries.

Encoding In SSB, we adopt different encoding schemes for the `Aggregation` operator. Three queries in Flight Q4 have the aggregate function: `sum(lo_revenue - lo_supplycost)`, and we adopt the encoding optimization in Section 4.4 to make further compression. The aggregate function of queries in Flight Q1 is `sum(lo_extendedprice * lo_discount)`. However, the range of $lo\_extendedprice * lo\_discount$ is too large to be precisely represented as an integer by a 32-bit float. Therefore, in experiments, we use the approach in Section 4.5 to handle this situation. In evaluating Flight Q1, the X-coordinate only represents $lo\_extendedprice$ while $lo\_discount$ is stored in the GPU device memory.

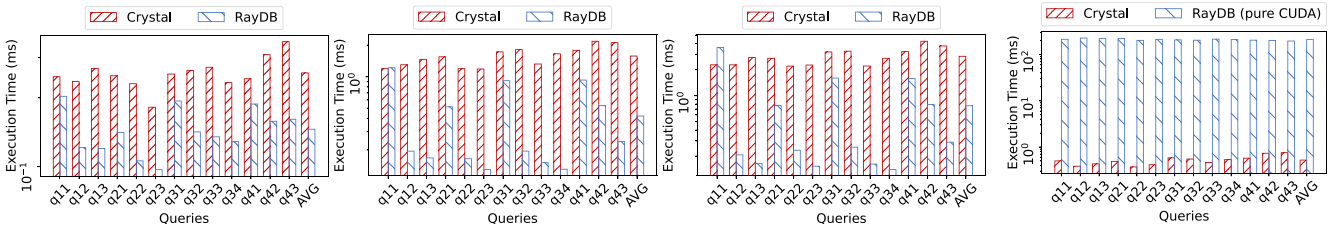
Table 1: Selectivity of queries in SSB

Query	q11	q12	q13		q21	q22	q23
Sel.(%)	1.99	0.07	0.02		0.80	0.16	0.02
Query	q31	q32	q33	q34	q41	q42	q43
Sel.(%)	3.67	0.14	5.76E-3	7.33E-5	1.59	0.38	0.04

6.2 Evaluation of Query Performance

Figure 12 illustrates the performance comparison between RayDB and Crystal. In order to ensure the fairness of the comparison between systems, the query execution time in the experiment is unified as starting after loading all input data into GPU memory and ending after the results are calculated. RayDB shows excellent performance on the SSB flat. At SF=1, RayDB is faster than Crystal on all queries, on average, by 82.08%. At SF=10, RayDB is faster than Crystal on 12 out of 13 queries and 5.4× faster on average. At SF=20, the situation is similar to that at SF=10. RayDB is faster than Crystal on 12 out of 13 queries, at least 1.0× faster and at most 18.3× faster. Over the entire SSB flat, RayDB is on average 8.5× faster than Crystal. It can be seen that RayDB maintains its performance advantage over Crystal in all SF cases. Considering that Crystal is by far the state-of-the-art GPU database system delivering superior query execution performance compared to other systems, the performance improvement is reasonably satisfactory.

Table 1 records the selectivity of each query in the SSB flat. Referring to Figure 12, it is observed that the query execution time of RayDB has a certain positive correlation with the selectivity of the query. The selectivity of q11, q31, and q41 is significantly higher than the other queries, and their execution time is also longer. The fundamental reason for the positive correlation between query execution time and query selectivity under certain conditions is that a lower selectivity implies a smaller number of data records in the query area, that is, a smaller amount of data to be accessed. Among the above queries, although q11 does not have the highest selectivity (1.99%, the maximum selectivity is 3.67% of q31), its execution time is particularly long, and it is even the only query for which RayDB has a longer execution time than Crystal. The reason is the particularity of Flight q1. The queries in Flight q1 do not include the `GroupBy` operator. In our implementation, we treat such queries as if all data records belong to the same unique group. Recall that atomic operations are used in Any Hit Shader to avoid synchronization issues, but they also limit parallelism, which affects performance. In the presence of only one group, all atomic operations target the same scalar value, further preventing the parallelism advantage of RT cores from being fully exploited and slowing down the execution of q11. This is also confirmed by the shift of the execution time of q11 from SF=1 to SF=20: when the dataset is small (SF=1), there are also fewer atomic operations, and the restriction of parallelism is not obvious. Therefore, RayDB is faster than Crystal. When the dataset becomes larger (SF=20), the



(a) SF = 1 (b) SF = 10 (c) SF = 20

Figure 12: Query execution time of RayDB and Crystal

Figure 13: Performance comparison of RayDB (pure CUDA) and Crystal

atomic operations increase accordingly, and the disadvantage of limited parallelism is further highlighted. At this time, RayDB is inferior to Crystal.

6.3 The Performance Gains from RT Cores and BVHs in RayDB

In order to disentangle the gains obtained from RT cores and BVHs, we implement a pure CUDA version of RayDB where the entire ray tracing process is computed by CUDA cores on the GPU. By comparing it with Crystal, we can evaluate the performance impact of the BVH index in the absence of RT cores, and then verify the contribution of RT cores. To achieve this, we switch to OptiX 5.1, which uses CUDA cores to compute BVH traversal and ray-triangle intersection tests. Since OptiX 5.1 is an old version, it does not support NVIDIA RTX 4090. Therefore, the experiments in this subsection are performed on an NVIDIA TITAN X (Pascal). Limited by the device memory size of TITAN X, we conduct experiments with a scale factor of 1. The performance results are shown in Figure 13.

On the 13 queries of SSB, RayDB (pure CUDA) is 258×-588× slower than Crystal, and 423× slower on average. Recall that RayDB is faster than Crystal on all queries with the same scale factor as in Section 6.2. This demonstrates that the BVH index alone does not work well due to the features of CUDA, and performance gains can only be achieved by cooperating with RT cores. With the application of hardware-accelerated ray tracing technology, the BVH traversal and ray-triangle intersection tests that originally needed to be computed by CUDA cores during the ray tracing process are offloaded to the RT core, which is specialized hardware designed for this purpose, freeing CUDA cores from thousands of instructions per ray, which could be an enormous amount of instructions for an entire ray tracing process. The presence of RT cores considerably accelerates the ray tracing process and makes the ray-tracing-based database possible.

6.4 GPU Memory Bandwidth Occupancy

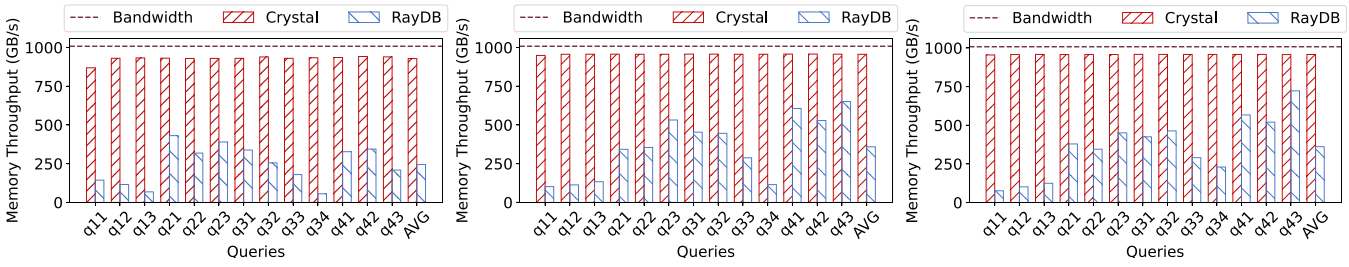
Figure 14 presents a comparison of the memory throughput of Crystal and RayDB on 13 queries of SSB flat. At SF=20, Crystal achieves a memory throughput of 97.11% to 97.51% of the memory bandwidth, with an average of 97.41%. The situation is rather similar in other SF cases. It can be argued that Crystal saturates the memory bandwidth. In contrast, at SF=20, RayDB's memory throughput is only 7.72% to 73.53% of the memory bandwidth, with an average of 36.74%. This ratio is even smaller in other SF cases. It can be seen that the memory bandwidth occupied by RayDB is considerably smaller than that of Crystal in all SF cases. This is due to the fact that RayDB drastically reduces the amount of data that needs to be accessed and the number of random memory accesses, freeing its performance from the memory bandwidth constraints. Based on this, we identify that the limiting factor for the performance of RayDB is the number of RT cores. The latest RTX 4090 is equipped with 128 RT cores versus 16384 CUDA cores. More RT cores mean that we can launch more rays to expedite intersection tests, and the performance is expected to be further enhanced. With the fast development of the ray tracing technique, the number of RT cores has also been increasing. The RTX 2080 Ti has only 68 RT cores, while this number grows to 84 on the RTX 3090 Ti and 128 on the RTX 4090. Therefore, since the memory bandwidth is no longer a limitation, RayDB is expected to benefit from the evolving architecture and achieve higher performance on future GPUs.

It is worth noting that there is no clear positive correlation between query memory throughput and query selectivity, which seems to contradict the conclusions we obtained. While RayDB reduces the amount of data that needs to be accessed with the help of ray tracing, it also increases the overhead of BVH traversal and ray-triangle intersection tests. Since the BVH is stored in the device memory, BVH traversal and ray-triangle intersection tests also require access to the memory, and this partial memory access is closely related to the BVH structure. The BVH structures corresponding to different queries show great differences, and their effect on the memory throughput shows strong stochasticity, ultimately shaping the results shown in Figure 14. Nevertheless, overall, the query memory throughput is still significantly degraded.

6.5 BVH Construction Overhead

Figure 15 shows the time of building the BVH in RayDB on each query of SSB flat for scale factors 1, 10, and 20. Experimental results show that the BVH building time scales linearly with the dataset size. From SF=1 to SF=10, the dataset size increases by about 9×, and the average building time increases by 9.95×. The dataset size roughly doubles from SF=10 to SF=20, while the average building time also grows by 1.01×. At a certain dataset size, the BVH building time is relatively stable, and the difference in building time between different queries does not exceed 11%.

In the three cases of SF=1, SF=10, and SF=20, the average BVH building time is 10.30 ms, 112.85 ms, and 227.84 ms, respectively,



(a) SF = 1 (b) SF = 10 (c) SF = 20

Figure 14: Query memory throughput of RayDB and Crystal

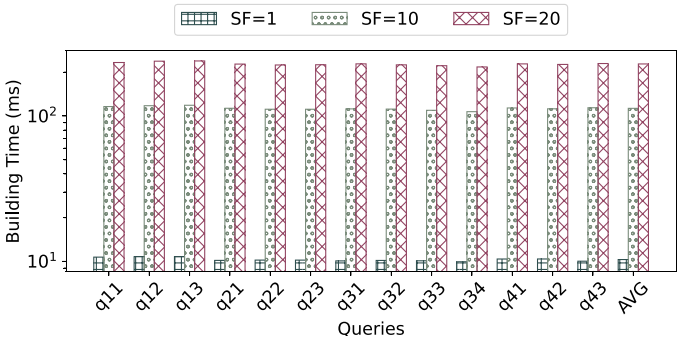


Figure 15: BVH building time of RayDB

while the average query execution time is 0.26 ms, 1.58 ms, and 2.85 ms, respectively. Therefore, the average execution time is 2.50%, 1.40%, and 1.25% of the average BVH building time, respectively. Due to the huge gap between the building time and the execution time, it is imperative to construct the BVH in advance. In RayDB, we build BVHs offline, which can be viewed as indexes or materialized views. For a given query, RayDB chooses a BVH that covers the attributes in the three operators.

6.6 Comparison between Query-Level and Operator-Level Ray Tracing

In Section 2.4, we explain why we choose to use ray tracing to accelerate the entire query rather than each operator. In the following, we conduct experiments to verify our conclusions. In view of the fact that there have been few studies on accelerating a single operator using ray tracing, which mainly focuses on the Scan operator (e.g., RTScan), we implement the Scan operator of the query by RTScan and the other operators by RayDB. RTScan transmits its execution result to RayDB in the form of a bit vector, where each bit records whether the corresponding data record satisfies predicates. RayDB enforces the Z-coordinate to be 0 for all primitives. Accordingly, rays are launched at the $Z = 0$ plane as the query area. In this way, all primitives will be intersected by rays, and RayDB needs to use the bit vector to determine whether a data record satisfies the predicates and should continue to participate in the following computation. Since RTScan suffers from out-of-memory issues when SF=20, we down-regulate SF to 16 in this experiment. The performance comparison with the case where RayDB implements the entire query is shown in Figure 16. The performance of ray tracing for individual operators exhibits a noticeable decrease when compared to ray tracing for the entire query. On the 13 queries of SSB flat, ray tracing for the entire query is 13.3× - 555.1× faster than ray tracing for individual operators and 194.9× faster on average. The reason for this staggering performance gap is that ray tracing is used to accelerate Scan and other operators separately, which means that RayDB (disable Scan) requires access to the entire dataset, taking away the benefit of ray tracing. Overall, this result is sufficient to demonstrate the inefficiency of ray tracing for individual operators and the necessity of choosing ray tracing for the entire query.

6.7 Comparison between Encoding and Splitting

In Section 4, faced with the case where a single Scan and GroupBy operator involves multiple data attributes, we give a solution that is encoding. However, there is a more intuitive way, which is to split the data attributes involved in the operator into two parts. The first part contains only one data attribute, which is still used for RayDB. The data attributes in the other part are stored in device memory in the form of arrays that can be accessed according to primitive indices. We refer to this approach as splitting.

Figure 17 is a plot of the performance comparison between splitting and encoding on the Scan operator when SF=20. In the experiments, splitting on Scan uniformly selects the first data attribute involved in the Scan operator for RayDB and stores the remaining data attributes in the device memory. When RT cores detect a primitive that intersects a ray, the corresponding Any Hit Shader reads from the device memory the remaining data attributes of the corresponding primitive and determines whether they satisfy the conjunctive predicates contained in Scan. Experimental results show that encoding on Scan performs significantly better than splitting on Scan. on the 13 queries of SSB flat, encoding on Scan is 3.3× to 176.9× faster than splitting on Scan, with an average of 31.1×. The reason for such a large performance difference is the difference in the amount of data that needs to be accessed. For example, if the conjunctive predicates are $p_1 \wedge p_2$, p_1 has a selectivity of 50% and p_2 has a selectivity of 20%, then encoding on Scan will access 50%×20%=10% of the data records, and splitting on Scan will access 50% of the data records. In this case, the amount of data they need to access differs by a factor of 5. Due to the different selectivity of each predicate in the conjunctive predicates, the performance difference between encoding on Scan and splitting on Scan on each

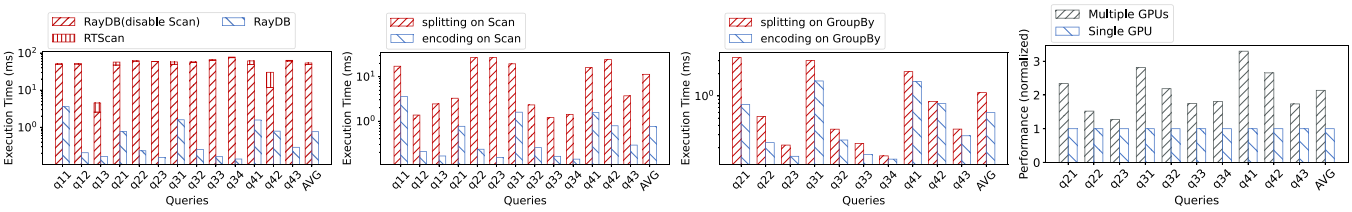


Figure 16: Performance comparison between RayDB and RayDB+RTScan

Figure 17: Performance improvement with encoding on Scan

Figure 18: Performance improvement with encoding on GroupBy

Figure 19: Scalability on multiple GPUs

query of SSB flat also shows a large difference. Overall, however, encoding on `Scan` outperforms splitting on `Scan` by a wide margin.

Figure 18 illustrates the performance comparison of splitting and encoding on the `GroupBy` operator when SF=20. Splitting on `GroupBy` is analogous to splitting on `Scan`. Since the queries in Flight q1 do not involve the `GroupBy` operator, the experiment is performed only on the remaining 10 queries of SSB flat. It can be observed that the difference in performance between splitting on `GroupBy` and encoding on `GroupBy` is smaller, 74.61% on average, compared to the difference between splitting on `Scan` and encoding on `Scan`, which is 31.1× on average. This is because splitting on `GroupBy` does not change the amount of data records accessed compared to encoding on `GroupBy` but only increases the overhead of accessing the remaining data attributes stored in the device memory. However, only the remaining data attributes of primitives that intersect rays will be accessed, so this overhead is not apparent in most SSB queries with generally low selectivity. Regardless, given the slight performance benefit, encoding on `GroupBy` remains the better option.

6.8 Scalability on Multiple GPUs

We evaluate the performance of RayDB on a single RTX 4080 SUPER and four RTX 4080 SUPER with a scale factor (SF) of 20, and the experimental result is shown in Figure 19. In this section, we only adopt the parallel execution strategy where the data space is equally partitioned along the Y-axis as an example of parallel execution. For the same reasons as in Section 6.7, we exclude Flight q1 from our experiments. The experimental result indicates that the performance of four GPUs working in parallel is 2.1× than that of a single GPU on average and can reach 3.3× at most.

While parallel execution on multiple GPUs significantly enhances performance, further improvements are constrained by the following two main factors: Firstly, the preparatory work before launching rays (e.g., determining the query region) needs to be performed on each GPU and cannot be parallelized. The time spent in this part has small differences between queries and is relatively fixed, approximately 0.07 ms in this experiment. As a result, for queries with long execution times, the performance improvement is large because the time taken for preparatory work is a small proportion of the execution time and vice versa. Secondly, the simple strategy of partitioning the space equally leads to a certain load imbalance between each GPU, which ultimately affects the performance. It is our future study to explore a more reasonable way of partitioning to achieve load balancing among GPUs.

7 RELATED WORK

Applications of RT cores Since RT cores are originally designed to render physically correct reflections, refractions, shadows, and indirect lighting in real time [27], there are numerous studies that leverage it to accelerate rendering workloads, including graphic rendering [8, 16], ambient occlusion [7], and simulation in physics [10, 37]. In addition, there are also many studies that creatively use RT cores to accelerate non-rendering workloads such as data processing, including K-nearest neighbor search [25, 38], database scan [14, 21], range minimum queries [22], point location search [24, 34], and rendering of unstructured meshes [23]. In summary, RT cores demonstrate significant potential for accelerating both rendering and non-rendering tasks.

Accelerating queries with GPUs GPUs offer strong parallelism and high-bandwidth memory, making them an attractive candidate for accelerating database queries. Therefore, GPU-accelerated database systems have been widely and deeply studied for more than 10 years [9, 17–19, 30, 35, 36]. There are three types of computational cores on the GPU: CUDA cores, Tensor cores, and RT cores. CUDA cores are responsible for integer and floating-point operations, so most GPU database systems are based on CUDA cores. Among them, Crystal [32] is the state-of-the-art CUDA-based GPU database that provides superior query execution performance compared to other systems. Tensor cores provide significant speedups to matrix operations. To leverage the computational power of Tensor cores, TCUDB [15] maps query operators to efficient matrix operators and implements a Tensor-based GPU database. RT cores accelerate Bounding Volume Hierarchy (BVH) traversal and ray-triangle intersection tests in ray tracing, and RayDB is the first study to make use of them to accelerate database queries, filling the gap in related directions.

8 CONCLUSION

In this paper, we propose RayDB, a query engine that accelerates database queries with ray tracing cores. Instead of implementing each operator independently like CUDA-based implementations, RayDB maps the core operators in a query as one RT job. The approach brings several performance advantages, including accessing data with an optimized sequential access pattern, reducing the amount of data to be accessed, and exploiting the parallelism of RT cores. RayDB breaks the memory bandwidth restriction on query performance and significantly outperforms the state-of-the-art CUDA-based GPU and CPU query engines.

REFERENCES

[1] Analytics at Meta. 2022. Using LogTime Denormalization for Data Wrangling at Meta. https://medium.com/meta-analytics/using-log-time-denormalization-for-data-wrangling-at-meta-3b6fc050268a.

[2] Burak Bacioglu and Meenakshi Jindal. 2021. Elasticsearch Indexing Strategy in Asset Management Platform (AMP). https://netflixtechblog.com/elasticsearch-indexing-strategy-in-asset-management-platform-amp-99332231e541

[3] Piotr Bialas and Adam Strzelecki. 2016. Benchmarking the cost of thread divergence in CUDA. In *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I 11*. Springer, 570–579.

[4] BlazingSQL. 2021. BlazingSQL. https://github.com/BlazingDB/blazingsql.

[5] Peter A Boncz et al. 2002. *Monet: A next-generation DBMS kernel for query-intensive applications.* Ph.D. Dissertation. Ph. d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands.

[6] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 128–139.

[7] Kevin Egan, Frédo Durand, and Ravi Ramamoorthi. 2011. Practical filtering for efficient ray-traced directional occlusion. In *Proceedings of the 2011 SIGGRAPH Asia Conference*. 1–10.

[8] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. 2006. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. 41–50.

[9] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. 1603–1618.

[10] Franco Fuschini, Hassan El-Sallabi, Vittorio Degli-Esposti, Lasse Vuokko, Doriana Guiducci, and Pertti Vainikainen. 2008. Analysis of multipath propagation in urban environment through multidimensional measurements and advanced ray tracing simulation. *IEEE Transactions on Antennas and Propagation* 56, 3 (2008), 848–857.

[11] Google Cloud. 2025. *Use nested and repeated fields.* Google LLC.

[12] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877* (2022).

[13] HeavyDB. 2022. HeavyDB. https://github.com/heavyai/heavydb.

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

[15] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. Tcudb: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*. 1360–1374.

[16] Henrik Wann Jensen and Per Christensen. 2007. High quality rendering using ray tracing and photon mapping. In *ACM SIGGRAPH 2007 courses*. 1–es.

[17] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*. 555–569.

[18] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB experience of building a CPU/GPU hybrid database product. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2999–3013.

[19] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[20] 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.

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

[22] Enzo Meneses, Cristóbal A Navarro, Héctor Ferrada, and Felipe A Quezada. 2024. Accelerating range minimum queries with ray tracing cores. *Future Generation Computer Systems* 157 (2024), 98–111.

[23] Nate Morrical, Will Usher, Ingo Wald, and Valerio Pascucci. 2019. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)*. IEEE, 256–260.

[24] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. 2020. Accelerating unstructured mesh point location with RT cores. *IEEE transactions on visualization and computer graphics* 28, 8 (2020), 2852–2866.

[25] 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*. 289–300.

[26] Xuan-Thuan Nguyen, Hong-Thu Nguyen, Trong-Thuc Hoang, Katsumi Inoue, Osamu Shimojo, Toshio Murayama, Kenji Tominaga, and Cong-Kha Pham. 2016. An efficient FPGA-based database processor for fast database analytics. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1758–1761.

[27] NVIDIA. 2018. NVIDIA TURING GPU ARCHITECTURE. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[28] Patrick ONeil, Elizabeth ONeil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 237–252.

[29] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)* 29, 4 (2010), 1–13.

[30] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

[31] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2021. CFM: SIMT Thread Divergence Reduction by Melding Similar Control-Flow Regions in GPGPU Programs. *CoRR* (2021).

[32] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[33] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 411–420.

[34] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. *High Performance Graphics (Short Papers)* 7 (2019), 13.

[35] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.

[36] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.

[37] Zhengqing Yun and Magdy F Iskander. 2015. Ray tracing for radio propagation modeling: Principles and applications. *IEEE access* 3 (2015), 1089–1100.

[38] 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*. 76–89.