RayBooster: A Ray Tracing Engine to Accelerate SedonaDB [Industry]

Liang Geng
The Ohio State University
Columbus, Ohio, USA
geng.161@osu.edu

Rubao Lee
The Ohio State University
Columbus, Ohio, USA
lee.11875@osu.edu

Dewey Dunnington
Wherobots, Inc
Seattle, Washington, USA
dewey@wherobots.com

Feng Zhang
Wherobots, Inc
Seattle, Washington, USA
feng@wherobots.com

Jia Yu
Wherobots, Inc
Seattle, Washington, USA
jiayu@wherobots.com

Xiaodong Zhang
The Ohio State University
Columbus, Ohio, USA
zhang.574@osu.edu

ABSTRACT

Hardware acceleration is becoming increasingly critical for spatial databases, as their workloads are geometrically complex, data-intensive, and subject to growing real-time requirements. Building on our prior strong evidence that Ray Tracing (RT) cores can significantly accelerate spatial queries through dedicated hardware support, this paper presents *RayBooster*, the first solution that incorporates RT-core acceleration into a production-grade geospatial database system, Apache SedonaDB. Our approach not only delivers substantial performance improvements but also does so cost-effectively. We focus on spatial joins, which dominate execution time and computational cost in spatial databases. To enable this integration, we bridge the mismatch between spatial query engines and RT hardware through three key system innovations. First, to overcome the lack of random access in the standard Well-Known Binary format, we design a GPU-optimized Structure of Arrays storage layout. Second, we eliminate indexing scalability bottlenecks by constructing a monolithic Bounding Volume Hierarchy tree that encodes geometry IDs into the Z-axis of the geometric scene, bypassing the overhead of managing millions of micro-indexes. Third, to manage the combinatorial complexity of diverse geometry types and spatial predicates, we develop a unified *RelateEngine*. This engine utilizes RT cores to compute the Dimensionally Extended 9-Intersection Model, serving as a universal topological descriptor. Furthermore, we implement a memory-aware execution strategy to mitigate out-of-memory failures through robust, best-effort resource management. Seamlessly integrated as an extension to SedonaDB, *RayBooster* delivers up to a 5.93× performance speedup on SpatialBench and provides a 59.02% reduction in operational costs, effectively transforming these idle RT units into a highly efficient engine for spatial analytics.

PVLDB Reference Format:
Liang Geng, Rubao Lee, Dewey Dunnington, Feng Zhang, Jia Yu, and Xiaodong Zhang. *RayBooster*: A Ray Tracing Engine to Accelerate SedonaDB [Industry]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pwrliang/sedona-db/tree/vldb.

1 INTRODUCTION

The volume of geospatial data has grown exponentially with the widespread adoption of GPS-enabled devices, satellites, and automated driving systems, making large-scale spatial data analysis an increasingly urgent requirement [13, 75, 79]. To address this big-data challenge, researchers have developed a new spatial analytics ecosystem characterized by large-scale computing and storage capacity, high scalability, compatibility with low-cost commodity clusters. A representative early example of this effort is HadoopGIS [1]. Consequently, most modern OLAP databases adopt this "scale-out" design principle, processing data in batches to exploit massive parallelism across clusters. Following this paradigm with efforts from both academia and industry, the most widely used spatial database, Apache Sedona [71, 72], has been developed.

While distributed spatial systems effectively address rapidly growing data volumes, issues remain in processing complex geometries and meeting real-time interactive query requirements. These system-heavy databases are often ill-suited for agile analytics, where users iteratively modify SQL queries to explore data. In addition, they incur high end-to-end latency. For example, provisioning a distributed Apache Sedona cluster involves substantial deployment effort, machine costs, and inter-node communication overhead. Therefore, these systems are most cost-effective for massive datasets without strict low-latency requirements.

For spatial data processing with fast-response requirements, an alternative ecosystem has emerged based on single-node analytical engines running on powerful servers and following a "scale-up" paradigm. Research and development on these single-node engines has grown rapidly, enabling a wide range of applications [34, 49]. Representative systems include DataFusion [35], DuckDB [34, 54], GeoPandas [65], and Apache SedonaDB [3]. These systems have democratized access to OLAP capabilities by enabling interactive data exploration and visualization on local laptops and workstations.

doi:XX.XX/XXX.XX

However, this promise of "local interactivity" often breaks down due to a single machine's limited computational power. For example, spatial joins remain slow on CPUs [1, 28], causing queries over medium-sized datasets to stall for minutes or hours [2]. Such latency disrupts the interactive feedback loop and forces data scientists into costly waiting cycles. Spatial joins typically use a two-phase strategy: a filtering phase leveraging spatial indexes (e.g., R-tree [7, 11, 23]) to prune non-overlapping geometries, followed by a refinement phase evaluating geometric predicates [14] on the intersecting geometry pairs. Due to the limited capability of a single node, both stages may become performance bottlenecks.

In fact, a highly effective hardware accelerator for single-node spatial engines already sits on users' desks: Ray Tracing (RT) cores, widely available in modern consumer laptops and workstations [9]. During SQL execution, these cores remain idle because existing software stacks do not use them. Prior research demonstrates RT cores' potential to accelerate spatial workloads like spatial joins [18, 19], and k-Nearest Neighbor (KNN) search [44, 80]. However, no industrial-grade product yet exploits this capability. This gap motivates our research and development efforts to integrate RT-core acceleration directly into the production-quality SedonaDB engine, aiming to eliminate the "waiting game" in local spatial analytics.

While the algorithmic potential of RT cores for accelerating individual execution phases has been demonstrated in prototype systems and micro-benchmarks [18, 19], incorporating these techniques into a robust production engine introduces a distinct set of systematic challenges for us to address:

- **Data Format Mismatch.** SedonaDB relies on the Well-Known Binary (WKB) format to maintain compatibility with data sources in GeoParquet [50] and computational geometry libraries like GEOS [22]. While WKB is ideal for compact serialization, its stream-oriented structure prevents the random geometry access required by massive GPU parallelism. Efficient refinement requires a GPU-native format that supports random access to arbitrary geometry.
- **Indexing Scalability.** Handling complex geometries during refinement typically requires fine-grained indexing (e.g., building an index for every polygon). Constructing and storing millions of micro-indexes on the GPU is prohibitive due to memory allocation overhead and the high latency of host-device synchronization.
- **Combinatorial Complexity.** The WKB format supports diverse geometry types (`Point`, `LineString`, `Polygon`, and Multi-variants). Considering both sides of the join, coupled with various predicates (e.g., `ST_Contains`). Hardcoding GPU kernels for hundreds of geometry-predicate combinations is unmaintainable and operationally infeasible.
- **Memory Management.** The physical execution plan in SedonaDB processes many partitions in parallel. Without strict resource awareness, this aggressive scheduling often disregards GPU memory quotas, leading to frequent Out-Of-Memory (OOM) failures when scaling to industrial datasets. In addition, spatial joins involve many temporary device memory allocations, which can make "small joins" even slower than CPU-based joins due to the allocation overhead.

To bridge the data format gap, we introduce a GPU-optimized storage format based on a Structure of Arrays (SoA) design. By segregating offsets, vertices, and geometry types, this format compactly represents an array of geometries while enabling high-throughput random access. To resolve the indexing scalability barrier, we design a monolithic BVH construction schema. Instead of managing millions of small indexes, we arrange geometries on a 2D plane and encode geometry IDs into the unused Z-axis of the hardware-accelerated Bounding Volume Hierarchy (BVH). This novel mapping allows us to build a single, global index that saturates GPU resources without frequent host interaction.

To tame the combinatorial complexity, we introduce a unified RT-based *RelateEngine*. Rather than implementing ad-hoc kernels for every combination, this engine utilizes RT cores to compute the Dimensionally Extended Nine-Intersection Matrix (DE-9IM) [1]. This matrix serves as a universal topological descriptor, allowing a single optimized engine to resolve arbitrary geometry types and predicates. Finally, we adopt asynchronous memory allocation to reduce the allocation overhead and implement a memory-aware execution strategy that dynamically monitors available device memory and orchestrates chunk executions to prevent OOM errors.

We have developed an RT accelerated spatial join engine, called *RayBooster*, making the following contributions.

(1) **System Integration:** We provide the first integration of RT-core acceleration into a production database system, accelerating both filter and refinement phases.
(2) **RT-based RelateEngine:** We introduce a unified evaluation engine that computes DE-9IM using RT cores, supporting any combination of geometry types and spatial predicates with a single code path.
(3) **Robust Execution:** *RayBooster* inherits SedonaDB's memory-aware scheduling that ensures the completion of queries under a given memory budget while striving to maintain high-performance. An asynchronous memory allocation strategy is also used to reduce synchronization overhead.
(4) **Performance & Cost:** We demonstrate that our approach achieves up to 5.93× speedup over SedonaDB on SpatialBench (§8.2) while reducing 59.02% operational cost, offering a cost-effective solution for industrial spatial analytics.

2 BACKGROUND

2.1 Apache SedonaDB

SedonaDB is an OLAP engine designed for the high-performance analysis of spatial datasets that fit within a single node. Additionally, its architecture is not limited to a single machine. At Wherobots, it serves as the foundation of our next-generation WherobotDB production runtime. To handle distributed workloads, an internal project called SedonaGlue orchestrates multiple SedonaDB instances across a cluster. The specific mechanics of this orchestration fall outside the scope of this paper.

SedonaDB is implemented in Rust and built on Apache DataFusion [35], which provides an extensible query planner and a vectorized execution engine to treat geospatial operations as first-class citizens. Figure 1 shows the architecture of SedonaDB.

[1]This paper focuses exclusively on relational spatial joins; distance and KNN joins are reserved for future research.

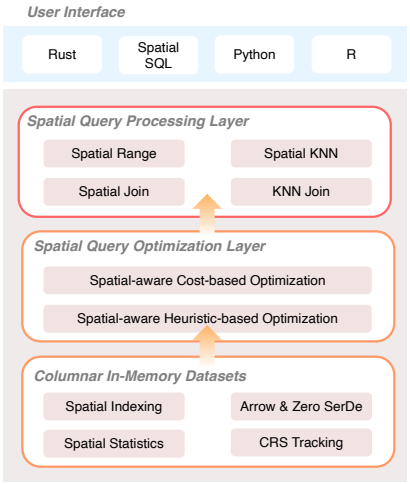


Figure 1: Architecture of Apache SedonaDB

Columnar In-Memory Datasets. SedonaDB adopts Apache Arrow as its in-memory storage format. While conventional spatial databases (e.g., PostGIS, DuckDB) store geometries as opaque BLOBs, incurring significant memory copy, serialization, and deserialization (SerDe) overhead. SedonaDB achieves zero SerDe cost by storing geometries as WKBs in Arrow binary arrays. This enables zero-copy data access and allows Arrow's computing kernels to exploit SIMD capabilities on modern CPUs to accelerate data manipulation. The storage layer also maintains spatial statistics (e.g., global bounding box, feature/point counts) to guide query optimization. SedonaDB also tracks the Coordinate Reference System (CRS) throughout the query lifecycle, ensuring data integrity by preventing operations on mismatched units or projections.

Query Planning and Spatial Join. SedonaDB implements a logical optimization rule that pushes spatial predicates into unconstrained join nodes, transforming Cartesian products into spatial theta-joins. This structural rewrite ensures the physical planner identifies the operation as an index-supported execution. The spatial join follows the build-probe paradigm. The build phase materializes the smaller relation to construct an in-memory spatial index, such as an R-tree. The probe phase then streams the larger relation, querying this index via Minimum Bounding Rectangles (MBRs) to identify intersecting candidate pairs. This lookup constitutes the filter stage, a major performance bottleneck due to the massive volume of random memory accesses during tree traversal, as our subsequent experiments show that this stage can account for more than 30% of the spatial join execution time (§8.5).

Execution then proceeds to the refinement stage, where geometric computations apply a spatial predicate to them. In SedonaDB, this task is delegated to high-performance computational geometry backends [6, 22]. With the WKB format, the engine eliminates the serialization overhead incurred when crossing library boundaries. This modularity ensures interoperability and allows the selection of the most efficient backend per data distribution. Based on storage-layer geospatial statistics, SedonaDB's cost-based optimizer may construct a feature-level index on either side to accelerate refinement for complex geometries. Finally, the refined candidate pair IDs are passed downstream to produce the query results. Determining the exact spatial relationship [12] often requires exhaustive comparisons of line segments, imposing a heavy computational load on processing complex geometries. In §8.5, we show that all queries take more than 60% time on the refinement stage.

Interfaces and Query Processing SedonaDB integrates with DataFusion by exploiting DataFusion's extensible optimizer API. In SedonaDB, geospatial functions (e.g., `ST_Intersects`) are registered as User-defined Functions (UDFs) within the DataFusion context. This allows the SQL parser to recognize spatial functions while maintaining compatibility with the standard SQL. SedonaDB supports both spatial join with relation predicates and KNN join. In addition, SedonaDB also provides interfaces for other programming languages for interoperability, including Rust, Python, and R.

Performance We use SpatialBench (§8.2) to compare SedonaDB against PostGIS, DuckDB, and GeoPandas on an AWS m7i.2xlarge instance. We focus on spatial relational joins: low-latency queries (Q2–Q6) and heavy joins (Q9–Q11). While Figure 2(a) shows PostGIS has competitive execution times, its significant index-building overhead negates this advantage for low-latency queries. DuckDB and SedonaDB achieve similar low-latency performance, whereas GeoPandas struggles at larger scales due to a lack of query optimization and multi-core underutilization. By contrast, DuckDB and SedonaDB leverage columnar layouts, vectorized execution, multi-core parallelism, and query optimization to achieve strong performance. SedonaDB excels on the intersection-over-union metric in Q9. For heavy joins (Q10–Q11), SedonaDB consistently delivers strong results, aided by an adaptive strategy that selects the best partition-level algorithm based on spatial statistics. Meanwhile, DuckDB encounters scaling issues on Q10 that hinder interactive processing, and it fails to complete Q11 within an hour. Figure 2(b) shows that on large-scale datasets, SedonaDB is the only system capable of completing all queries.

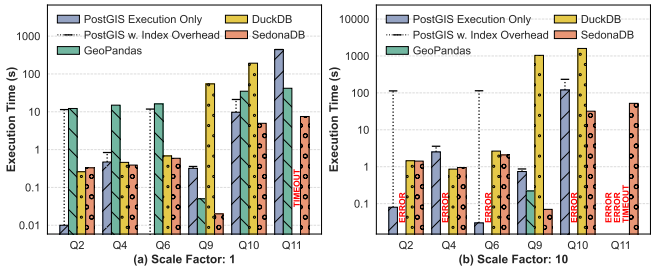
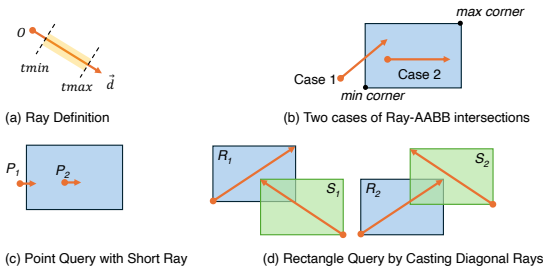


Figure 2: Performance comparison with SpatialBench.

2.2 Ray-tracing and RT cores

Ray tracing simulates light transport by tracing rays for photorealistic rendering. Historically computationally prohibitive for real-time applications due to massive ray-geometry intersection calculations, this changed when NVIDIA introduced RT cores [9]. These fixed-function hardware units accelerate the two most expensive pipeline stages: (1) BVH Traversal, locating potential intersecting primitives, and (2) Ray-Geometry Intersection, calculating the exact hit point. By offloading these tasks from general-purpose Streaming Multiprocessors (SMs), RT cores achieve orders-of-magnitude speedups over software implementations [9].



(a) Ray Definition (b) Two cases of Ray-AABB intersections

(c) Point Query with Short Ray (d) Rectangle Query by Casting Diagonal Rays

Figure 3: (a) and (b) illustrate the definition of the ray and the cases of ray-AABB intersection test; (c) illustrates the point query with *Case 2* and range query with *Case 1*

While originally designed for photorealistic rendering, the fundamental operation performed by RT cores, i.e., finding the intersection between a ray and a geometric primitive, is similar to many common spatial search problems. Recent research has demonstrated the efficacy of repurposing this hardware for non-rendering workloads, such as serving as an index for relational databases [26], database scan [40, 61], nearest neighbor search [42, 44, 80], matrix-multiplication [77], and spatial queries [18, 19].

Most research on RT core repurposing relies on Ray-Axis Aligned Bounding Box (AABB) intersection tests. As shown in Figure 3(a), a ray is a half-line originating at point O with direction vector $\vec{d}$, parameterized by t. There are two intersection cases. *Case 1: O* is outside the AABB, but the ray hits its boundary. *Case 2: O* is within the AABB, regardless of boundary hits. Mapping search problems to ray tracing leverages the optimized BVH traversal of RT cores, achieving significant speedups over conventional methods. To exploit RT cores, developers can use NVIDIA OptiX [51], a specialized framework built on the CUDA ecosystem. Within this pipeline, users implement programmable shaders: the AnyHit shader handles each potential ray-geometry intersection, while the ClosestHit shader is triggered by the intersection nearest to the ray origin.

2.3 Spatial Query by Ray Tracing

As mentioned in §2.1, the execution of the spatial join consists of the filter and refinement stages. They are both amenable to the RT acceleration, as demonstrated in previous research works, *LibRTS* [19] and *RayJoin* [18], respectively.

LibRTS introduces a novel approach to accelerate point and range queries in the filter stage by repurposing RT cores. Specifically, it maps a point query to a ray-box intersection test with *Case 2*, where the query point serves as the ray origin. As shown in Figure 3(c), we cast short rays from points P_1 and P_2 to identify their intersections. RT cores will traverse the BVH guided by the rays and report intersections to SMs. As a result, P_1 hits the boundary (*Case 1*) but is not within the box, so it is filtered out. P_2 is within the box and is included in the query results. For the range query, LibRTS employs a dual-pass strategy based on box-diagonal intersection tests (*Case 1*). This involves a "forward pass" that casts rays along query box diagonals against the BVH, followed by a "backward pass" that casts anti-diagonal rays to capture symmetric overlapping conditions. As shown in Figure 3(d), if boxes R_1 and S_1 intersect, one box's diagonal or anti-diagonal must intersect the other box. Simulating these (anti-)diagonals with rays allows the fixed-function RT cores to handle the computationally intensive tree traversal, significantly outperforming software-based indexing on CUDA cores.

While LibRTS optimizes the filter stage, RayJoin leverages RT cores to accelerate the computationally demanding refinement stage, specifically targeting Line Segment Intersection (LSI) and Point-in-Polygon (PIP) queries. RayJoin repurposes the RT rendering pipeline to map spatial joins to segment-segment intersections. For LSI, the framework utilizes the `AnyHit` shader to identify all intersecting segments. For the PIP query, RayJoin decomposes polygons into separated line segments with neighboring information, effectively casting it as a Point Location problem [17, 41]. RayJoin formulates this by casting a ray from the query point and using the `ClosestHit` shader to identify the closest line segment to the ray origin. By determining the orientation or relationship of this closest segment, RayJoin efficiently leverages the hardware's native intersection searching capabilities to accelerate the query.

3 *RAYBOOSTER* OVERVIEW

RayBooster is essentially an alternative physical plan implementation for SedonaDB, which employs a three-layer architecture: the *Storage Layer*, the *Indexing Layer*, and the *Refinement Layer*, as illustrated in Figure 4. The workflow begins with a CPU-based `ParallelWKBLoader` that parses WKBs in Arrow arrays from a spatial join. The Storage Layer transforms these into a GPU-optimized representation (§4.1) while dynamically classifying queries as either point-only or range queries based on their geometric extents. The Indexing Layer then dispatches these queries to specialized RT shaders, which are executed by OptiX. These shaders cast geometry-specific rays that traverse the BVH tree on RT cores to identify spatially intersecting candidate pairs (§5).

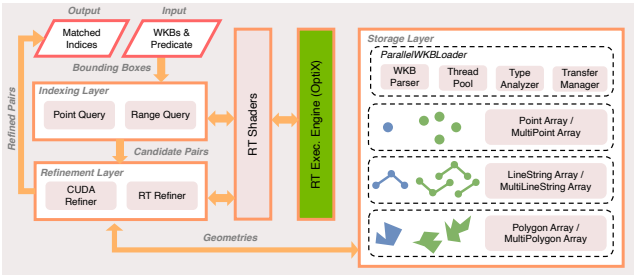
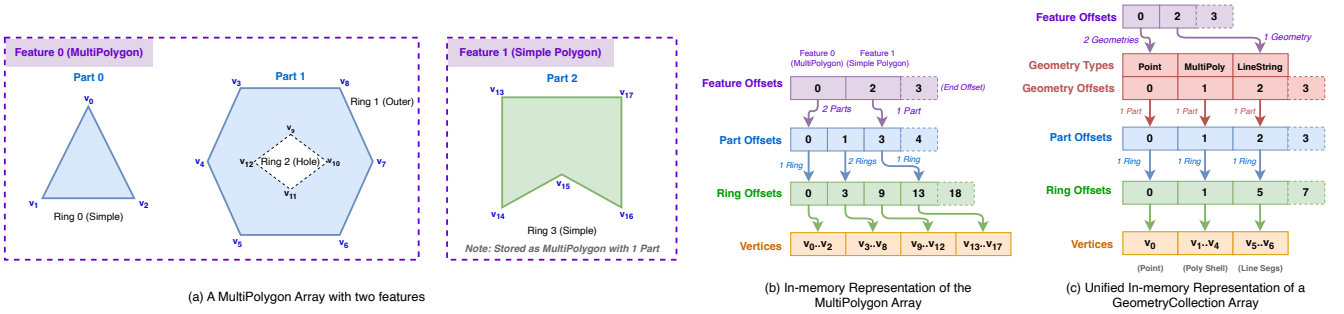


Figure 4: Architecture of *RayBooster*. Both filtering and refinement are offloaded to RT cores.

Because MBR intersections do not guarantee satisfaction of the spatial predicate, the refinement layer is necessary to eliminate false positives. *RayBooster* adopts a dual-path refinement strategy to ensure comprehensive operator coverage. An RT-based refiner processes computationally heavy, ubiquitous operators (e.g., PIP), while a standard CUDA-based refiner acts as a fallback for simpler or less common predicates. This design provides immediate system completeness for SedonaDB while facilitating the incremental transition of all operators to the RT pipeline. The final output—the IDs of verified, intersecting geometries—is then passed to the upper execution engines. We detail the refinement stage in §6.



(a) A MultiPolygon Array with two features

(b) In-memory Representation of the MultiPolygon Array

(c) Unified In-memory Representation of a GeometryCollection Array

Figure 5: (a-b) shows a `MultiPolygon` array and its in-memory storage format; (c) shows a unified `GeometryCollection` array that can contain any type of geometries

4 GPU-FRIENDLY STORAGE FORMAT

4.1 Storage Format

Our primary storage design principle is to support random geometry access, i.e., retrieving arbitrary polygon, ring, or vertices in $O(1)$. Another design goal is to utilize contiguous memory spaces to facilitate coalesced memory accesses, while maintaining a compact footprint to conserve limited GPU memory.

Geometry data is inherently hierarchical. As shown in Figure 5(a), a `MultiPolygon` array contains multiple features[2], each comprising various `Polygon` parts with an exterior ring and optional interior rings (holes). To store these in contiguous memory space, we use hierarchical offset arrays to navigate from features to parts, rings, and vertices. Vertices are stored in an ordered `double2` array to exploit vectorized LD/ST instructions. Figure 5(b) illustrates this using Feature, Part, and Ring Offset arrays.

Algorithm 1: Retrieve Feature i from `MultiPolygon` Array

Input : Feature index i; Offset Arrays O_F, O_P, O_R; Vertex Array V
Output : Set of rings (vertices) belonging to Feature i

```
/* Step 1: Identify range of Parts for Feature i */
p_start ← O_F[i], p_end ← O_F[i+1];
/* Step 2: Iterate over all Parts */
for j ← p_start to p_end − 1 do
    /* Identify range of Rings for Part j */
    r_start ← O_P[j], r_end ← O_P[j+1];
    /* Step 3: Iterate over all Rings */
    for k ← r_start to r_end − 1 do
        /* Identify range of Vertices for Ring k */
        v_start ← O_R[k], v_end ← O_R[k+1];
        /* Step 4: Access contiguous vertex data */
        Ring ← V[v_start ... v_end − 1];
        Process(Ring);
    end
end
```

Given a feature ID, we can directly access its parts, rings, and vertices using the offset arrays. Algorithm 1 outlines this hierarchical retrieval process, with Figure 5(b) providing concrete values. For instance, to retrieve `Feature 0`, we first consult the Feature Offset array (O_F). The range $[O_F[0], O_F[1])$ yields the start ($p_{start} = 0$) and end ($p_{end} = 2$) indices in the Part Offset array (O_P), indicating that `Feature 0` comprises two parts. We iterate through these parts (Line 3). For `Part 0`, the range $[O_P[0], O_P[1])$ in the Ring Offset array (O_R) reveals a single ring (`Ring 0`) spanning vertex indices $[0, 3)$. For `Part 1`, the range $[O_P[1], O_P[2])$ identifies two rings: `Ring 1` bounded by $[1, 2)$ in O_R (yielding vertex indices $[3, 9)$), and `Ring 2` bounded by $[2, 3)$ in O_R with vertex indices $[9, 13)$.

This design natively supports an array of `MultiPolygon` geometries, as well as mixed collections of `Polygon` and `MultiPolygon` objects (where `Polygon` is treated as a single-part `MultiPolygon`). Similarly, we also provided `LineString` and `MultiLineString` arrays, which require fewer offset layers.

`GeometryCollection` serves as a heterogeneous container for any geometry type. Despite its complexity, we represent it using hierarchical offset arrays. As depicted in Figure 5(c), this format introduces an additional layer tracking the specific geometry type and offsets for each part. Because `GeometryCollection` objects can be arbitrarily nested (e.g., `GeometryCollection(Point, GeometryCollection(Polygon, Point))`), we flatten these hierarchies into a linear sequence with depth (e.g., $[\texttt{Point}_0, \texttt{Polygon}_1, \texttt{Point}_1]$) to ensure efficient GPU traversal. While this unified format accommodates any type, its structural complexity incurs higher memory-access costs. Consequently, *RayBooster* dynamically prioritizes the most lightweight format. If a dataset contains only one type (or its `Multi` counterpart), we use the corresponding specialized, lower-overhead storage variant rather than universally using the `GeometryCollection` array.

4.2 High-performance WKB Loader

Converting WKB arrays into our GPU-friendly format poses a significant bottleneck due to the massive volume and irregular layout of real-world spatial data. A single WKB array might mix simple geometries with complex anomalies containing millions of vertices. Given this severe irregularity, *RayBooster* delegates this conversion to the CPU via the `ParallelWKBLoader`, leveraging the CPU's efficiency in handling complex, branch-heavy parsing. During loading, we employ a two-pass scanning strategy to collect metadata, further improving overall efficiency and resource utilization.

In the first pass, a `TypeAnalyzer` scans the data to identify unique geometry types and infers the least generic type capable of representing the entire array (analogous to type upcasting), thereby minimizing representation overhead. The capacities of offset containers are also approximated using the number of bytes of WKBs. With this information, we can pre-allocate offset arrays without

[2]A feature refers to a geometry object in a database record, such as a `Point` or `MultiPolygon`.

suffering from resizing overheads. Since we only read metadata of WKBs in this pass, it is very lightweight and fast.

The second pass is where the data conversion and loading happen. The WKB array is partitioned into chunks and distributed across a `ThreadPool` to eliminate thread-creation overhead during repeated spatial joins. Each thread will parse the chunked WKBs and count the number of parts, rings, and vertices. Then, a GPU-based `inclusive_scan` is used to create the offset arrays.

To mitigate straggler effects from irregular data, a load-aware `TransferManager` handles chunk sizing. Rather than splitting by element count, it calculates a byte-size prefix sum array and uses binary search to find optimal split points, ensuring threads process equal byte volumes. To prevent OOM errors during data loading, it dynamically sizes chunks based on available memory space.

5 FILTERING WITH RT CORES

The goal of the filter stage in spatial joins is to identify candidate pairs whose MBRs overlap. While conventional filtering relies on software-based spatial indices such as R-trees, *RayBooster* achieves high throughput by leveraging a hardware-accelerated BVH. In a typical two-way spatial join, the smaller table is designated the *build side*, while the larger one is designated the *probe side* (§2.1). Depending on whether the geometry has a spatial extent (e.g., `Polygon` vs `Point`), *RayBooster* employs different techniques for index construction and querying.

Figure 6 shows the entire workflow of the filtering pipeline. The process begins by constructing a BVH over the build side's MBRs (Step ①). Subsequently, the probe side issues parallel queries against the BVH (Step ②). These queries are dynamically classified as either point queries (§5.2) or range queries (§5.3) based on their geometric properties. Probe geometries without spatial extent trigger point queries, whereas those with extent trigger range queries (§2.3). The resulting intersecting pairs are immediately flushed to an `IntersectionBuffer` (Step ③), implemented as a concurrent, append-only queue. We elaborate on the details below.

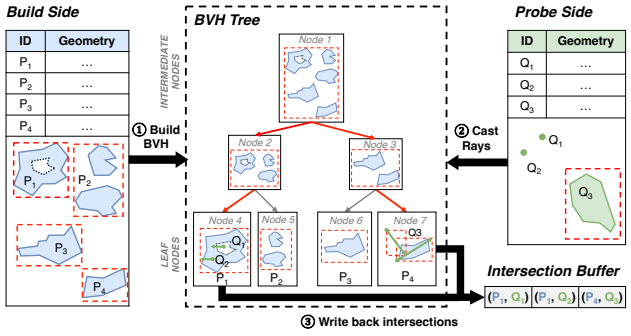


Figure 6: Filter Stage of spatial join by ray-tracing

5.1 Build the BVH

Indexing MBRs. For geometries with a spatial extent, we compute a tightly enclosing 2-D MBR to serve as a computationally inexpensive approximation for the intersection tests. Each MBR is mapped to a 3-D OptiX AABB by setting the z-axis to zero. This array of AABBs is then fed to the NVIDIA OptiX engine to construct the hardware-accelerated BVH[3]. OptiX exposes several build configurations, allowing developers to trade construction cost for traversal performance or memory compaction. By default, *RayBooster* is tuned for maximum query performance and minimal memory footprint, incurring only a minor compaction overhead. However, users retain the flexibility to override these defaults via a `SET` statement prior to executing their SQL queries.

Indexing Points. Because the OptiX does not natively support indexing points, we must circumvent this limitation to index zero-extent points. We use an AABB to enclose multiple spatial proximal points while curtailing dead space to reduce false positives. *RayBooster* first spatially clusters the points using a space-filling curve. An AABB is then generated to tightly enclose a cluster of spatially proximal points. This strategic grouping drastically reduces the overall index size and improves query performance.

Addressing Precision Limitations. A critical systems challenge arises from the precision limitation of ray-tracing frameworks. Operations in OptiX are strictly limited to FP32 precision [29], whereas robust geospatial databases mandate FP64 precision. Relying solely on FP32 can lead to false negatives and missed intersections. To address this issue, *RayBooster* adopts the "Conservative Representation" technique introduced in *RayJoin* [18]. This technique creates a just-big-enough AABB in FP32 that encloses the MBR in FP64, guaranteeing no loss of valid candidate pairs.

5.2 Point Query

RT-based point querying relies on the ray-box intersection test, where each query point is modeled as a short ray (§2.3). Upon constructing the BVH, we cast short rays from the probe points. Each intersected AABB will be returned to SMs by RT cores. We subsequently retrieve their original MBRs with the AABB IDs, and perform an FP64 intersection test against the query point. Figure 6 demonstrates the above process, where Q_1 and Q_2 represent query points. We cast rays from the points as origins against the build-side BVH. The hardware traversal begins at the root node (Node 1), navigates through the internal node (Node 2), and terminates at the leaf node (Node 4), successfully yielding the candidate intersections (P_1, Q_1) and (P_1, Q_2). The BVH traversal is conducted on the RT cores and the intersections can be collected with the `AnyHit` shader.

5.3 Range Query

Range querying requires a more complex, two-pass diagonal-MBR intersection test (§2.3). For MBRs on the probe side, we first cast rays along their diagonals against the build-side BVH to identify intersecting AABBs. Then, a symmetric second pass is executed: rays are cast along the anti-diagonals of the build-side AABBs against a BVH for the probe-side. As demonstrated in [19], this dual-pass ray casting guarantees the identification of all overlapping MBRs without duplication or omission. Q_3 in Figure 6 shows the execution of a range query. This query casts rays along the MBR diagonal of polygon P_4. A ray along the anti-diagonal of P_4 is also cast, but it does not intersect $Q3$. Finally, the identified intersection (P_4, Q_3) is appended to the intersection buffer.

[3]The BVH construction is executed on CUDA cores, whereas the subsequent traversal is accelerated by RT cores.

6 REFINEMENT WITH RT CORES

6.1 Solving Combinatorial Complexity

In a spatial join, the left and right sides can consist of any of the seven standard geometry types. When combined with over a dozen spatial predicates (e.g., `Contains`, `Touches`, `Crosses`), the number of unique refinement implementations approaches 500. Hardcoding each combination is tedious, error-prone, and hostile to system maintainability. To ensure `RelateEngine` remains a robust, maintainable component, we systematically reduce this complexity across both predicate types and geometry types.

Predicate Reduction. We map the evaluation of spatial predicates to one computation: computing the DE-9IM model. This matrix-based approach provides a systematic way to resolve the combinatorial explosion of topological relationships, especially when dealing with complex or multi-part geometries [14, 59]. This model is described by a 3×3 matrix that represents the intersection of the interior (I), boundary (B), and exterior (E) of two geometries. The matrix indicates the dimension of the intersection: F (no intersection), 0 (point), 1 (line), or 2 (area). Figure 7 shows the matrices for two Point-Polygon queries and one `Polygon-Polygon` query. After computing the DE-9IM model, a spatial predicate is evaluated via a logical AND operation between the predicate's mask matrix and the computed matrix. Figure 7(c) shows this for the `ST_Intersect` predicate. With the mask, the predicate evaluates to true if the 2×2 top-left elements are not all F.

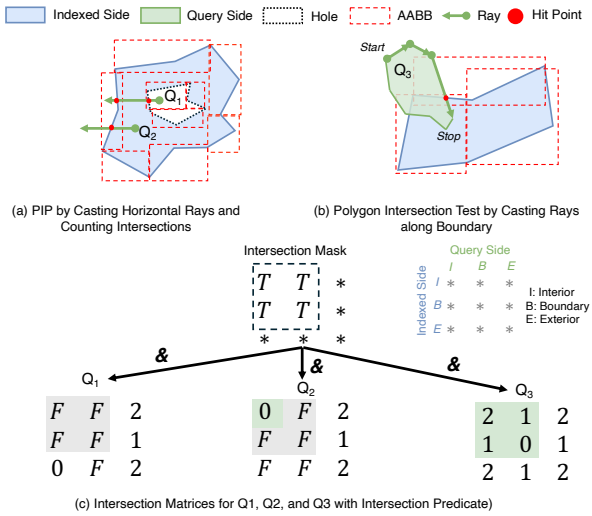
Geometry Type Reduction. To further reduce combinatorial complexity, we represent the singular WKB geometry types using their `Multi` variants; e.g., a simple `Polygon` is internally treated as a `MultiPolygon` with exactly one part. This aligns with formal spatial data models that generalize topological operations over multi-component spatial objects rather than treating simple geometries as isolated special cases [59]. Furthermore, we exploit the symmetric properties of the DE-9IM model. For example, the refinement logic for a `MultiPolygon-MultiPoint` join can be equivalently derived from the `MultiPoint-MultiPolygon` implementation by swapping the join sides and transposing the matrix.

6.2 Calculating DE-9IM Matrix with RT cores

Calculating the DE-9IM matrix is amenable to RT core acceleration. By simulating geometric vertices and line segments as strategically cast rays, RT hardware can evaluate all necessary interior, boundary, and exterior intersections.

Build BVH. For a given pair of candidate geometries, we construct an index at line-segment level for the more complex geometry (e.g., having more vertices), designating it the *indexed side*, while the other acts as the *query side*[4]. We employ an AABB to enclose adjacent line segments [5] on the indexed side. Rays are then cast based on the specific geometry type of the query side. We demonstrate this below for Point-Polygon and Polygon-Polygon relations.

Example 1: Point-Polygon Evaluation. Figure 7(a) illustrates the evaluation of a `Point-Polygon` relationship. For query points,



(a) PIP by Casting Horizontal Rays and Counting Intersections

(b) Polygon Intersection Test by Casting Rays along Boundary

(c) Intersection Matrices for Q1, Q2, and Q3 with Intersection Predicate

Figure 7: Demonstration of casting rays to find geometry element intersections and compute the DE-9IM model.

we cast an infinite horizontal ray. When the RT core detects a ray-AABB hit, we visit the enclosed line segments to perform an exact intersection test. *RayBooster* utilizes the ray-crossing (even-odd) algorithm to determine if a point lies within a ring [63]. In the figure, rays from both Q_1 and Q_2 intersect the exterior ring exactly once (an odd number), indicating they are inside the outer boundary. However, robust refinement requires verifying holes. We need to recast the ray for Q_1 to determine whether the ray intersects the polygon's interior. Because Q_1 is found to reside within a hole, it is correctly classified as being outside the polygon's interior. As shown in Figure 7(c), the resulting interior-interior cell for Q_1 is labeled F. Conversely, Q_2 does not intersect any holes. Its interior-interior intersection is a point, so the value is 0. In `RelateEngine`, the degenerated cases, like a point-on-boundary, are also checked and mapped to the corresponding DE-9IM model.

Example 2: Polygon-Polygon Evaluation. Polygon intersection testing is decomposed into boundary intersection tests, and each line segment can be simulated as a ray. As shown in Figure 7(b), assuming the top-left line segment is the first segment of the query polygon's exterior ring, we cast rays along that segment's trajectory. If no intersection occurs, we proceed to the next segment until a hit is found. In this example, the interior-interior intersection forms an area (matrix value 2), the interior-boundary intersections form line strings (matrix value 1), and the boundary-boundary intersection results in points (matrix value 0).

In practice, we do not need to calculate every DE-9IM matrix element, `RelateEngine` evaluates core topological relationships (e.g., interior-interior) and derives the rest using dimensional invariants. For instance, given a point and a polygon, the $E \cap E$ is invariably 2D, and $E \cap B$ is 1D. The engine only resolves the point's location: if inside, $I \cap I = 0D$; if on the edge, $I \cap B = 0D$. By populating the matrix with these deduced states, our unified engine achieves the same time complexity as dedicated, hardcoded predicates (e.g., `ST_Intersects`). Thus, the overhead of completing the full DE-9IM matrix via constant-time logical derivations is negligible.

[4]Note: These designations are localized to the refinement stage and are distinct from the term of *build* and *probe* sides used in the earlier filter stage.
[5]By default, a group of 32 adjacent line segments is enclosed by an AABB to fully exploit warp-parallelism for computing the AABB corners.

6.3 Monolithic BVH by Z-Stacking

Calculating the intersection matrix via RT cores requires indexing each individual geometry, meaning a spatial join with millions of features would require building millions of distinct BVHs. This per-geometry indexing approach is fatally bottlenecked by GPU synchronizations and massive BVH construction overheads. To solve this, we introduce a novel *monolithic BVH* design achieved through "Z-Stacking." Because hardware ray-tracing frameworks (like OptiX) are natively designed for 3-D space, and geospatial queries are confined to 2-D plane space, the Z-axis is entirely unused. We can repurpose this spare Z-axis to encode feature IDs [6] and build a monolithic BVH for all the features. We place the 2-D coordinates of each geometry feature onto a distinct X-Y plane, where the Z-coordinate corresponds to the feature's unique ID. This allows us to construct a single BVH for the entire batch of geometries.

Figure 8 illustrates this Z-Stacking technique in action. First, we extract the unique feature IDs on the indexed side from the candidate pairs. For example, polygon P_1 is mapped to the X-Y plane at $Z = 1$, while polygon P_4 is mapped to the plane at $Z = 4$ (Figure 8(b)). Subsequently, the query-side geometries cast rays exclusively along their corresponding X-Y planes. As shown in Figure 8(c), Q_1 and Q_2 are constrained to $Z = 1$, while Q_3 is placed on the plane with $Z = 4$. Q_1 and Q_2 cast horizontal rays to count intersections with P_1. As evaluated previously, Q_1 falls within a hole (yielding no interior intersection), while Q_2 falls within the valid polygon interior (yielding a 0-dimensional point intersection). The resulting matrices are shown in Figure 8(d).

6.4 CUDA-based Implementations as Fallback

While mapping all refinement operators to ray-tracing provides immense performance benefits, the engineering effort required to translate various queries remains substantial, despite reduced combinatorial complexity. To guarantee that *RayBooster* functions immediately as a drop-in GPU-accelerated engine, the `RelateEngine` provides standard CUDA-based implementations for all operators as a fallback. Currently, *RayBooster* features fully optimized, RT-accelerated operators for Point-Polygon relationships and their `Multi` variants, effectively resolving the most demanding PIP queries. This dual-path architecture ensures usability while allowing us to incrementally port lower-priority operators to RT cores later.

7 MEMORY MANAGEMENT

7.1 Host Memory Management

Spatial joins require space for intermediate data structures, such as a spatial index and the geometries produced by the evaluated SQL expression. *RayBooster* inherits the robust execution framework of SedonaDB, which ensures the completion of execution under a given memory constraint while trying its best to maintain high performance. SedonaDB may partition the build and probe side using a KDB-Tree [56] to create balanced partitions, which are then spilled to disk as Arrow IPC files with low I/O overhead. SedonaDB uses DataFusion's `MemoryPool` mechanism to request memory allocation, and *RayBooster* inherits this mechanism to operate. When the spatial join's memory footprint reaches the pool's configured threshold, further allocations are denied, and SedonaDB will trigger the disk-spilling mechanism. *RayBooster* benefits this spilling workflow by replacing the default CPU-based spatial index with an RT-accelerated spatial index. When a spilled partition is loaded on demand, the "probe side" data streams through that partition, and it queries this RT-accelerated index to find matching geometries.

[6] To avoid FP32 precision limitations on the Z-axis, the original feature IDs are renumbered into a compact, zero-based contiguous sequence during BVH construction. This prevents integer overflow and guarantees exact ID retrieval without precision loss.

7.2 Device Memory Management

Reducing Allocation Overhead. Spatial joins require device memory allocations for geometries, BVHs, and worklists. Standard allocation calls (e.g., `cudaMalloc` and `cudaFree`) are expensive and cause host-device synchronization stalls [64]. *RayBooster* uses the RAPIDS Memory Manager (RMM) to solve this issue [55]. We use RMM's coalescing, best-fit pool sub-allocator. All on-device data structures are wrapped in RMM's `device_uvector` and `device_buffer` containers to use RMM's memory allocator. This approach bypasses `cudaMalloc`'s synchronization overhead and leverages the CUDA memory pool APIs for fast memory allocation.

Handling Unpredictable Buffer Sizes. Memory consumption during filtering is highly unpredictable because the exact number of candidate pairs is unknown. Gauging a safe buffer size over-allocates memory, therefore wasting space. To address this issue, *RayBooster* adopts a two-pass execution model, including an initial counting pass followed by a materializing pass. In the counting pass, we cast rays and count the number of intersections, which is used to allocate the exact required memory space. In the second pass, we cast the rays again and store the results in the allocated buffer. Since the RT-accelerated index is fast, this two-pass approach incurs negligible overhead and avoids memory allocation waste, supporting large queries with the very limited device memory.

Adaptive Chunking. Building the BVH in a single batch may exhaust device memory. We use adaptive chunking to determine a safe chunk size based on free memory and the BVH size. Since *RayBooster* operates on a pre-allocated memory pool by RMM, `cudaMemGetInfo` cannot accurately measure free memory, so we use RMM's `tracking_resource_adaptor` to compute the unused space in the memory pool, along with unclaimed device memory.

Under Unified Memory Architecture. The newly introduced NVIDIA DGX Spark [48] may simplify memory management. This paradigm renders device allocators (RMM) unnecessary, as RT cores natively access regular pointers without explicit synchronizations. Hardware-managed page-faulting and large shared memory pools supersede complex software workarounds for unpredictable buffer sizing. Furthermore, out-of-core disk spilling transitions from a device constraint to a system-wide limit. This architectural shift abstracts low-level memory orchestration, allowing *RayBooster* to efficiently process irregular spatial workloads while simplifying software design. Optimizing for DGX Spark remains future work.

8 EVALUATION

8.1 Settings

Baselines. As this paper focuses on single-node OLAP systems, we exclude distributed frameworks like Apache Sedona. Since SedonaDB's advantages over DuckDB and GeoPandas are established in §2.1, we omit them. Our primary comparison evaluates SedonaDB

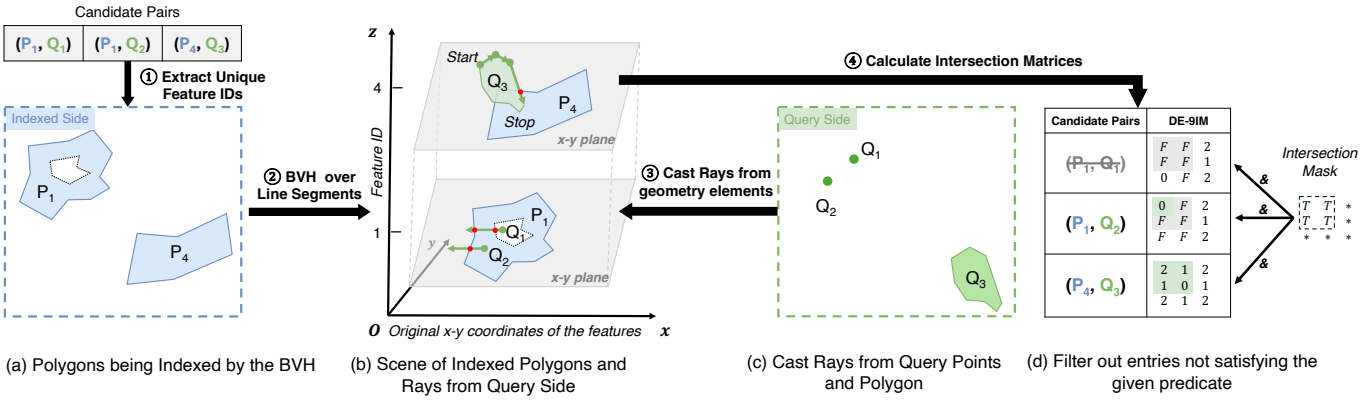


Figure 8: Execution workflow of RT-based refinement with Z-stacking technique. (a): Polygons on the indexed side; (b): Intersection Scene of the indexed polygons and queries; (c): Two query points and a query polygon; (d): Refinement results by applying a mask matrix to the intersection matrices.

with and without RT-acceleration by *RayBooster*. To the best of our knowledge, PG-Strom [30] is the only open-source spatial database that offers GPU acceleration by offloading spatial indexing and refinement to the GPU, but it relies on CUDA rather than RT cores.

Methodology. Historically, the field lacked a dedicated benchmark for spatial databases until the introduction of SpatialBench (§8.2), a sub-project under Apache Sedona. It provides a standardized, scalable, and open-source framework tailored for geospatial analytics. Its unbiased methodology ensures reproducible performance comparisons across diverse runtimes and environments.

Metrics. We mainly use running time as the metric for performance evaluation. We conduct our experiments with one warmup iteration followed by five additional executions, reporting the average end-to-end SQL execution time excluding the warmup round. A roofline model could be helpful for analyzing GPU utilization [10]. Unfortunately, the current profilers do not expose any performance counters of RT cores due to their proprietary nature [43].

Hardware. Detailed in Table 1, evaluations are conducted across a variety of GPU architectures in a workstation, cloud, and the Ohio Supercomputing Center (OSC). Notably, while the NVIDIA A100 and H100 do not have dedicated RT cores, *RayBooster* remains functional on these GPUs via software-emulated BVH traversal provided by OptiX [47]. With no additional maintenance efforts, we can assess performance in both hardware-accelerated and emulated environments. Without explicitly specifying, we always evaluate SedonaDB on an AWS m7i.2xlarge instance.

Environment and Configuration. We install PGStrom v6.1 extension on PostgreSQL 15. To have a fair comparison [53], we set the PGStrom configurations to the values recommended on its official website and enable indexes on geometry columns. *RayBooster* is a library written in CUDA/C/C++, and it provides a C interface for interacting with SedonaDB via the Foreign Function Interface (FFI). We use Rust 1.90.0, CUDA 12, OptiX 8, and RAPIDS 25.12.00. The only configuration we tune for *RayBooster* is the DataFusion batch size, which is set to 100K for Q2-Q9 and 2M for Q10 and Q11.

8.2 SpatialBench

To evaluate the performance and scalability of *RayBooster* under realistic industrial conditions, we utilize SpatialBench [4], a modern,

Table 1: Hardware specifications and pricing.

GPU	RT Cores	FP64 Perf.	VRAM (Bandwidth)	CPU (RAM)	Platform / Unit Price
RTX 3090	2nd Gen (82)	556.0 GFLOPS	24GB (936 GB/s)	i9-12900 (64GB)	Workstation –
A10	2nd Gen (72)	976.3 GFLOPS	24GB (600 GB/s)	8 vCPUs (32GB)	AWS g5.2xl $1.21/hr
L4	3rd Gen (60)	473.3 GFLOPS	24GB (300 GB/s)	8 vCPUs (32GB)	AWS g6.2xl $0.98/hr
L40S	3rd Gen (142)	1.43 TFLOPS	48GB (864 GB/s)	8 vCPUs (64GB)	AWS g6e.2xl $2.24/hr
A100	SW Emul.	9.75 TFLOPS	40GB (1.56 TB/s)	EPYC 7H12 (472GB)	OSC –
H100	SW Emul.	33.45 TFLOPS	94GB (3.36 TB/s)	Xeon 8470 (1TB)	OSC –

open-source benchmark suite designed exclusively for geospatial SQL. While standard OLAP benchmarks like TPC-H are industry norms, they lack native geometry types. Consequently, they fail to adequately evaluate spatial bottlenecks arising from complex predicates, geometry types, and spatial joins. SpatialBench bridges this gap by combining a systematic benchmarking methodology with large-scale urban mobility datasets, such as the NYC Taxi and Limousine Commission datasets [45]. The benchmark queries are derived from real-world optimization challenges at WherobotsDB that capture representative customer spatial patterns. They serve a purpose similar to TPC-H for relational databases by providing practical, though not exhaustive, coverage of spatial operations.

Data Model and Generation The benchmark employs an extended star schema consisting of a Trip fact table (representing rideshare trajectories with spatial pickup and dropoff coordinates) and multiple dimension tables, such as Zone (hierarchical administrative boundaries) and Building (polygon footprints). To reflect the natural skew and density variations [32] of real-world geographic data, the synthetic data generator restricts geometries to continent-bounded polygons and distributes coordinates using a Hierarchical Thomas process [66]. This multi-level clustering algorithm realistically mimics multi-scale urban settlement patterns, producing heavy-tailed distributions of dense city centers surrounded by sparse rural outliers. This pattern places heavy demands on spatial indexing and partitioning strategies. We evaluate the systems on

Scale Factors (SF) 1 (1.8GB) and 10 (7.0GB), omitting SF100 (>56GB) as it exceeds typical single-node capacities.

Query Workload The SpatialBench workload comprises 12 analytical queries (Q1–Q12) that evaluate distinct facets of a spatial query engine. Since this paper focuses on spatial relational joins only (§1), we use a subset of SpatialBench, as listed in Table 2.

Table 2: Relational Spatial Joins from SedonaBench Query Workload. *: Refinement stage in Q9 fallbacks to CUDA.

Query	Description	Spatial Characteristics	Scenario
Q2	Count trips starting within Coconino County (Arizona) zone	• Point-in-polygon spatial filtering (ST_Intersects) • Subquery with spatial geometry selection • Simple aggregation on spatially filtered data	Count all trips originating within a specific administrative boundary (county) for regional transportation statistics.
Q4	Zone distribution of top 1000 trips by tip amount	• Subquery with ordering and limiting; Aggregation on spatially joined results • Point-in-polygon spatial join (ST_Within) • Multi-step query; spatial filtering, and grouping	Analyze the geographic distribution of high-value trips (by tip amount) to understand premium service areas.
Q6	Zone statistics for trips within a 50km radius of Sedona city center	• Polygon containment check (ST_Contains) • Point-in-polygon spatial join (ST_Within)	Analyze trip patterns in zones within a metropolitan area around a city center.
Q9*	Building Conflation (duplicate/overlap detection via Intersection over Union (IoU))	• Self-join with spatial intersection (ST_Intersects); Area calculations (ST_Area) • Geometric intersection operations; Complex geometric ratio calculations	Detect duplicate or overlapping building footprints in GIS datasets to identify data quality issues.
Q10	Zone statistics for trips starting within each zone	• Point-in-polygon spatial join (ST_Within) • Aggregation with multiple metrics • LEFT JOIN on zones with no trips	Analyze trip patterns and performance metrics for each administrative zone.
Q11	Count trips that cross between different zones	• Multiple point-in-polygon spatial joins • Filtering based on spatial relationship results	Identify inter-district/city trips to analyze cross-boundary travels.

8.2.1 Query Characteristics. Queries Q2, Q4, Q6, and Q9 represent small, latency-sensitive spatial joins, as one of the input relations contains very few rows after filtering. In these scenarios, the computational workload is insufficient to saturate the GPU. Consequently, the primary objective is to ensure that offloading to the GPU does not introduce performance regressions (e.g., due to data transfer overhead over PCIe) [38, 73]. Queries Q10 and Q11 are heavy joins that trigger a large volume of point-in-polygon operations.

8.3 Overall Performance

Table 3 summarizes the query results, including $SF = 1$ and $SF = 10$. We evaluated PGStrom on a g5.2xlarge instance, a standard configuration for GPU-based tasks (e.g., inference) at Wherobots. Since PGStrom is an extension for the PostGIS OLTP database, an exact apples-to-apples comparison is difficult. Therefore, we followed the SpatialBench methodology by explicitly listing both queries without indexing, indexed queries, and index-building times, providing readers with a transparent view of overall performance.

8.3.1 Queries with Latency Sensitivity Joins. As listed in Table 2, queries Q2–Q9 represent latency-sensitive joins. Considering execution time only, PGStrom demonstrates low latencies due to prebuilt indexes (e.g., Q6 is under 50ms). However, factoring in index construction time, which can be an order of magnitude longer, erases this performance edge. Figure 9 shows the execution time breakdown of PGStrom[7]. We can see that for Q2 and Q4, most of the time is spent on scanning operations, and the spatial-related functions are not a bottleneck, while Q9 is bottlenecked by the filter stage.

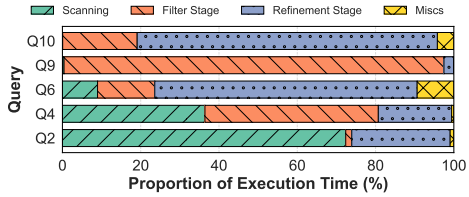


Figure 9: SQL execution breakdown (SF=1) of PGStrom Indexed. "Miscs" include sort/group/aggregate operations.

SedonaDB overperforms PGStrom due to its in-memory design, despite building indexes at runtime. When enabling *RayBooster*, execution times are comparable to or slightly faster than SedonaDB on the CPU across various GPU models. Interestingly, profiling reveals *RayBooster* also reduces table scanning times, a counterintuitive phenomenon explored in §8.5. These results align with our expectations. Since SedonaDB is not bottlenecked by these lightweight queries, enabling RayBooster successfully avoids adding latency overhead

8.3.2 Queries with Heavy Joins. Queries Q10 and Q11 involve computationally heavy joins, which are the cases where *RayBooster* excel. As shown in Listing 1, Q11 performs a two-way spatial join that generates intensive PIP operations, which is the exact workload that *RayBooster* offers to optimize.

```
SELECT COUNT(*) AS cross_zone_trip_cnt
FROM trip_geom t
JOIN zone_geom pickup_zone
  ON ST_Within(t.pickup_geom, pickup_zone.geom)
JOIN zone_geom dropoff_zone
  ON ST_Within(t.dropoff_geom, dropoff_zone.geom)
WHERE pickup_zone.z_zonekey != dropoff_zone.z_zonekey
```

Listing 1: SQL query of Q11: calculating cross-zone trips.

When SF=1, SedonaDB's CPU implementation required 7.51s to execute Q11 on an m7i.2xlarge instance. Enabling *RayBooster* reduced this to approximately 2s across all evaluated GPU models. The fastest execution time for Q11 was 1.61s on an RTX3090, yielding a 4.66× speedup over the CPU baseline. Although the H100 lacks dedicated RT cores, its ultra-high memory bandwidth and FP64 throughput allow it to marginally outperform most models equipped with RT cores, especially for large-scale datasets. However, this advantage is not significant. For instance, the consumer-grade RTX 3090 completed the query in 6.91 seconds for Q11 when $SF = 10$, while H100 took 5.65s, merely 22.3% slower than the enterprise-grade H100. For Q10, PGStrom took more than 23s for execution alone, where the majority of time was spent on Figure 9 on the refinement stage, while SedonaDB's adaptive join makes it only take about 5s, which is 4.69× faster. After enabling *RayBooster*, it delivers speedups on Q10 ranging from 2.09× to 3.94× across the tested GPU models when $SF = 1$. The advantage of *RayBooster* is more obvious for $SF = 10$, achieving speedups from 4.93× to 9.68×

[7]We cannot succeed on Q11 with "EXPLAIN ANALYZE" as PGStrom raises errors.

Table 3: Average execution times of SpatialBench queries in seconds with standard deviations. "PGStrom Build+Exec." represents the average SQL execution times plus the index build-up time on geometry columns used in the corresponding query.

Query	SF	PGStrom Indexed	PGStrom Build + Exec.	PGStrom No Index	SedonaDB	*RayBooster* on Various GPUs					
						RTX3090	L4	L40S	A10	A100	H100
Q2	1	0.17 ± 0.00	16.69	0.74 ± 0.09	0.33 ± 0.02	0.29 ± 0.01	0.35 ± 0.01	0.35 ± 0.02	0.42 ± 0.01	0.42 ± 0.04	**0.27 ± 0.01**
	10	0.44 ± 0.01	182.63	6.42 ± 0.05	1.50 ± 0.04	1.07 ± 0.02	1.56 ± 0.02	1.59 ± 0.03	1.91 ± 0.03	1.02 ± 0.05	**0.78 ± 0.00**
Q4	1	0.73 ± 0.00	1.51	35.19 ± 0.11	0.39 ± 0.01	0.30 ± 0.01	0.39 ± 0.02	0.39 ± 0.01	0.56 ± 0.20	0.32 ± 0.01	**0.22 ± 0.00**
	10	6.04 ± 0.01	7.97	105.20 ± 0.34	0.98 ± 0.06	0.74 ± 0.04	1.00 ± 0.01	0.97 ± 0.02	1.33 ± 0.14	0.58 ± 0.08	**0.32 ± 0.02**
Q6	1	0.01 ± 0.00	17.31	5.67 ± 0.03	0.59 ± 0.01	0.44 ± 0.01	0.62 ± 0.02	0.65 ± 0.03	0.89 ± 0.19	0.59 ± 0.02	**0.39 ± 0.01**
	10	0.04 ± 0.00	184.16	113.65 ± 0.32	2.21 ± 0.02	1.54 ± 0.11	2.35 ± 0.07	2.26 ± 0.06	2.89 ± 0.11	1.55 ± 0.18	**0.94 ± 0.02**
Q9	1	0.43 ± 0.00	0.48	47.10 ± 0.13	**0.02 ± 0.00**	0.04 ± 0.02	0.03 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.04 ± 0.00	0.03 ± 0.00
	10	1.13 ± 0.00	1.32	610.38 ± 1.19	**0.07 ± 0.00**	0.08 ± 0.01	0.10 ± 0.00	0.10 ± 0.00	0.12 ± 0.00	0.13 ± 0.00	0.10 ± 0.00
Q10	1	23.25 ± 0.53	39.77	Timeout	4.96 ± 0.09	**1.26 ± 0.08**	1.53 ± 0.04	1.58 ± 0.14	2.40 ± 0.70	2.37 ± 0.23	1.77 ± 0.14
	10	112.41 ± 6.46	294.60	Timeout	33.21 ± 0.27	4.03 ± 0.08	5.60 ± 0.17	4.99 ± 0.17	6.95 ± 0.37	5.55 ± 0.41	**3.43 ± 0.19**
Q11	1	246.24 ± 2.00	247.02	Timeout	7.51 ± 0.12	**1.61 ± 0.08**	2.01 ± 0.04	1.93 ± 0.03	2.57 ± 0.10	2.07 ± 0.05	1.66 ± 0.04
	10	Timeout	-	Timeout	53.34 ± 0.39	6.91 ± 0.35	10.85 ± 1.20	8.55 ± 0.05	13.52 ± 0.82	7.91 ± 0.18	**5.65 ± 0.11**

across all the GPU models. These results confirm that *RayBooster* is a highly effective performance booster for computationally intensive queries.

8.4 Cost-effectiveness

Evaluating raw performance metrics in isolation is insufficient for enterprise use cases, where financial cost is a critical factor. We calculate the operational cost of each query by multiplying its execution time by the hourly instance rate. AWS is the major cloud platform at Wherobots, so we evaluate *RayBooster* with various EC2 instances. The CPU baseline (SedonaDB) runs on an m7i.2xlarge instance, priced at $0.4032 USD per hour, which is the major instance type in our company. Table 1 details the GPU instance types and their corresponding hourly rates.

Figure 10(a) illustrates the per-query cost across the CPU and GPU instances. For latency-sensitive joins (Q2-Q9), the cheapest GPU instance g6.2xlarge consistently incurs higher costs than the CPU baseline. However, for computationally heavy joins, GPU acceleration directly translates into cost savings. For example, executing Q10 (SF=10) on the CPU costs approximately $0.003720 USD, whereas the L4 GPU reduces this to $0.001524 USD, yielding a 59.02% cost reduction due to a steep 5.93× speedup. Figure 10(b) aggregates the total execution cost across all queries. The results demonstrate that the A10 and L4 GPU models deliver overall cost savings. In contrast, the L40S instance incurs a higher cost than the CPU baseline due to its steep hourly rate. When factoring in both the extra costs for small queries and the substantial savings on heavy joins, *RayBooster* reduces the total workload cost by 38.11% when utilizing the L4 GPU.

8.5 Running Time Analysis

To understand the performance bottlenecks and analyze how *RayBooster* reduces execution time, we profiled the query execution. Figure 11 presents a breakdown of the major execution time components for a lightweight query (Q2) and a computationally heavy query (Q11) on a g6e.2xlarge instance. To observe the execution behavior and capture true wall-clock time, we set the engine to use a single partition. Without this setting, the profiler reports the aggregated CPU time across all partitions, which exceeds the actual wall-clock time. Thus, the absolute execution times in this targeted analysis differ from the end-to-end results presented in Table 3.

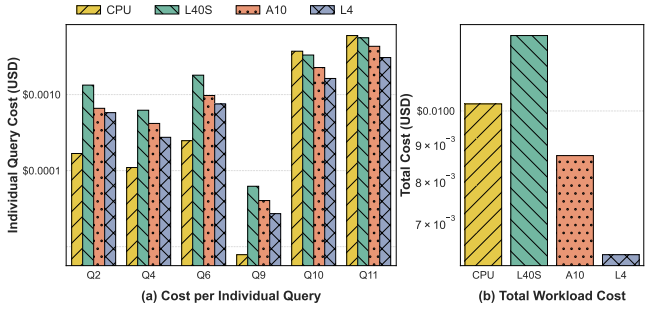


Figure 10: Cost-effectiveness analysis on different platforms with SpatialBench (SF=10); (a) log scale, (b) linear scale

The *Zone* table contains highly irregular polygon boundaries representing administrative zones. Parsing and processing these complex geometries consumes significant CPU resources in SedonaDB, extending scan times. Interestingly, although *RayBooster* targets spatial joins, upstream scan operations also execute substantially faster. This could be attributed to the offloading of computationally intensive spatial joins to RT cores, which reduces CPU contention, allowing the CPU to process scans more efficiently. Consequently, execution times for both operations are reduced. Isolating the spatial join itself, *RayBooster* achieves 4.62× and 4.07× speedups for Q2 and Q11, respectively.

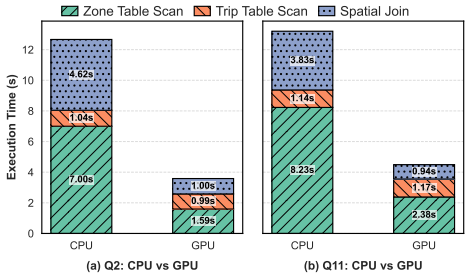


Figure 11: Comparing major time components of CPU and GPU joins (SF=1, Single Partition, g6e.2xlarge)

Figure 12 details the execution time breakdown of the spatial join operation. As shown, the refinement phase consistently dominates execution, consuming over 64% of the time across all queries, followed by filtering and WKB conversion. While the conversion

overhead is negligible for lightweight queries such as Q2, Q4, and Q6, it becomes a significant bottleneck for Q9 due to the large number of polygons processed on both sides of the join. Furthermore, index construction for the filtering phase is highly efficient, accounting for up to 4.2% of the total time across all queries.

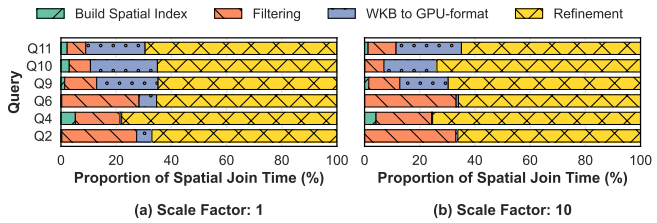


Figure 12: *RayBooster* execution time breakdown

8.6 Effectiveness of Memory Management

Figure 13(a) shows the performance impact of memory allocation strategies in *RayBooster*, comparing the default cudaMalloc against an asynchronous memory allocator used for dynamic structures like the BVH, IntersectionQueue, and temporary storage. The results indicate that asynchronous memory allocation is consistently beneficial, as it avoids unnecessary host-device synchronizations, yielding a 5% to 36% speedup on the SF=10 dataset. The performance gains are substantial, particularly for latency-sensitive joins, where we observe execution time improvements of 36% for Q2 and 27% for Q6. Furthermore, even for compute-bound queries such as Q10 and Q11, the asynchronous allocator provides notable speedups of 5% and 6%, respectively. Figure 13(b) illustrates *RayBooster*'s spilling behavior under memory constraints ranging from 5GB to 15GB, with Q11 memory-hungry three-table join. At a 5GB limit, heavy spilling slows execution to 88.93s, over 8× slower than pure in-memory processing (Table 3). However, performance improves almost linearly as the budget increases up to 9GB, with spilling relieved. Once the budget exceeds 11GB, it fully supports Q11, yielding near-in-memory performance. These results confirm *RayBooster*'s robustness in heavily memory-constrained environments.

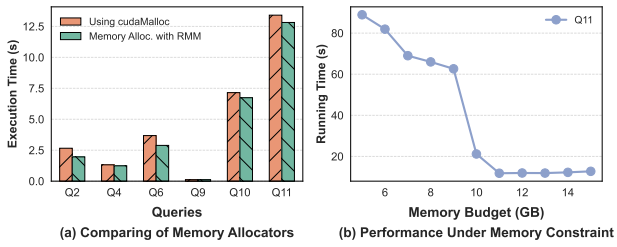


Figure 13: Effectiveness of memory optimization strategies

9 RELATED WORK

Repurposing RT Cores Several recent studies have leveraged RT cores to optimize KNN searches and spatial queries. For instance, RTNN improves KNN performance through spatial reordering and query partitioning [80]. TrueKNN supports KNN searches without a predefined radius [44]. JUNO achieves high-dimensional neighbor searches on RT cores [39]. Arkade [42] adapted KNN to support non-Euclidean space.

GPU Databases Over the past decade, substantial research into GPU-accelerated databases has yielded numerous prototypes and optimization techniques [8, 16, 24, 31, 37, 52, 57, 58, 60, 62, 67, 69, 70, 73, 74, 78]. Although this foundational work has led to the development of several production-grade GPU databases, native support for GPU-accelerated spatial queries remains scarce. For example, while the OLAP database HEAVY.AI [25] supports spatial operations for massive datasets, it does not offload these tasks to the GPU. RateupDB [36], a full-featured CPU/GPU HTAP system, lacks spatial querying capabilities. Kinetica [33], a distributed memory-first OLAP database, offers GPU acceleration for spatial, but does not use RT technology. Furthermore, empirical comparisons against it are precluded by its terms of service. Finally, PG-Strom [30] extends PostgreSQL with CUDA acceleration, serving as a primary baseline in our evaluation. *RayBooster* is the first production-level product that uses RT cores to accelerate spatial queries.

GPU Spatial Data Processing Despite the proliferation of GPU-accelerated spatial processing research, few solutions are production-ready. cuSpatial, a spatial processing library within NVIDIA's RAPIDS suite [46], uses an octree to index points and implements PIP operations via a CUDA-based crossing number algorithm. In contrast, our approach builds a polygon-level index and uses RT cores to accelerate PIP tests, achieving much faster performance than cuSpatial [18]. Other representative works [5, 68] include GPP, which offers a robust polygon overlay library using a uniform grid. RasterJoin [15, 76] leverages the conventional graphics rendering pipeline for spatial aggregation. While these standalone libraries advance GPU spatial processing, none provide a native SQL interface for seamless database integration.

Spatial Join Techniques The filtering stage of spatial joins can be further improved by using two spatial indices with the synchronized traversal technique [27]. However, it requires accessing the internal node of a spatial index, and the prohibition of explicit traversal with RT cores renders this technique unusable. Recently, Georgiadis et al. introduced a new technique to improve the efficiency of spatial joins with an intermediate filtering technique that utilizes raster interval approximations to deduce topological relations before evaluating exact geometries [20, 21]. These works are directly related to this paper.

10 CONCLUSION

Hardware acceleration is essential for spatial database systems due to their inherently geometric, data-intensive, and real-time workloads. Motivated by evidence that RT cores provide powerful support for geometric computation, we introduce *RayBooster*, the first production-grade integration of RT-core acceleration into SedonaDB. Targeting spatial joins, the dominant execution cost, we designed system-level mechanisms bridging the architectural gap between spatial query processing and RT hardware. Fully integrated into SedonaDB, *RayBooster* demonstrates up to a 5.93× performance improvement on SpatialBench while reducing operational costs by 59.02%. This shows that underutilized RT cores can be successfully transformed into a highly efficient, cost-effective engine for large-scale spatial analytics in production systems.

REFERENCES

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In *Proceedings of the VLDB endowment international conference on very large data bases*, Vol. 6. p1009.

[2] Apache Sedona. 2026. *SpatialBench Single Node Benchmarks.* https://sedona.apache.org/spatialbench/single-node-benchmarks/#__tabbed_2_2 Benchmark conducted September 22, 2025. Accessed: 2026-01-27.

[3] Apache Software Foundation. 2025. *Apache Sedona-db Documentation.* https://sedona.apache.org/sedonadb/latest/ Accessed: 2026-01-12.

[4] Apache Software Foundation. 2026. SpatialBench - Apache Sedona. https://sedona.apache.org/spatialbench/. Accessed: 2026-02-21.

[5] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and Efficient Polygon Overlay on Parallel Stream Processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 304–313.

[6] Josh Baker. 2026. TG: Geometry library for C. https://github.com/tidwall/tg. Accessed: 2026-02-05.

[7] Pedro Gabriel Kohl Bertella, Yuri Kaszubowski Lopes, Rafael Alves Paes de Oliveira, and Anderson Chaves Carniel. 2022. A systematic review of spatial approximations in spatial database systems. *Journal of Information and Data Management* 13, 2 (2022).

[8] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209.

[9] John Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44.

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

[11] Anderson Chaves Carniel and Cristina Dutra Aguiar. 2023. Spatial index structures for modern storage devices: A survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 9 (2023), 9578–9597.

[12] Anderson Chaves Carniel. 2024. Defining and designing spatial queries: the role of spatial relationships. *Geo-spatial Information Science* 27, 6 (2024), 1868–1892.

[13] Mingmin Chi, Antonio Plaza, Jon Atli Benediktsson, Zhongyi Sun, Jinsheng Shen, and Yangyong Zhu. 2016. Big data for remote sensing: Challenges and opportunities. *Proc. IEEE* 104, 11 (2016), 2207–2219.

[14] Eliseo Clementini, Paolino Di Felice, and Peter Van Oosterom. 1993. A small set of formal topological relationships suitable for end-user interaction. In *International symposium on spatial databases*. Springer, 277–295.

[15] Harish Doraiswamy and Juliana Freire. 2020. A GPU-friendly Geometric Data Model and Algebra for Spatial Queries. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1875–1885.

[16] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. NestGPU: Nested query processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1008–1019.

[17] William Randolph Franklin, Venkateshkumar Sivaswami, David Sun, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1994. Calculating the area of overlaid polygons without constructing the overlay. *Cartography and Geographic Information Systems* 21, 2 (1994), 81–89.

[18] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 124–136.

[19] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2025. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 396–411.

[20] Thanasis Georgiadis and Nikos Mamoulis. 2026. Scalable Spatial Topology Joins. In *EDBT*. 110–116.

[21] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. 2025. Raster Interval Object Approximations for Spatial Intersection Joins. *The VLDB Journal* 34, 1 (2025), 8.

[22] GEOS developers. 2026. *GEOS – Geometry Engine, Open Source.* https://github.com/libgeos/geos Accessed: 2026-02-05.

[23] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[24] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proc. VLDB Endow.* 15, 11 (July 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[25] HEAVY.AI. 2026. HEAVY.AI Documentation. https://docs.heavy.ai/

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

[27] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1 (2007), 7–es.

[28] Wenqi Jiang, Oleh-Yevhen Khavrona, Martin Parvanov, and Gustavo Alonso. 2025. SwiftSpatial: Spatial Joins on Modern Hardware. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.

[29] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[30] Kohei KaiGai. 2025. *PG-Strom: Master development repository.* https://github.com/heterodb/pg-strom

[31] Tomas Karnagel, René Müller, and Guy M Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@ VLDB* 8 (2015), 20.

[32] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. 2020. SpiderWeb: A Spatial Data Generator on the Web. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. 465–468.

[33] Kinetica DB, Inc. 2026. Kinetica: The Real-Time Database for AI and Analytics. https://www.kinetica.com/

[34] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. 2022. DuckDB-Wasm: Fast Analytical Processing for the Web. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3574–3577.

[35] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*. 5–17.

[36] 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.

[37] 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.

[38] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, et al. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4518–4531.

[39] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2023. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. *arXiv preprint arXiv:2312.01712* (2023).

[40] 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.

[41] Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Wenli Li. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. 45–54.

[42] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 14–25.

[43] Vani Nagarajan, Rohan Gangaraju, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. 2025. RT-BarnesHut: Accelerating Barnes-Hut Using Ray-Tracing Hardware. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/3710848.3710885

[44] 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.

[45] New York City Taxi and Limousine Commission. 2025. TLC Trip Record Data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2026-02-21.

[46] NVIDIA Corporation. 2023. cuSpatial: GPU-Accelerated Geospatial and Spatiotemporal Algorithms. https://github.com/rapidsai/cuspatial. Software available from github.com.

[47] NVIDIA Corporation. 2024. *NVIDIA OptiX 8.1 - Programming Guide.* https://raytracing-docs.nvidia.com/optix8/guide/index.html Accessed: 2026-05-05.

[48] NVIDIA Corporation. 2026. *NVIDIA DGX Spark User Guide.* NVIDIA. https://docs.nvidia.com/dgx/dgx-spark/dgx-spark.pdf Accessed: 2026-05-02.

[49] Observable, Inc. 2026. Observable: The Collaborative Data Platform. https://observablehq.com/ Accessed: 2026-02-28.

[50] Open Geospatial Consortium. 2025. *GeoParquet: Server-native spatial data in Parquet.* https://geoparquet.org/ Accessed: 2026-01-12.

[51] 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.

[52] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment* 14, 2 (2020), 202–214.

[53] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[54] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[55] RAPIDS Development Team. 2026. RMM: RAPIDS Memory Manager. https://github.com/rapidsai/rmm. https://github.com/rapidsai/rmm

[56] John T Robinson. 1981. The KDB-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[57] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 14. 708.

[58] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th international conference on scientific and statistical database management*. 1–12.

[59] Markus Schneider and Thomas Behr. 2006. Topological Relationships Between Complex Spatial Objects. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 39–81.

[60] 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.

[61] Xuri Shi, Kai Zhang, X Sean Wang, Xiaodong Zhang, and Rubao Lee. 2025. RayDB: Building Databases with Ray Tracing Cores. *Proceedings of the VLDB Endowment* 19, 1 (2025), 43–55.

[62] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 698–709.

[63] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. 1974. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)* 6, 1 (1974), 1–55.

[64] RAPIDS Development Team. 2023. *RAPIDS: Libraries for End to End GPU Data Science*. https://rapids.ai

[65] The GeoPandas Developers. 2026. *GeoPandas Documentation*. https://geopandas.org/en/stable/ Accessed: 2026-01-21.

[66] Marjorie Thomas. 1949. A generalization of Poisson's binomial limit for use in ecology. *Biometrika* 36, 1/2 (1949), 18–25.

[67] Diego G Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A Boncz. 2018. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *ADMS@ VLDB*. 1–10.

[68] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. In *Proceedings of the VLDB endowment international conference on very large data bases*, Vol. 5. 1543.

[69] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.* 7, 11 (2014), 1011–1022.

[70] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 44–54.

[71] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 1–4.

[72] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial Data Management in Apache Spark: the GeoSpark Perspective and Beyond. *GeoInformatica* 23, 1 (2019), 37–78.

[73] 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.

[74] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE international conference on big data (Big Data)*. IEEE, 273–283.

[75] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. 2020. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access* 8 (2020), 58443–58469.

[76] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freire. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.

[77] Hongrui Zhang, Yunan Zhang, and Hung-Wei Tseng. 2025. RTSpMSpM: Harnessing Ray Tracing for Efficient Sparse Matrix Computations. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 359–373.

[78] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[79] Yu Zheng. 2015. Trajectory Data Mining: An Overview. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 3 (2015), 1–41.

[80] 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.