



RT-RkNN: Reverse k Nearest Neighbor Queries as a Graphics Ray Casting Problem

Zhengyang Bai
RIKEN Center for Computational
Science
Kobe, Japan
zhengyang.bai@riken.jp

Peng Chen*
RIKEN Center for Computational
Science
Kobe, Japan
peng.chen@a.riken.jp

Mohamed Wahib*
RIKEN Center for Computational
Science
Kobe, Japan
mohamed.attia@riken.jp

ABSTRACT

Reverse k nearest neighbor (RkNN) queries are fundamental in spatial databases, location-based analytics, and recommendation systems. Existing state-of-the-art techniques rely on spatial pruning supported by R-trees and their variants. However, their pruning effectiveness degrades significantly in challenging scenarios where the number of facilities is small, the user population is dense, or the value of k is large. To overcome these limitations, we formulate the RkNN query in two-dimensional geometric spaces as a graphics ray casting problem, in which users are modeled as rays and facilities are represented as geometric primitives. Based on this formulation, we design the first algorithm and provide an implementation that exploits dedicated hardware ray tracing cores on modern GPUs. This novel approach preserves strong filtering performance even for large values of k , dense user populations, and highly sparse facility distributions. Extensive experimental results demonstrate that our method outperforms state-of-the-art algorithms in diverse settings, especially in scenarios where traditional pruning strategies become inefficient.

PVLDB Reference Format:

Zhengyang Bai, Peng Chen, and Mohamed Wahib. RT-RkNN: Reverse k Nearest Neighbor Queries as a Graphics Ray Casting Problem. PVLDB, 19(9): 1963 - 1976, 2026.
doi:10.14778/3819518.3819527

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/simon2/RT-RkNN>.

1 INTRODUCTION

Reverse k Nearest Neighbor (RkNN) queries are fundamental operators in spatial databases and location-based services, with applications spanning influence analysis, recommendation systems, and spatial impact assessment [6, 12, 18, 23, 27, 41, 56, 58]. Given a set of facilities F and users U , an RkNN query retrieves all users for whom a given facility is one of their k nearest neighbors. Unlike the standard k nearest neighbor (k NN) query that captures local proximity, RkNN characterizes a facility's *influence set* or effective

market reach, supporting operational efficiency and protecting substantial capital investments in domains such as retail site selection and delivery services [39]. Beyond spatial applications, RkNN has gained renewed attention in artificial intelligence, where modeling influence often outweighs similarity alone. It refines density estimation and cluster expansion in DBSCAN variants [5, 22], and mitigates hubness in graph neural networks, promoting more balanced graph structures and improved representation learning [44]. Due to their utility, significant research has focused on developing efficient RkNN processing techniques [8, 19, 27, 28, 45, 47, 50, 51, 60–63], as well as variants like dynamic, continuous, probabilistic, and batched RkNN [1, 3, 4, 7, 9–11, 14, 25, 29, 46, 49, 59].

A critical distinction from the standard k NN query is the nature of their search spaces. While a k NN query operates within a compact, query-centered region, an RkNN query must examine user-centric influence regions distributed across the entire data space, resulting in a significantly larger and more complex search area. To manage this complexity, most state-of-the-art solutions rely on spatial pruning strategies to eliminate candidates early. These include region-based pruning [50, 63] and half-space pruning [8, 51, 60, 62], typically built upon spatial indexes like the R-tree [20] or its variants (e.g., R*-tree [2]). These methods leverage geometric relationships to achieve strong practical performance on large datasets. Despite their success, the efficiency of these pruning-based strategies is predicated on the selectivity of geometric relationships. This reliance becomes a critical limitation in realistic and increasingly common scenarios where this selectivity diminishes. For instance, California's 337 hospitals serve millions of patients annually [24], while around 50 last-mile facilities deliver 2.3 million daily packages to 8.8 million residents in New York City [40]. In such cases, the expansive influence region of each facility severely undermines spatial pruning, motivating the pursuit of a robust RkNN method that does not degrade under these challenging conditions.

We identify three specific scenarios where traditional pruning-based RkNN algorithms face significant performance degradation: 1) *Small Facility Sets*: A small number of facilities leads to expanded potential influence regions for each, drastically reducing the number of candidates that can be pruned. 2) *Large User Populations*: As the cardinality of U grows, spatial pruning retains a much larger pool of candidate users, leading to a prohibitive increase in verification costs. 3) *Large k Values*: Increasing k weakens the geometric criteria for discarding candidates, as the conditions for exclusion become harder to satisfy, thereby diminishing the pruning effect.

To address these challenges, we propose a fundamental formulation of the two-dimensional (2D) RkNN problem, called RT-RkNN.

*Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819527

We model it as a three-dimensional (3D) graphics ray casting problem, where users are treated as rays and facilities are encoded as geometric primitives. This novel formulation allows the RkNN evaluation to be mapped to massively parallel ray-primitive intersection tests. Crucially, the core geometric operations in this model align perfectly with the capabilities of GPU Ray Tracing (RT) cores, which were originally designed for 3D rendering. By leveraging these specialized hardware accelerators, our approach minimizes warp divergence, exploits massive parallelism, and delivers scalable RkNN performance that remains stable even in settings where traditional spatial pruning becomes ineffective.

This paper makes the following contributions:

- We formulate 2D RkNN queries as a 3D graphics ray casting problem and establish the formal equivalence between ray-primitive occlusion and RkNN's spatial pruning relationships.
- We provide the first algorithm derived from this new formulation, which can leverage mature technologies in computer graphics and enables massively parallel execution of ray-primitive intersection tests.
- We present the application of GPU ray tracing cores to RkNN query processing, demonstrating how hardware-accelerated ray-primitive intersection tests can be effectively leveraged for efficient spatial pruning and user verification.
- Extensive experiments on real-world datasets with up to more than 23 million users demonstrate that our approach outperforms state-of-the-art baselines across diverse parameter settings. Notably, our method performs particularly well in scenarios where traditional spatial pruning becomes inefficient, such as with large k values, high user cardinality, or sparse facility distributions.

2 BACKGROUND & RELATED WORK

In this section, we first provide the formal definition of the target problem and address the scope of this paper in subsection 2.1. We review the RkNN related work in subsection 2.2, and the use of RT cores in spatial queries in subsection 2.3.

2.1 Problem Statement

RkNN queries are classified into *monochromatic RkNN queries* and *bichromatic RkNN queries* [27].

Monochromatic RkNN. Let $P = \{p_1, \dots, p_n\}$ be a set of points in a metric space. For any point $p \in P$, let $NN_k(p; P)$ denote the set of the k nearest neighbors of p within $P \setminus \{p\}$. A monochromatic RkNN query for a point $q \in P$ returns

$$RkNN(q; P) = \{p \in P \setminus \{q\} \mid q \in NN_k(p; P)\}.$$

In other words, p is an RkNN of q if q is among the k closest points to p in the same dataset.

Bichromatic RkNN. Let $F = \{f_1, \dots, f_m\}$ be a set of facilities and $U = \{u_1, \dots, u_n\}$ be a set of users, where F and U lie in the same metric space and $F \cap U = \emptyset$. For any user $u \in U$, let $NN_k(u; F)$ denote its k nearest facilities in F . A bichromatic RkNN query for a facility $f \in F$ returns

$$RkNN(f; F, U) = \{u \in U \mid f \in NN_k(u; F)\}.$$

That is, u is an RkNN of f if f is one of the k nearest facilities to u .

RkNN queries are challenging because the membership condition for a user u depends on how the distance $\text{dist}(u, f)$ compares to its distances to *all* other facilities in F . Formally, u is an RkNN of f if and only if

$$|\{a \in F \setminus \{f\} \mid \text{dist}(u, a) < \text{dist}(u, f)\}| < k.$$

This condition requires determining the rank of $\text{dist}(u, f)$ within the entire distance set $\{\text{dist}(u, a)\}_{a \in F}$. Unlike forward kNN, which only identifies the k smallest distances for each query point, the reverse condition depends on the *global ordering* of distances around every user. Consequently, RkNN queries inherently involve many distance comparisons per user, making them mathematically global and significantly more complex.

Scope of this paper. Following existing RkNN techniques that primarily target 2D spatial data that arise in location-based services [8, 50, 60, 63], we also put the focus of this paper on 2D location data. Although our approach applies to both monochromatic and bichromatic RkNN queries, we focus on the bichromatic setting in our performance evaluation for easier presentation as it represents the more general case with the monochromatic case reducible to it. The performance of our approach on monochromatic RkNN queries is only briefly discussed. Although, as mentioned above, RkNN queries have been extensively studied with various variants, we focus on the most widely used static queries in Euclidean space in this paper. Finally, given the substantially larger memory in modern machines, even our largest dataset (containing over 23 million points) occupies only about 218 MB, far below the capacity of a typical PC. We therefore assume that all data can be fully loaded into main memory. As a result, we do not evaluate disk I/O performance in this paper and instead report runtime directly.

2.2 Existing RkNN Algorithms

While early studies on reverse nearest neighbor (RNN) queries relied on substantial preprocessing to enable efficient query evaluation [19, 27, 28, 45, 47, 61], such preprocessing introduces significant overhead and lacks the flexibility required for dynamic updates in real-time services. We therefore focus on four representative preprocessing-free techniques, selected for their wide adoption and strong performance.

SIX [50], known as the first algorithm that does not need any pre-computation, uses a *regions-based* pruning strategy which partitions the space around the query point into six equal angular partitions of 60° each. For a given partition P , the distance from the query to its k -th nearest facility in that region defines a filtering threshold: any user located in P at a distance greater than the threshold cannot be an RkNN of the query facility. Figure 1a illustrates the idea that users lying in the shaded area are pruned because at least k facilities within the same partition are closer to them than the query. The remaining users are treated as candidates to be verified in the subsequent verification stage, which is performed by issuing a range query centered at the candidate with radius $\text{dist}(u, q)$, where u is the candidate user and q is the query facility. The candidate is confirmed as an RkNN result if and only if the range query contains no more than k facilities. However, since a separate range query is required for each candidate user, the verification phase leads to substantial overhead.

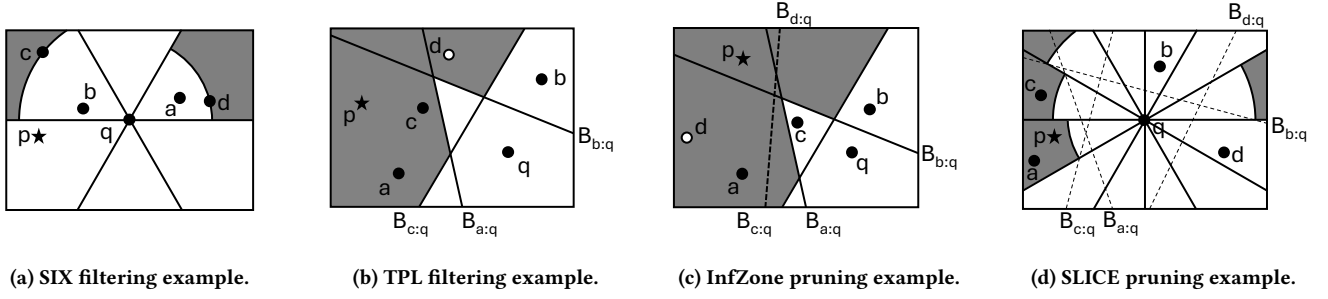


Figure 1: Illustration of SIX [50] filtering, TPL [51] filtering, InfZone [8] pruning, and SLICE [63] pruning ($k = 2$).

TPL [51] is the first algorithm which applied spatial pruning based on *half-space* pruning to Rk NN queries and inspired many follow up works [8, 60, 62], including ours. Given a query point q and a facility f , the perpendicular bisector $B_{f:q}$ divides the space into two half-spaces: the region $H_{f:q}$ that contains f and the opposite region $H_{q:f}$. Any point p lying in region $H_{f:q}$ indicates that p is closer to facility f rather than q . A point lying in at least k such half-spaces cannot regard q as one of its k nearest facilities and can therefore be pruned. The TPL algorithm iteratively accesses unpruned facilities in increasing order of their distance to q , and each accessed facility contributes a half-space that potentially enlarges the prunable region. Consider the example in Figure 1b, where the bisectors of facilities a , b , and c partition the space. When $k = 2$, the shaded region corresponds to points that fall in at least two of the half-spaces $H_{a:q}$, $H_{b:q}$, and $H_{c:q}$, and thus any point in this region can be safely filtered. Facility d , however, lies in the region that can be filtered, so it is not used to build bisectors for spatial pruning. User p lies inside the prunable region and therefore cannot be an Rk NN candidate. As TPL’s filtering phase may leave false positives like SIX, the algorithm includes a subsequent refinement stage to verify the remaining candidates.

Influence Zone (InfZone) [8] relies on the concept of the *influence zone*, which represents the region in which a user is guaranteed to be an Rk NN of the query point q . In other words, a point p is an Rk NN of q if and only if p lies inside the influence zone. To estimate the influence zone, InfZone incrementally intersects the half-spaces generated by the facilities. The construction of the influence zone, illustrated in Figure 1c, is also driven by the half-spaces of facilities relative to the query point q . As each facility is processed, the region is filtered by removing any area excluded by at least k facilities, and the remaining area is the influence zone. Facilities whose bisectors cannot affect the influence zone (like facility d in Figure 1c) are ignored. The InfZone algorithm builds the zone by tracking the intersections of unpruned bisectors. A critical insight is that the influence zone is star-shaped [43], which enables an efficient pruning strategy: a facility f is unnecessary if $\min_{v \in V} \text{dist}(f, v) > \text{dist}(v, q)$ for every convex vertex v of the current zone. However, a direct check against all $O(m^2)$ vertices, where m is the number of bisectors, is computationally heavy. To mitigate this, two inexpensive filters are applied first: (1) a facility f can be directly pruned if

$$\text{dist}(f, q) > 2 \times \max_{v \in V} \text{dist}(v, q), \quad (1)$$

and (2) a facility f cannot be pruned if

$$\text{dist}(f, q) < 2 \times \min_{p \in E} \text{dist}(p, q), \quad (2)$$

where V is the vertex set and E is the boundary of the influence zone. The expensive vertex check is only performed for facilities that do not meet either of these two conditions, which significantly reduces the overall computation cost. The verification stage in InfZone consists of a single geometric containment check. Any user located inside the influence zone is directly reported as an Rk NN, whereas any user outside it is immediately filtered out. This mechanism allows the algorithm to avoid costly distance computations and eliminates the need for a candidate examination process.

SLICE [63] enhances the filtering power of the regions-based pruning, which was first proposed in the SIX approach [50] while retaining its computational efficiency. SLICE partitions the space around the query point q into 12 regions of equal angular extent, a number determined to be optimal. Within a partition P , a facility f defines two key arcs: an upper arc, which describes the area that can be pruned by f , and a lower arc, which defines the area that is safe from f . For any partition P and facility f , the radius of two arcs can be determined in $O(1)$ time by evaluating the maximum and minimum subtended angles of f over P , which are equivalent to the distances from the intersection points between f ’s bisector $B_{f:q}$ and the two radial boundaries of P . During the pruning phase, for each partition, SLICE maintains the k -th smallest upper arc as its *bounding arc* (r_p^R). Any point p in P with $\text{dist}(p, q) > r_p^R$ is guaranteed to be pruned by at least k facilities and is filtered. As shown in Figure 1d, the shaded area is the space pruned by facilities a , b , c and d . We can see that in SLICE, each facility can potentially contribute to the spatial pruning for multiple partitions. Similar to SIX and TPL, SLICE’s filtering phase may also leave false positives; therefore, a verification phase is necessary. For the verification phase, candidate users are those lying inside their partition’s bounding arc. A facility is considered *significant* for a partition if its lower arc is smaller than the partition’s bounding arc, meaning it can potentially prune candidates. SLICE creates a sorted *significant list* of these facilities for each partition. To verify a candidate user u , the algorithm checks facilities in this sorted list, counting how many prune u . The process stops early if k pruning facilities are found, which means u can be filtered or if the lower arc of the current facility exceeds $\text{dist}(u, q)$, which means u is confirmed as an Rk NN result of q .

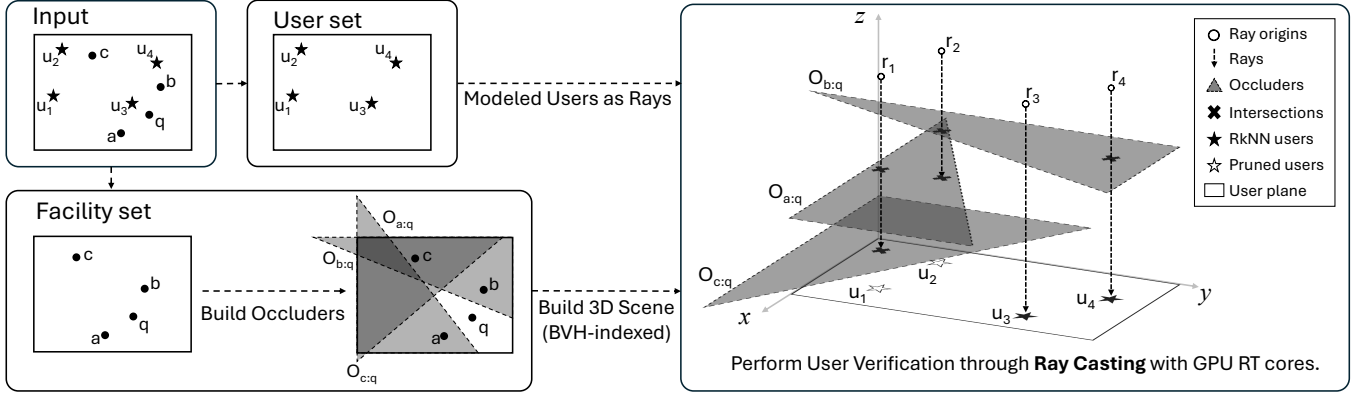


Figure 2: Overview of the RT-RkNN workflow. The input is separated into a user set and a facility set. Users are modeled as rays, while facilities are used to construct occluders based on the invalid sides of pairwise bisectors. All occluders are embedded as layered triangles in a 3D scene indexed by a BVH. During query processing, each user ray is cast perpendicularly to x, y -plane. A user is reported as an $RkNN$ result if and only if its ray intersects fewer than k occluders.

2.3 RT Cores in Spatial Queries

In recent years, RT cores have transcended graphics, emerging as a powerful primitive for accelerating database operations including indexing, scan, and query [21, 31, 48], as well as spatial and distance-based queries such as kNN [32, 38, 64], high-dimensional approximate nearest neighbor search [30], DBSCAN clustering [37], spatial joins [16], and spatial indexing [17]. By encoding spatial proximity as geometric primitives and traversing a Bounding Volume Hierarchy (BVH) [34] on dedicated hardware, RT cores sidestep the branching that degrades standard SIMT execution, reducing distance comparisons to hardware-level intersection tests and enabling concurrent evaluation of millions of user–facility relationships at low latency.

Among these tasks, RT-core-accelerated kNN is the most closely related to our problem, as both involve point-to-point distance evaluation. However, all existing kNN methods [32, 38, 64] using RT cores adopt a query-centered formulation: each point acts as a ray origin, and candidate neighbors are identified through intersection tests or distance filtering relative to that single origin of query point. While this design aligns well with the objective of kNN search, it is fundamentally incompatible with the semantics of $RkNN$, where the goal is to determine for which users a facility remains competitive against all other facilities. Consequently, query-centered formulations cannot capture the required cross-facility ranking relationships inherent in $RkNN$, thereby motivating the development of a distinct geometric formulation.

3 PROPOSED APPROACH: RT-RKNN

In this section, we describe how an $RkNN$ query can be formulated as a graphics ray casting problem, prove its correctness, and present our algorithm based on this new geometric interpretation that directly aligns with dedicated hardware ray tracing cores on modern GPUs. To offer a high-level view of the proposed method, Figure 2 summarizes the end-to-end RT-RkNN workflow, from inputs to constructing occluders and the BVH-indexed scene, and

finally performing ray-casting-based user verification on GPU RT cores.

3.1 $RkNN$ Formulation as Graphics Ray Casting

Ray casting, a fundamental operation in modern computer graphics [57], determines how rays interact with geometric primitives in a scene. A ray is emitted from an origin and travels through a 3D environment while reporting intersections with objects such as triangles. Formally, in ray casting, a ray is defined as

$$\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}, \quad (3)$$

where $\mathbf{o} \in \mathbb{R}^3$ is the ray origin, $\mathbf{d} \in \mathbb{R}^3$ is the ray direction, and $t \in [t_{\min}, t_{\max}]$ specifies the ray’s active interval. A ray intersects a geometric primitive P if and only if

$$\{\mathbf{r}(t) \mid t \in [t_{\min}, t_{\max}]\} \cap P \neq \emptyset. \quad (4)$$

When P is a triangle with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^3$, the widely used Möller–Trumbore test [36] determines an intersection by solving

$$\mathbf{r}(t) = (1 - a - b)\mathbf{v}_0 + a\mathbf{v}_1 + b\mathbf{v}_2, \quad (5)$$

for barycentric coordinates $a \geq 0, b \geq 0$, and $a + b \leq 1$. These concepts, such as rays, triangle primitives, and ray–primitive intersection tests, form the basis for the geometric constructions used in the following subsection. Although a single ray may intersect multiple primitives, real-time graphics pipelines typically limit the number of processed intersections for performance reasons [15]. This operational model naturally aligns with the structure of $RkNN$ queries.

Half-space pruning techniques for $RkNN$ rely on perpendicular bisectors between facilities, where each bisector separates space into a region in which the query facility is closer and a region in which a competing facility is closer. We reinterpret this geometric relationship directly as a graphics ray casting problem: users act as ray origins, and the invalid half-spaces induced by facility pairs are encoded as occluding geometric primitives. Under this formulation, determining whether a user belongs to the $RkNN$ result reduces to counting the number of occluder primitives intersected by their

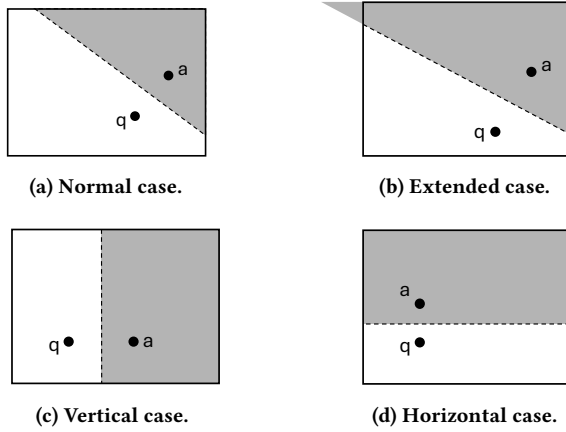


Figure 3: Four occluder construction scenarios.

corresponding ray. If a ray intersects at least k such occluders, then at least k competing facilities are closer to that user than the query facility, and the user can be pruned.

For each facility pair with respect to a query facility, the invalid side of their bisector can be represented as one or more triangular occluders. Figure 3 illustrates the four possible occluder construction scenarios: (a) The normal case, where the invalid side forms a triangle naturally; (b) The extended case, where the invalid region of a non-vertical and non-horizontal bisector does not naturally form a triangular shape because the bisector extends beyond the bounded domain, and thus the exact polygonal boundary is replaced by a single triangle that fully covers the intended invalid space; (c) The vertical bisector case; and (d) The horizontal bisector case. When the bisector is neither vertical nor horizontal, the invalid region forms a triangle. Otherwise, the invalid region becomes rectangular and can be modeled using two triangles. Each occluder is constructed by intersecting the bisector with the rectangular domain boundary, producing two intersection points. Together with one or two boundary vertices lying on the invalid side, these points form either a single triangle or two triangles, depending on the shape of the invalid region.

As illustrated in the right side of Figure 2, we embed all the occluders of corresponding facilities into a simple 3D scene to align the representation with the ray casting direction and to avoid geometric degeneracies. Users lie on a plane (e.g., $z = 0$), and rays are cast perpendicularly through the scene from the same (x, y) coordinates as the users but with a larger z value. The occluders are positioned as horizontal layers parallel to the xy -plane, each placed at a distinct height, ensuring rays traverse the occluders in a consistent front-to-back order before reaching the user plane. In this 3D embedding, each user corresponds to a single ray shot through the layered occluders. The RkNN query reduces to counting the number of occluders intersected by this ray. If the ray hits at most $k-1$ occluders, the user does not have more than k closer facilities than query q and remains as one of the query results. Conversely, if it intersects k or more occluders, the user can be pruned. Because the occluders are arranged as discrete layers, intersections occur in strictly ordered depth, enabling early termination once the hit

count reaches k . Figure 2 illustrates the example with $k = 2$: rays r_3 and r_4 intersect fewer than two occluders and are reported as RkNN results of q , while r_1 and r_2 each hit at least two occluders and the corresponding users are pruned.

To formalize the notion of *occluder*, *scene* and *ray*, we provide the formal notation used in our geometric representation.

Definition 3.1 (Occluder). Let $R \subset \mathbb{R}^2$ be a rectangular domain, and let $a, q \in R$ be two facilities with coordinates (x_a, y_a) and (x_q, y_q) , where q is the query facility and $B_{a,q}$ denotes the perpendicular bisector between a and q . If $B_{a,q}$ is neither vertical nor horizontal, among the four vertices of R we select the vertex lying on the invalid side of $B_{a,q}$ and farthest in the direction from a toward q , denoted by v . The bisector intersects the two boundary edges incident to v , producing two distinct intersection points p_1 and p_2 . If $B_{a,q}$ is vertical or horizontal, there are two farthest vertices of R on the invalid side of the bisector with equal distance to $B_{a,q}$, denoted by v_1 and v_2 . The bisector intersects the two boundary edges of space R are denoted as p_1 and p_2 . To embed the geometry into 3D space, we assign a unique height $z_{a,q} > 0$ shared by all lifted points $(v, p_1, p_2$ or $v_1, v_2, p_1, p_2)$ such that no two occluders are associated with different facility pairs sharing the same z -coordinate.

The occluder corresponding to the facility pair (a, q) is then defined as

$$O_{a,q} = \begin{cases} \Delta(v, p_1, p_2), & \text{if } x_a \neq x_q \text{ and } y_a \neq y_q, \\ \Delta(v_1, p_1, p_2) \cup \Delta(v_2, p_1, p_2), & \text{if } x_a = x_q \text{ or } y_a = y_q. \end{cases} \quad (6)$$

Definition 3.2 (Scene). Let F be the set of all facilities and let $q \in F$ be the query facility. For every facility $a \in F \setminus \{q\}$, let $O_{a,q}$ denote the occluder associated with the pair (a, q) as defined in Definition 3.1. The *scene* corresponding to query facility q is defined as the set:

$$\mathcal{T}_q = \{O_{a,q} \mid a \in F \setminus \{q\}\}. \quad (7)$$

Definition 3.3 (Ray). For a user u with its coordinates $\mathbf{u} \in \mathbb{R}^2$, as rays are cast from the same (x, y) -coordinates of users and projected perpendicular to xy -plane, we define the ray corresponding to u as:

$$\mathbf{r}_u = (\mathbf{u}, z_u) + t \cdot (0, 0, -1), \quad (8)$$

based on (3), where z_u is large enough to ensure that the ray origin is higher than all occluders $O \in \mathcal{T}_q$ and $t \in [0, z_u]$.

3.2 Theoretical Equivalence and Correctness

To demonstrate that the ray casting formulation faithfully preserves the semantics of RkNN, we define the hit count $\mathcal{H}_q(\mathbf{r}_u)$ as the number of occluders intersected by the ray $\mathbf{r}_u$:

$$\mathcal{H}_q(\mathbf{r}_u) = |\{O \in \mathcal{T}_q \mid \exists t \in [0, z_u] \text{ satisfying (4)}\}|. \quad (9)$$

Using this definition, our ray casting interpretation becomes equivalent to the classical RkNN condition:

$$u \in \text{RkNN}(q) \iff \mathcal{H}_q(\mathbf{r}_u) < k. \quad (10)$$

LEMMA 3.4 (CORRECTNESS OF RAY CASTING FORMULATION). For any facility q and user u , let $\mathbf{r}_u$ be defined as in Definition (8) and $\mathcal{H}_q(u)$ be defined as in Equation (9). Then u is a reverse k nearest neighbor of q if and only if $\mathcal{H}_q(\mathbf{r}_u) < k$, i.e., Equation (10) holds.

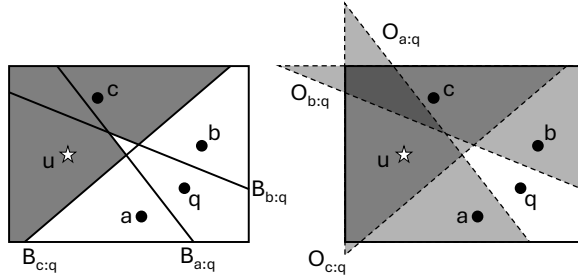


Figure 4: Equivalence between classical RkNN with ray-casting-based formulation ($k = 2$). Half-space pruning (left). Ray casting formulation (right).

PROOF. Consider the perpendicular bisector between q and any competing facility f . Consistent with concepts used in related work, the invalid side of this bisector is precisely the region in which f is closer to u than q . Whenever a ray $\mathbf{r}_u$ cast from u perpendicularly to the user plane intersects the occlusion triangle, it means u lies in a space indicating that f is strictly closer to u than q . Each intersection counted in $\mathcal{H}_q(\mathbf{r}_u)$ therefore corresponds to one distinct facility is closer than q with respect to u .

Therefore, if $\mathcal{H}_q(\mathbf{r}_u) \geq k$, then at least k facilities are closer to u than q , so q cannot be among the k nearest neighbors of u , implying $u \notin \text{RkNN}(q)$. Conversely, if $\mathcal{H}_q(u) < k$, then fewer than k facilities closer than q for u , meaning that q is within the k nearest facilities to u , and therefore $u \in \text{RkNN}(q)$. This completes the proof. $\square$

Figure 4 shows the equivalence between classical RkNN’s half-space pruning and our ray casting formulation using an example of finding RkNN of facility q with competitor facilities a , b , and c . The shaded area in the figure on the left demonstrates the pruned space under two invalid sides of bisectors, which corresponds to the regions shaded with two or more gray levels in figure on the right, which demonstrates an area covered by at least two occluders in a top-down view. Users like u that lie in these areas can be filtered directly.

3.3 Data Indexing and Proposed Algorithm

However, in our ray casting formulation, solving Möller–Trumbore intersection tests in Equation (5) for all ray-primitive pairs are impractical as the number of rays and primitives increases. To address this, we apply a well-established technique in computer graphics, bounding volume hierarchy (BVH) [34], to index the primitives, which significantly reduces the number of intersection tests. A BVH is a tree data structure that organizes geometric primitives into a hierarchy of nested bounding volumes, typically axis-aligned bounding boxes (AABBs). Each node in the tree represents an AABB that contains either a group of child AABBs or primitives. This structure allows the ray-primitive intersection test to be performed as an efficient single-path ray traversal over the BVH.

Figure 5 illustrates a BVH constructed for a 2D scene containing four distinct triangle primitives. For example, if a ray intersects primitive C, the traversal path to locate the intersection would be $B0 \rightarrow B2 \rightarrow B5 \rightarrow C$. Notably, a ray may intersect multiple primitives, so its traversal path through the scene is not necessarily

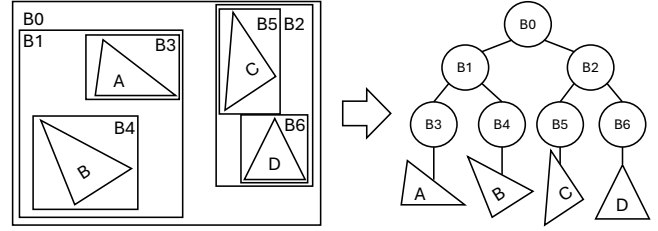


Figure 5: A BVH example (right) corresponding to a 2D scene (left) with four distinct triangle primitives (A, B, C, and D). The tree nodes, labeled B0 to B6, represent the corresponding AABBs in the 2D space. Primitives always appear as leaf nodes in the BVH.

linear. In a 3D scene, the BVH is typically built by recursively dividing the space along the longest axis, creating two subspaces and forming corresponding AABBs.

Compared with R-trees and their variants, BVHs are preferred for ray casting because their hierarchical structure is designed specifically to optimize ray-primitive intersection queries. In a BVH, the bounding volumes of sibling nodes do not overlap, which ensures that each primitive resides in exactly one leaf node and prevents redundant traversal of the same geometry. This non-overlapping organization allows rays to move through a clean and efficient hierarchy without repeated tests. In addition, BVH traversal follows a simple and predictable pattern in which the algorithm descends the hierarchy, tests bounding volumes, and prunes subtrees that cannot be intersected. These properties make BVHs highly efficient for processing the millions of rays issued in modern graphics workloads and a natural fit for ray casting applications.

Algorithm 1 summarizes our graphics ray casting approach for processing RkNN queries. The computation proceeds in two phases: *scene construction*, where occluders associated with facilities are embedded into a 3D representation; and *ray casting*, where rays representing users are issued to determine whether each user can be pruned. During scene construction (Lines 1–8), for each facility $f \in F \setminus \{q\}$, we compute the perpendicular bisector with q and encode the invalid side as occlusion triangles following Definition 3.1. To avoid constructing unnecessary occluders and to eliminate the need for a candidate verification phase after filtering like SIX [50], TPL [51] and SLICE [63], we apply InfZone-style pruning [8]: once a facility’s occluder is already fully covered by k previously constructed occluders, it is discarded because no ray can reach it. This pruning significantly reduces the number of remaining facilities, denoted $m \ll |F|$, and enables full utilization of the GPU during ray casting rather than falling back to CPU-based candidate verification. The construction of occluders requires $O(m)$, while the pruning logic contributes $O(m^2)$, dominated by intersection calculations between bisectors. A BVH is then built over the resulting scene $\mathcal{T}_q$. Unlike facility R-trees used in prior work, which can be reused across different queries, the BVH structure must be rebuilt for each query because the occlusion geometry depends on the specific query facility. While BVH construction could be amortized when the facility set is stable, similar strategies could equally benefit TPL and InfZone via cached bisector sets; for fairness, we apply

Algorithm 1 RT-RkNN Query for Facility q .

Input: Facility q ; user set U ; facility set F ; pruning threshold k .

Output: RkNN users set of facility q : $\mathcal{R}_q$.

```
1: Scene Construction: Construct occluders.
2:  $\mathcal{B}_q \leftarrow \text{InfZone\_pruning}(q, F, k)$ .
3:  $\mathcal{T}_q \leftarrow \emptyset$ 
4: for each facility  $f \in \mathcal{B}_q$  do
5:   Build occluder  $O_{f:q}$  based on Definition 3.1.
6:    $\mathcal{T}_q \leftarrow \mathcal{T}_q \cup O_{f:q}$ 
7: end for
8: Build a BVH over  $\mathcal{T}_q$ .

9: Ray Casting: Count intersections along user rays.
10:  $\mathcal{R}_q \leftarrow \emptyset$ 
11: for each user  $u \in U$  do
12:   Construct a ray  $\mathbf{r}_u$  from  $u$  based on Definition 8.
13:    $c \leftarrow 0$ 
14:   Traverse the BVH of  $\mathcal{T}_q$ .
15:   for each occluder  $O$  intersected by  $\mathbf{r}_u$  do
16:      $c \leftarrow c + 1$ 
17:     if  $c \geq k$  then
18:       break {Early exit: user is prunable}
19:     end if
20:   end for
21:   if  $c < k$  then
22:      $\mathcal{R}_q \leftarrow \mathcal{R}_q \cup \{u\}$ 
23:   end if
24: end for
25: return  $\mathcal{R}_q$ 
```

no such amortization to any method. As BVH construction typically requires $O(m \log m)$ [53–55], the overall complexity of the scene construction phase becomes $O(m^2 + m \log m)$.

After the scene is built, ray casting begins (Lines 9–24). Each user u emits a ray $\mathbf{r}_u$ into the scene, and intersections are evaluated accordingly. Early termination is applied: once k intersections are detected, the user is pruned (Line 18). Since a single intersection search under BVH traversal has an expected cost of $O(\log m)$ [52], processing a ray with at most k allowable intersections results in a total expected complexity of $O(k \log m)$ for determining whether the user belongs to the RkNN result. It is worth noting that, unlike prior work that indexes users using an R-tree or its variants, our approach does not index users at all. While this increases the asymptotic cost of the ray casting phase to $O(k|U| \log m)$, it enables full exploitation of parallelism, allowing all rays to be processed independently, especially by GPUs associated with RT cores and yielding substantial performance benefits in practice.

3.4 Implementing RT-RkNN using RT cores

To fully leverage hardware acceleration, we implement the proposed formulation algorithm using RT cores.

RT cores were originally designed to accelerate ray tracing in computer graphics, a more complex form of ray casting that incorporates effects such as reflection, refraction, and shadows. Before it was introduced, ray tracing was difficult to scale on GPUs because

Algorithm 2 RT-RkNN Implementation using OptiX.

Input: The constructed scene $\mathcal{T}_q$, scene height m .

Output: User verification result $isRkNN$.

```
1: procedure Ray Generation
2:    $t_{\min}, t_{\max} \leftarrow \{0, m + 1\}$ 
3:    $u \leftarrow \text{optixGetLaunchIndex}()$ 
4:    $\mathbf{o} \leftarrow (u_x, u_y, m + 1)$ 
5:    $\mathbf{d} \leftarrow (0, 0, -1)$ 
6:    $c \leftarrow 0$ 
7:    $isRkNN \leftarrow \text{TRUE}$ 
8:    $\text{optixTrace}(\mathcal{T}_q, \mathbf{o}, \mathbf{d}, t_{\min}, t_{\max}, \text{payloads}(c, isRkNN))$ 
9: end procedure

10: procedure Any-hit
11:    $c \leftarrow c + 1$ 
12:   if  $c < k$  then
13:      $\text{optixIgnoreIntersection}()$ 
14:   else
15:      $isRkNN \leftarrow \text{FALSE}$ 
16:      $\text{optixTerminateRay}()$ 
17:   end if
18: end procedure
```

different rays typically produce different numbers of ray-primitive intersections, resulting in irregular memory access and divergent control flow that conflict with the SIMT execution model. RT cores address these challenges by providing hardware-accelerated ray-triangle and ray-AABB intersection tests during BVH traversal and by scheduling similar intersection tasks together to reduce divergence and improve throughput. After formulating RkNN queries as ray casting, our algorithm can be implemented on RT cores with relatively modest programming effort using their corresponding APIs.

There are several RT APIs, including Nvidia OptiX [42], AMD HIP-RT [33], DirectX Raytracing [35] and Vulkan [26]. All of these APIs share a similar programming model. In this paper, we choose Nvidia OptiX for our implementation due to its cutting-edge performance and strong compatibility with CUDA. It is worth noting that while the implementation in this paper targets Nvidia GPUs' RT cores, porting the algorithm to AMD GPUs, whose ray tracing hardware is functionally similar, is straightforward. To use OptiX, developers follow a predefined workflow:

- (1) Create a GPU context.
- (2) Build a BVH.
- (3) Create a program pipeline.
- (4) Build the Shader Binding Table (SBT).
- (5) Launch rays.

A context is first created to associate OptiX with a specific CUDA device. In step (2), a BVH corresponding to the scene is constructed. The program pipeline in step (3) consists of several programmable stages:

- *Ray generation*: The entry point to the pipeline, defines the ray by origin point $\mathbf{o}$ and direction $\mathbf{d}$ based on equation (3).

- *Intersection*: Implements a ray-primitive intersection test, invoked during BVH traversal. Programmers do not need to implement it when using the default hardware intersection test.
- *Any-hit*: Called when a new, potentially closest intersection of a ray is found. Not necessary when only the closest intersection is counted.
- *Closest-hit*: Called when the ray finds the closest intersection.
- *Miss*: Called after confirming a ray does not hit any primitive in the scene.

OptiX uses a single ray programming model [42], where each ray executes this pipeline independently. The SBT constructed in step (4) links primitives with their associated programs and serves as the binding layer between rays and scene behavior. Finally, rays are launched and will explore the scene based on the program pipeline defined before.

As shown in Algorithm 2, our OptiX implementation of ray casting stage is straightforward because the formulated RkNN computation aligns naturally with ray tracing. Only the Ray Generation and Any-hit programs are required. We use Any-hit instead of Closest-hit because we are interested in counting up to k intersections per ray rather than identifying the nearest one. The Miss program can be omitted, as no additional processing is required when a ray misses all occluders. The Intersection program is left empty to leverage the default hardware-accelerated triangle intersection routine provided by the RT cores. OptiX will trace the ray just generated by calling `optixTrace()`.

User’s coordinates are embedded in the rays which can be obtained by `optixGetLaunchIndex()`. The height of the ray origins is set to $m + 1$ to ensure it is higher than all occluders in the scene. During traversal, once c reaches k , the ray is terminated early using `optixTerminateRay()`. Otherwise, intermediate intersections invoke `optixIgnoreIntersection()`, allowing traversal to continue. Under OptiX’s single ray programming model, the Ray Generation program launches one ray per user and stores both the intersection count c and the user verification result *isRkNN* via OptiX payloads variables.

4 EVALUATION

This section outlines the experimental setup and reports performance with detailed analysis.

4.1 Experimental Setup

System: We compare our RT-RkNN approach against three baseline algorithms introduced in subsection 2.2: TPL, InfZone, and SLICE. In the figures, InfZone is abbreviated as *INF*, and our RT-RkNN method is abbreviated as *RT* for clarity. All algorithms were implemented from scratch, using shared common routines for comparable operations. To emphasize query-time efficiency, all baselines use R*-trees rather than standard R-trees, providing a stronger query-time foundation for the competing methods. The only exception is the BVH, which is constructed exclusively for our RT-RkNN method. Implementations are written in C++11 and compiled using NVCC 12.4 with -O2 flag, and OptiX 7.7 enabled where applicable. Experiments were conducted on a workstation equipped with an AMD EPYC 7453 28-core CPU, 512 GB of system memory, and an Nvidia RTX A6000 GPU with 48 GB of GPU memory, running Ubuntu 22.04.5 LTS.

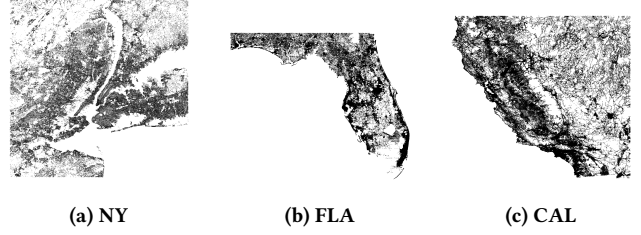


Figure 6: Visualization of datasets.

Table 1: Six real-world road-network datasets.

Symbol	Description	Number of Points
USA	Full USA	23,947,347
CTR	Central USA	14,081,816
E	Eastern USA	3,598,623
CAL	California and Nevada	1,890,815
FLA	Florida	1,070,376
NY	New York City	264,346

Table 2: Amortized user indexing cost in dataset USA

Algorithms	Operation	Runtime (s)
All baselines	R*-tree construction	147.241
RT-RkNN (ours)	Plain GPU transfer	0.010251

Datasets: We use six real-world road-network datasets covering New York City (NY), Florida (FLA), California and Nevada (CAL), Eastern USA (E), Central USA (CTR), and the full USA (USA). These datasets contain between 264,346 and 23,947,347 points and were obtained from the DIMACS repository [13]. The spatial distributions of representative datasets are visualized in Figure 6, and detailed statistics are provided in Table 1.

Evaluation Configurations: Following prior work, we vary k from 1 to 25 (default $k = 10$), with additional values up to 200 to examine large- k scalability. We use two facility settings: a *default* setting of 1,000 randomly selected facilities (aligning with existing baselines) and a *sparse* setting of 100 facilities (representing sparse facility distributions). For both settings, except for points selected as facilities, all remaining points are used as users. Results are averaged over 100 random queries for the sparse setting and 1,000 queries for settings with 1,000 or more facilities, ensuring statistical robustness and consistency. All algorithms (including RT-RkNN) follow a two-stage execution model whose stage names and internal logic differ across methods; we adopt the convention of [62], calling the first stage *filtering* and the second *verification*, to enable direct runtime-breakdown comparisons across algorithms.

4.2 Amortized Operation Costs

In all baseline algorithms, R*-trees built over facilities and users are amortized across queries and thus excluded from the performance

scaling analysis. RT-RkNN likewise benefits from amortization: although it constructs a facility R*-tree to support InfZone-style pruning during scene construction, it does not require a user R*-tree. Instead, the user set is uploaded once to GPU memory and reused throughout the workload. The same applies to OptiX context and program pipeline creation, which is amortizable and trivial. Although amortizable costs do not contribute to per-query scalability, they remain relevant when comparing overall preprocessing burden across algorithms, especially in workloads with short query sequences. As shown in Table 2, RT-RkNN incurs significantly less indexing overhead than the baselines while retaining the advantages of amortized preprocessing, making it comparatively lightweight in preprocessing cost.

4.3 Impact of Varying k Settings

Figure 7 and Figure 8 analyze the impact of k on runtime under the sparse facility and default facility settings, respectively. Although only a subset of datasets is shown, all datasets exhibit the same overall trend. Under the sparse facility setting, RT-RkNN consistently outperforms all baseline algorithms except when $k = 1$, where all methods complete in approximately 3 ms. Under the default facility density, where baseline algorithms benefit from stronger spatial pruning opportunities, RT-RkNN still outperforms TPL and InfZone across all tested values of k . Even when compared with SLICE, which is known to be more resilient to increases in k , RT-RkNN begins to outperform it once k reaches 10–20. For the extremely large k setting, we only compare RT-RkNN with SLICE on our largest dataset USA. Figure 9 illustrates that for extremely large k scenarios, our RT-RkNN outperforms the state-of-the-art algorithm SLICE in all cases. Remarkably, our RT-RkNN achieves up to 16.4 $\times$ speedup over SLICE, 34.6 $\times$ over InfZone, and as much as 54.7 $\times$ over TPL when $|F| = 100$ and $k = 25$ on the CAL dataset. This result reflects the issue discussed in subsection 2.2: as k increases, pruning-based methods lose pruning effectiveness, resulting in larger candidate sets and substantially more verification work. In contrast, for RT-RkNN, increasing k simply allows a ray to continue traversal rather than terminating early. The algorithm, therefore, degrades gracefully, requiring only additional intersection counting while still benefiting from hardware acceleration. Consequently, the impact of larger k values on runtime remains modest compared with traditional approaches.

4.4 Impact of Varying Data Sizes

We study the impact of data size on each algorithm using all 6 datasets, which span a wide range of scales, with the largest dataset (USA) being about 100 times larger than the smallest (NY), under both the sparse facility and default facility settings. In a sparse facility setting, as shown in Figure 10a, our RT-RkNN outperforms all baselines across all datasets. Its runtime increases only slowly as data size grows, demonstrating the effectiveness of our algorithm with massively parallel ray-primitive intersection processing on GPU RT cores. In contrast, the performance of baseline algorithms degrades substantially once the dataset exceeds approximately 10 million points (datasets E and USA), reflecting their limited scalability under large input sizes. Figure 10b shows that under the

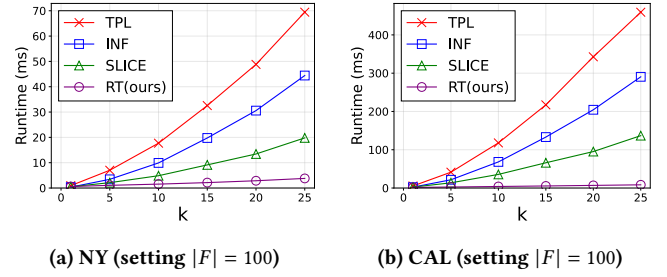


Figure 7: Impact of varying k on runtime in sparse facility setting.

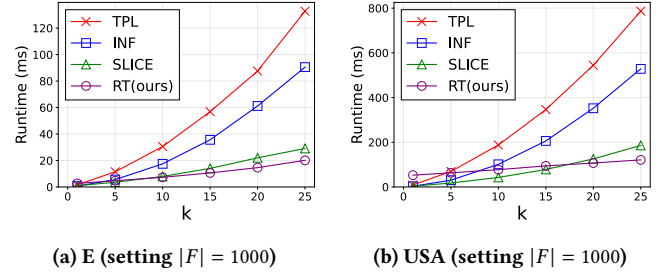


Figure 8: Impact of varying k settings on runtime using the default facility setting.

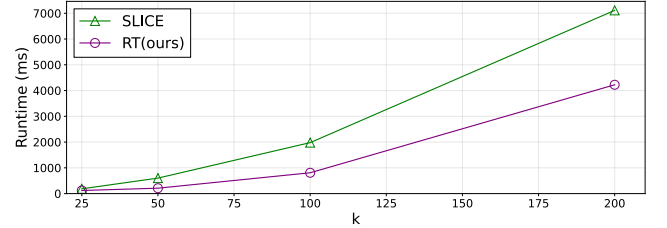
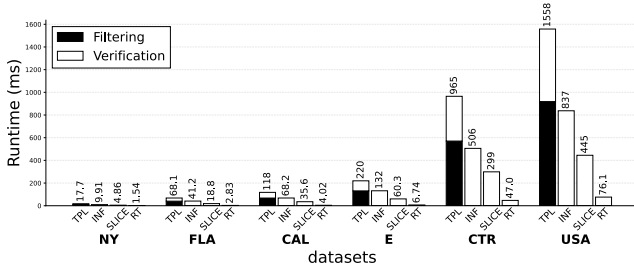


Figure 9: Impact of large k on runtime in default facility setting ($|F| = 1000$) of USA dataset.

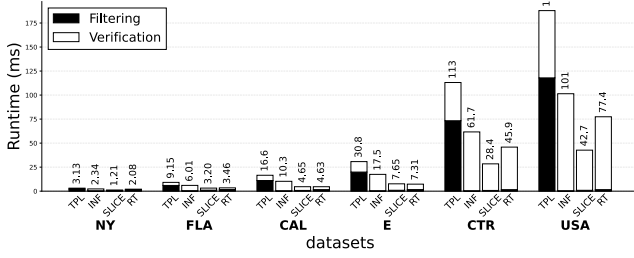
default facility setting, where spatial pruning remains highly effective, RT-RkNN does not outperform SLICE. This is primarily due to the additional overhead introduced by data transfer between main memory and GPU memory.

4.5 Impact of Varying Facility Cardinality

In this section, we fix the number of users to 1 million for CAL and 10 million for USA, and vary the facility cardinality $|F|$ to study its impact on performance. From Figure 11, we observe that RT-RkNN maintains nearly constant performance across all datasets, regardless of the number of facilities. In contrast, the baseline algorithms improve significantly as $|F|$ grows, benefiting from effective spatial pruning. From Figure 12, we observe that in both cases, the verification phase of the baseline algorithms becomes significantly faster as the facility cardinality increases, while RT-RkNN maintains a stable verification cost. The flat performance curve of RT-RkNN occurs because all computation that can benefit from GPU parallelism is already highly optimized, while the remaining overhead involved



(a) Sparse facility setting.



(b) Default facility setting.

Figure 10: Impact of data size on runtime in sparse ($|F| = 100$) and default ($|F| = 1000$) facility setting.

by the GPU RT cores pipeline, is inherently difficult to parallelize and therefore does not improve with larger facility sets. As a result, RT-RkNN remains largely unaffected by increases in $|F|$, whereas pruning-based methods gain efficiency from having more facilities. For *monochromatic RkNN queries*, where the query point and candidate points originate from the same set, spatial relationships tend to be more structured, providing a more favorable environment for spatial pruning as shown here. Under these conditions, RT-RkNN does not surpass the SLICE algorithm, which benefits greatly from strong pruning effectiveness.

4.6 Impact of Varying User Cardinality

In this section, we fix the number of facilities using both the sparse and default facility settings for the USA dataset, and vary the user cardinality $|U|$ to examine its impact on performance. All algorithms perform well when the number of users is small. However, performance degrades for every method as the user population becomes large. In the sparse facility setting, as shown in Figure 13a, RT-RkNN maintains strong performance even with very large user populations of up to 10^7 points, outperforming the baseline algorithms by a substantial margin. In the default facility setting, as illustrated in Figure 13b, RT-RkNN still does not surpass SLICE. As shown in Figure 14, in the sparse setting, RT-RkNN achieves the best performance due to its highly efficient verification phase, which benefits from massive GPU parallelization. However, this advantage does not carry over to the default facility setting, where SLICE becomes faster but RT-RkNN experiences performance degradation, especially when $|U|$ reaches 10^7 due to the additional overhead introduced by GPU data transfer. Nonetheless, although RT-RkNN does not outperform SLICE in every scenario, the performance

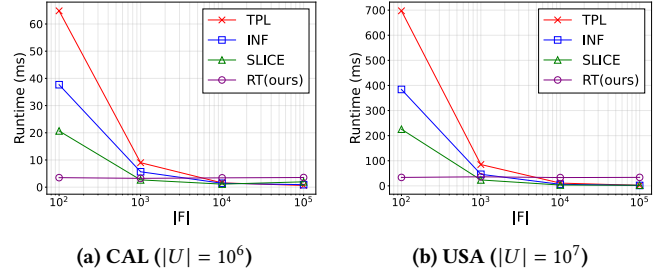
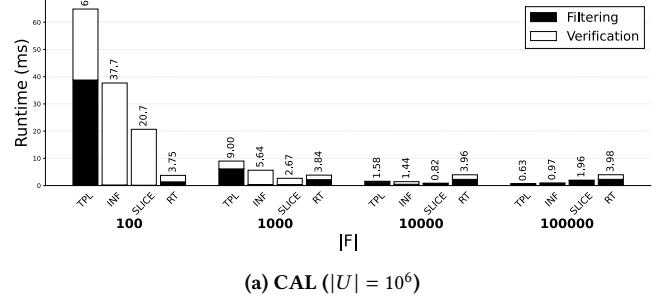
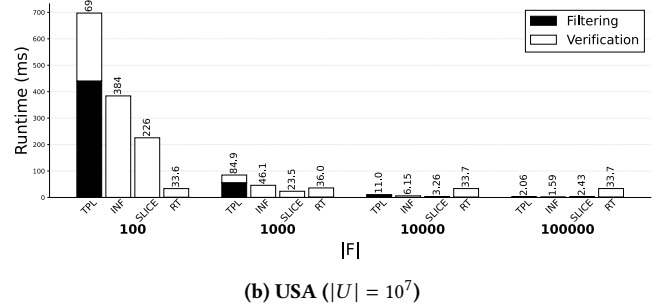


Figure 11: Total runtime under varying cardinality for datasets CAL ($|U| = 10^6$) and USA ($|U| = 10^7$).



(a) CAL ($|U| = 10^6$)



(b) USA ($|U| = 10^7$)

Figure 12: Breakdown of filtering and verification time under varying facility cardinality for datasets CAL ($|U| = 10^6$) and USA ($|U| = 10^7$).

gap in the default facility setting is relatively small compared to the substantial advantages RT-RkNN achieves in the sparse facility setting.

4.7 Runtime Breakdown Analysis

In this section, we provide a breakdown analysis of RT-RkNN to examine its advantages and limitations. According to the OptiX workflow, the *Scene Construction* stage can be divided into two components: occluder construction and BVH construction. Although the size of the Shader Binding Table (SBT) also depends on the scene configuration, we do not include the SBT construction time in our analysis, because RT-RkNN only requires hit or miss information and does not rely on per-primitive attributes, making SBT construction trivial. In practice, SBT setup takes approximately 0.062 ms even for the largest scene where $|F| = 10^7$, which is negligible compared to the other costs. Then, *Ray Casting* stage is divided into ray tracing

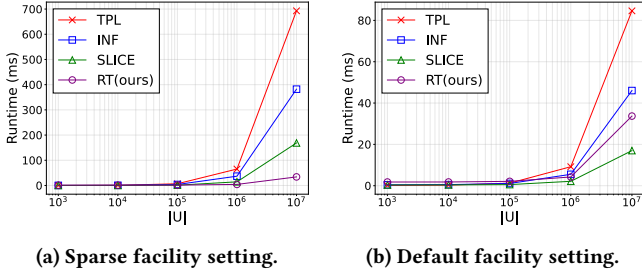


Figure 13: Total runtime under varying facility cardinality for sparse and default facility setting of dataset USA.

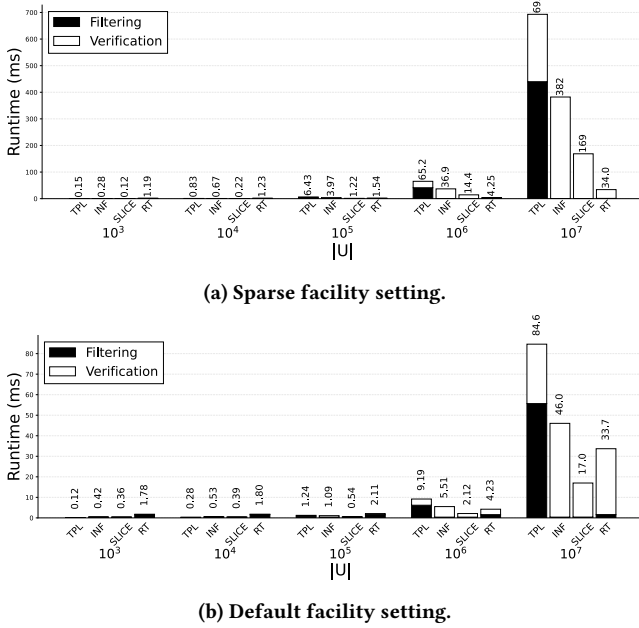


Figure 14: Breakdown of filtering and verification time under varying user cardinality for the USA dataset in sparse and default facility settings.

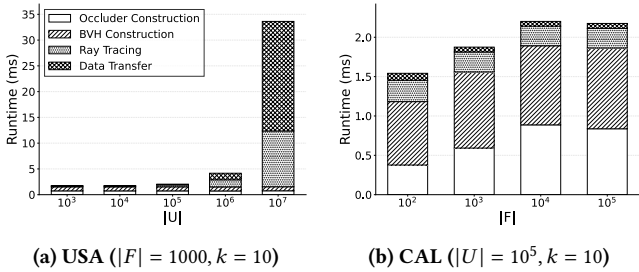


Figure 15: Runtime breakdown analysis under configurations with fixed $|U|$ and fixed $|F|$.

and data transfer, which copies the ray tracing result back to the main memory from the GPU.

Table 3: Average occluder counts of different occluder construction strategies in different facility cardinality.

	$ F = 10^2$	$ F = 10^3$	$ F = 10^4$
InfZone-style	37.27	46.34	48.44
Conservative	49.54	83.04	205.34
Non-pruning	99	999	9999

We fix $|F| = 10^3$ and vary $|U|$ from 10^3 to 10^7 to evaluate how user cardinality affects the performance of RT-RkNN on the USA dataset. Since the facility set is fixed, the constructed scene remains identical across all experiments. As shown in Figure 15a, RT cores sustain high ray tracing efficiency until the user population reaches approximately one million. However, when $|U|$ increases to ten million, ray tracing time and data transfer time begin to dominate. This is expected because 10^7 rays exceed the number that the GPU can process concurrently, causing some rays to wait while others are executed. It is important to note that data transfer constitutes pure overhead, influenced solely by user cardinality. When $|U| = 10^7$, data transfer accounts for more than half of the total runtime. This overhead becomes particularly significant in dense facility settings, where baseline algorithms achieve efficient RkNN computation with small k . This explains why RT-RkNN cannot outperform SLICE on very large datasets under default facility densities. We then fix $|U| = 10^5$ and vary $|F|$ from 10^2 to 10^5 to evaluate how facility cardinality affects the performance of RT-RkNN on the CAL dataset. The user cardinality is restricted to 10^5 to ensure that the *Ray Casting* stage does not dominate the overall runtime. As $|F|$ increases, the occluder construction time initially grows and reaches its peak around $|F| = 10^4$. Beyond this point, the construction time stabilizes or even slightly decreases because InfZone-style pruning removes most unnecessary occluders. This also explains why BVH construction time and ray tracing time remain largely unaffected by changes in $|F|$.

4.8 Impact of Different Occluder Counts

In this section, we investigate how the number of occluders affects the performance of RT-RkNN. As introduced earlier, we apply InfZone-style pruning to avoid constructing unnecessary occluders. However, in practice, we observe that when the facility set is small, constructing all occluders without pruning can actually be faster due to simpler control flow and better data locality. Motivated by this observation, we conduct supplementary experiments comparing three occluder construction strategies:

- *InfZone-style pruning*: The original pruning strategy introduced by InfZone, which aggressively removes as many unnecessary occluders as possible, with $O(m^2)$ complexity.
- *Conservative pruning*: InfZone-style pruning is applied only for the first few occluders (e.g., the first 20). Beyond that, a lightweight conservative test based on Equation (1) is used. Although not optimal, this approach still prunes most unnecessary occluders while substantially reducing pruning overhead.
- *Non-pruning*: As the name suggests, this strategy constructs all occluders without any pruning.

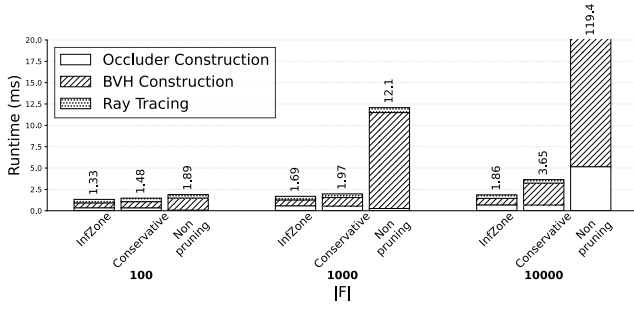


Figure 16: Impact of occluder counts on runtime under the default New York facility configuration.

The average occluder count produced by each pruning strategy under different facility cardinalities on the NY dataset is reported in Table 3, and the corresponding performance results are shown in Figure 16. Since our goal is to examine performance variations attributable specifically to the number of occluders, Figure 16 includes only the components directly affected: *Occluder Construction*, *BVH Construction*, and *Ray Tracing*.

We observe that *Ray Tracing* is not strongly affected by changes in the occluder count, except for the non-pruning strategy when $|F| = 9999$, which strongly complicates the scene for a ray to traverse (did not show in figure for clarity of other settings). This is because the NY dataset has a relatively small user cardinality, allowing rays to be processed with high parallel efficiency. The *No-pruning* strategy constructs occluders very quickly. However, the resulting large number of occluders significantly slows down subsequent stages, ultimately making it the slowest among all strategies even in a sparse facility setting. For *Conservative pruning*, although it removes approximately 98% of unnecessary occluders when $|F| = 10^4$, the remaining occluders still constitute roughly four times the number produced by *InfZone-style pruning*. While *Conservative pruning* is slightly faster than *InfZone-style pruning* in the *Occluder Construction* stage, this advantage is offset during BVH construction, resulting in inferior overall performance.

Consequently, *InfZone-style pruning* emerges as the most effective strategy for *Scene Construction*, providing the shortest total runtime and lower GPU memory usage.

4.9 Performance Evaluation w/o RT Cores

Since no publicly available GPU-based RkNN algorithms exist, we implement a GPU baseline, termed *InfZone-GPU*, which directly offloads *InfZone*’s verification phase to the GPU without using RT cores. We choose *InfZone* because it is the only baseline that produces no false positives, thereby eliminating the need for additional user verification and allowing most computation to remain on the GPU. We also include the original *InfZone* algorithm in our comparison to demonstrate the computational advantage provided by GPU execution.

On one hand, from Figure 17, we observe that *InfZone-GPU* outperforms the original CPU-based *InfZone* across all datasets, demonstrating the GPU’s substantial computing power even when the verification phase contains significant control divergence and

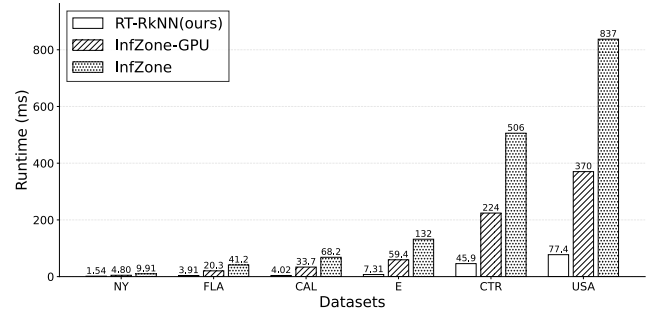


Figure 17: Performance comparison between RT-RkNN, GPU-based *InfZone* without RT cores, and CPU-based *InfZone* in the sparse facility setting of the NY dataset.

irregular memory accesses that are poorly aligned with the SIMD execution model.

On the other hand, RT-RkNN consistently outperforms *InfZone-GPU* across all datasets. As data size increases, the performance gap widens: *InfZone-GPU* degrades rapidly due to the growing control divergence across users during verification, whereas RT-RkNN maintains stable scaling behavior. This is because RT-RkNN leverages dedicated hardware intersection testing in RT cores, which eliminates the need for branch-heavy verification logic and drastically reduces control divergence. These results demonstrate that our ray casting formulation, which enables algorithm fit for RT cores hardware acceleration, is essential for achieving high performance in GPU-based RkNN computation.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we presented RT-RkNN, which formulates RkNN as graphics ray casting by modeling users as rays and facilities as occluders, establishing a formal equivalence between spatial pruning and ray-primitive intersection. Building on this, we design the first algorithm directly exploiting GPU RT cores for hardware-accelerated intersection testing and BVH traversal. Experiments on real-world datasets with up to more than 23 million points show that RT-RkNN significantly outperforms state-of-the-art algorithms, especially where traditional pruning becomes ineffective: sparse facility distributions, large user populations, and large k . RT-RkNN also eliminates user indexing, cutting preprocessing overhead while enabling massive parallelism across all user rays.

For future work, we plan to explore four directions: (1) adapting the approach for dynamic and continuous RkNN queries where facilities or users change over time; (2) investigating batched query processing to amortize scene construction costs across multiple query facilities; (3) exploring hybrid strategies that dynamically select between RT-RkNN and traditional pruning based on data characteristics; and (4) implementing and evaluate this formulation on other RT hardware.

ACKNOWLEDGMENTS

This work was supported in part by JSPS KAKENHI Grant Number JP24K20782.

REFERENCES

- [1] Elke Achtert, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Andreas Züfle. 2009. Reverse k-nearest neighbor search in dynamic and general metric databases. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) (EDBT '09). Association for Computing Machinery, New York, NY, USA, 886–897. <https://doi.org/10.1145/1516360.1516462>
- [2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, USA) (SIGMOD '90). Association for Computing Machinery, New York, NY, USA, 322–331. <https://doi.org/10.1145/93597.98741>
- [3] Rimantas Benetis, S. Jensen, Gytis Karundėdinauskas, and Simonas unde-finedalėnis. 2006. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal* 15, 3 (Sept. 2006), 229–249. <https://doi.org/10.1007/s00778-005-0166-4>
- [4] Thomas Bernecker, Tobias Emrich, Hans-Peter Kriegel, Matthias Renz, Stefan Zankl, and Andreas Züfle. 2011. Efficient probabilistic reverse nearest neighbor query processing on uncertain data. *Proc. VLDB Endow.* 4, 10 (July 2011), 669–680. <https://doi.org/10.14778/2021017.2021024>
- [5] Avory Bryant and Krzysztof Cios. 2018. RNN-DBSCAN: A Density-Based Clustering Algorithm Using Reverse Nearest Neighbor Density Estimates. *IEEE Transactions on Knowledge and Data Engineering* 30, 6 (2018), 1109–1121. <https://doi.org/10.1109/TKDE.2017.2787640>
- [6] S. Cabello, J.M. Díaz-Báñez, S. Langerman, C. Seara, and I. Ventura. 2010. Facility location problems in the plane based on reverse nearest neighbor queries. *European Journal of Operational Research* 202, 1 (2010), 99–106. <https://doi.org/10.1016/j.ejor.2009.04.021>
- [7] Muhammad Aamir Cheema, Xuemin Lin, Wei Wang, Wenjie Zhang, and Jian Pei. 2010. Probabilistic Reverse Nearest Neighbor Queries on Uncertain Data. *IEEE Transactions on Knowledge and Data Engineering* 22, 4 (2010), 550–564. <https://doi.org/10.1109/TKDE.2009.108>
- [8] Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2011. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *2011 IEEE 27th International Conference on Data Engineering*. 577–588. <https://doi.org/10.1109/ICDE.2011.5767904>
- [9] Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, Wei Wang, and Wenjie Zhang. 2009. Lazy updates: an efficient technique to continuously monitoring reverse kNN. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1138–1149. <https://doi.org/10.14778/1687627.1687755>
- [10] Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2012. Efficiently processing snapshot and continuous reverse k nearest neighbors queries. *The VLDB Journal* 21, 5 (2012), 703–728.
- [11] Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Xuefei Li. 2012. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal* 21, 1 (2012), 69–95.
- [12] Farhana Choudhury, J Shane Culpepper, Timos Sellis, and Xin Cao. 2016. Maximizing bichromatic reverse spatial and textual k nearest neighbor queries. In *42nd International Conference on Very Large Data Bases*. VLDB Endowment Inc., 456–467.
- [13] DIMACS. 2006. 9th DIMACS Implementation Challenge - Shortest Paths. <https://www.diag.uniroma1.it/~challenge9/download.shtml>. Accessed: 2025-11-24.
- [14] Tobias Emrich, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, Naixin Xu, and Andreas Züfle. 2010. Reverse k-Nearest Neighbor monitoring on mobile objects. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems* (San Jose, California) (GIS '10). Association for Computing Machinery, New York, NY, USA, 494–497. <https://doi.org/10.1145/1869790.1869870>
- [15] Epic Games. 2021. Ray Tracing Features Settings. https://dev.epicgames.com/documentation/en-us/unreal-engine/ray-tracing-features-settings?application_version=4.27. Accessed 2025-11-21.
- [16] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) (ICS '24). Association for Computing Machinery, New York, NY, USA, 124–136. <https://doi.org/10.1145/3650200.3656610>
- [17] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2025. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) (PPoPP '25). Association for Computing Machinery, New York, NY, USA, 396–411. <https://doi.org/10.1145/3710848.3710850>
- [18] Parisa Ghaemi, Kaveh Shahabi, John P. Wilson, and Farnoush Banaei-Kashani. 2012. Continuous maximal reverse nearest neighbor query on spatial networks. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems* (Redondo Beach, California) (SIGSPATIAL '12). Association for Computing Machinery, New York, NY, USA, 61–70. <https://doi.org/10.1145/2424321.2424330>
- [19] Yusuke Gotoh and Chiori Okubo. 2016. A Searching Method for Bichromatic Reverse k-Nearest Neighbor with Network Voronoi Diagram. In *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media* (Singapore, Singapore) (MoMM '16). Association for Computing Machinery, New York, NY, USA, 71–78. <https://doi.org/10.1145/3007120.3007133>
- [20] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) (SIGMOD '84). Association for Computing Machinery, New York, NY, USA, 47–57. <https://doi.org/10.1145/602259.602266>
- [21] Justus Henneberg and Felix Schuhknecht. 2023. RTInDeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4268–4281.
- [22] Lihua Hu, Hongkai Liu, Jifu Zhang, and Aiqin Liu. 2021. KR-DBSCAN: A density-based clustering algorithm based on reverse nearest neighbor and influence space. *Expert Systems with Applications* 186 (2021), 115763.
- [23] Pengfei Jin, Lu Chen, Yunjun Gao, Xueqin Chang, Zhanyu Liu, Shu Shen, and Christian S Jensen. 2023. Maximizing the influence of bichromatic reverse k nearest neighbors in geo-social networks. *World Wide Web* 26, 4 (2023), 1567–1598.
- [24] Jen Joyn. 2022. *California Hospitals Almanac – 2022 Edition*. Technical Report. California Health Care Foundation. <https://www.chcf.org/resource/california-hospitals-almanac/>. Accessed: 2025-11-19.
- [25] James M. Kang, Mohamed F. Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. 2007. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. In *2007 IEEE 23rd International Conference on Data Engineering*. 806–815. <https://doi.org/10.1109/ICDE.2007.367926>
- [26] Khronos Group. 2020. Vulkan Ray Tracing Extensions. <https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release>. Accessed 2025-11-21.
- [27] Flip Korn and S. Muthukrishnan. 2000. Influence sets based on reverse nearest neighbor queries. *SIGMOD Rec.* 29, 2 (May 2000), 201–212. <https://doi.org/10.1145/335191.335415>
- [28] Yang Li, Mingyuan Bai, Qingfeng Guan, Zi Ming, Xun Liang, Gang Liu, and Junbin Gao. 2023. CSD-RkNN: reverse k nearest neighbors queries with conic section discriminances. *International Journal of Geographical Information Science* 37, 10 (2023), 2175–2204.
- [29] King-Ip Lin, M. Nolen, and Congjun Yang. 2003. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. In *Seventh International Database Engineering and Applications Symposium, 2003. Proceedings*. 290–297. <https://doi.org/10.1109/IDEAS.2003.1214938>
- [30] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 549–565. <https://doi.org/10.1145/3620665.3640360>
- [31] 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).
- [32] 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* (Kyoto, Japan) (ICS '24). Association for Computing Machinery, New York, NY, USA, 14–25. <https://doi.org/10.1145/3650200.3656601>
- [33] Daniel Meister, Paritosh Kulkarni, Aaryaman Vasishta, and Takahiro Harada. 2024. HIPRT: A Ray Tracing Framework in HIP. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 44 (Aug. 2024), 18 pages. <https://doi.org/10.1145/3675378>
- [34] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiri Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 683–712.
- [35] Microsoft DirectX Team. 2025. DirectX at GDC 2025. <https://devblogs.microsoft.com/directx/directx-at-gdc-2025/>. Accessed 2025-11-21.
- [36] Tomas Möller and Ben Trumbore. 2005. Fast, minimum storage ray/triangle intersection (SIGGRAPH '05). Association for Computing Machinery, New York, NY, USA, 7–es. <https://doi.org/10.1145/1198555.1198746>
- [37] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973. <https://doi.org/10.1109/IPDPS54959.2023.00100>
- [38] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNs Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) (ICS '23). Association for Computing Machinery, New York, NY, USA, 289–300. <https://doi.org/10.1145/3577193.3593738>
- [39] Michael Nwogugu. 2006. Site selection in the US retailing industry. *Applied mathematics and computation* 182, 2 (2006), 1725–1734.

- [40] Office of the New York City Comptroller. 2025. *Fast Shipping. Slow Justice: Traffic, Worker, and Climate Hazards in Last Mile Delivery*. Technical Report. Office of the New York City Comptroller. <https://comptroller.nyc.gov/reports/fast-shipping-slow-justice/>
- [41] Xiao Pan, Shili Nie, Haibo Hu, Philip S. Yu, and Jingfeng Guo. 2022. Reverse Nearest Neighbor Search in Semantic Trajectories for Location-Based Services. *IEEE Transactions on Services Computing* 15, 2 (2022), 986–999. <https://doi.org/10.1109/TSC.2020.2968309>
- [42] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. 29, 4, Article 66 (July 2010), 13 pages. <https://doi.org/10.1145/1778765.1778803>
- [43] Franco P Preparata and Michael I Shamos. 2012. *Computational geometry: an introduction*. Springer Science & Business Media.
- [44] Milos Radovanovic, Alexandros Nanopoulos, and Mirjana Ivanovic. 2010. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research* 11, sept (2010), 2487–2531.
- [45] Maytham Safar, Dariush Ibrahim, and David Tanar. 2009. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia systems* 15, 5 (2009), 295–308.
- [46] Shuo Shang, Bo Yuan, Ke Deng, Kexin Xie, and Xiaofang Zhou. 2011. Finding the most accessible locations: reverse path nearest neighbor query in road networks. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Chicago, Illinois) (GIS '11). Association for Computing Machinery, New York, NY, USA, 181–190. <https://doi.org/10.1145/2093973.2093999>
- [47] Mehdi Sharifzadeh and Cyrus Shahabi. 2010. VoR-tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 1231–1242. <https://doi.org/10.14778/1920841.1920994>
- [48] 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.
- [49] Anbang Song, Ziqiang Yu, Wei Liu, Yating Xu, and Mingjin Tao. 2025. BRkNN-light: Batch Processing of Reverse k-Nearest Neighbor Queries for Moving Objects on Road Networks. In *Proceedings of the 19th International Symposium on Spatial and Temporal Data (SSTD '25)*. Association for Computing Machinery, New York, NY, USA, 80–89. <https://doi.org/10.1145/3748777.3748791>
- [50] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. 2000. Reverse nearest neighbor queries for dynamic databases.. In *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, Vol. 20. 0.
- [51] Yufei Tao, Dimitris Papadias, and Xiang Lian. 2004. Reverse kNN search in arbitrary dimensionality. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) (VLDB '04). VLDB Endowment, 744–755.
- [52] K. Vaidyanathan, S. Woop, and C. Benthin. 2022. Wide BVH traversal with a short stack (HPG '19). Eurographics Association, Goslar, DEU, 15–19. <https://doi.org/10.2312/hpg.20191190>
- [53] Ingo Wald. 2007. On fast Construction of SAH-based Bounding Volume Hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*. 33–40. <https://doi.org/10.1109/RT.2007.4342588>
- [54] Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. 26, 1 (Jan. 2007), 6–es. <https://doi.org/10.1145/1189762.1206075>
- [55] Ingo Wald and Vlastimil Havran. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in $O(N \log N)$. In *2006 IEEE Symposium on Interactive Ray Tracing*. 61–69. <https://doi.org/10.1109/RT.2006.280216>
- [56] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Timos Sellis, and Gao Cong. 2018. Reverse k Nearest Neighbor Search over Trajectories. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2018), 757–771. <https://doi.org/10.1109/TKDE.2017.2776268>
- [57] Turner Whitted. 2005. An improved illumination model for shaded display. In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California) (SIGGRAPH '05). Association for Computing Machinery, New York, NY, USA, 4–es. <https://doi.org/10.1145/1198555.1198743>
- [58] Raymond Chi-Wing Wong, M. Tamer Özsu, Philip S. Yu, Ada Wai-Chee Fu, and Lian Liu. 2009. Efficient method for maximizing bichromatic reverse nearest neighbor. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1126–1137. <https://doi.org/10.14778/1687627.1687754>
- [59] Wei Wu, Fei Yang, Chee Yong Chan, and Kian-Lee Tan. 2008. Continuous Reverse k-Nearest-Neighbor Monitoring. In *The Ninth International Conference on Mobile Data Management (mdm 2008)*. 132–139. <https://doi.org/10.1109/MDM.2008.31>
- [60] Wei Wu, Fei Yang, Chee-Yong Chan, and Kian-Lee Tan. 2008. FINCH: evaluating reverse k-Nearest-Neighbor queries on location data. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 1056–1067. <https://doi.org/10.14778/1453856.1453970>
- [61] Congyun Yang and King-Ip Lin. 2001. An index structure for efficient reverse nearest neighbor queries. In *Proceedings 17th International Conference on Data Engineering*. 485–492. <https://doi.org/10.1109/ICDE.2001.914862>
- [62] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. 2015. Reverse k nearest neighbors query processing: experiments and analysis. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 605–616. <https://doi.org/10.14778/2735479.2735492>
- [63] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Ying Zhang. 2014. SLICE: Reviving regions-based pruning for reverse k nearest neighbors queries. In *2014 IEEE 30th International Conference on Data Engineering*. 760–771. <https://doi.org/10.1109/ICDE.2014.6816698>
- [64] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) (PPoPP '22). Association for Computing Machinery, New York, NY, USA, 76–89. <https://doi.org/10.1145/3503221.3508409>