Proximity Queries on Point Clouds using Rapid Construction Path Oracle

Anonymous
Anonymous
Anonymous

Anonymous
Anonymous
Anonymous

ABSTRACT

The prevalence of computer graphics technology boosts the developments of point clouds in recent years, and researchers started to utilize their advantages over terrain surfaces (represented by *Triangular Irregular Networks*, i.e., *TIN*s) in proximity queries, including the *shortest path query*, the *k-Nearest Neighbor* (*kNN*) *query*, and the *range query*. Since (1) all existing on-the-fly and oracle-based shortest path query algorithms on a *TIN* are very expensive, (2) all existing on-the-fly shortest path query algorithms on a point cloud are still not efficient, and (3) there are no oracle-based shortest path query algorithms on a point cloud, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the shortest path query for a set of *Points-Of-Interests* (*POIs*) on the point cloud, which has a good performance (in terms of the oracle construction time, oracle size, and shortest path query time) due to the concise information about the pairwise shortest paths between any pair of POIs stored in the oracle. Our oracle can be easily adapted to answering the shortest path query for any points on the point cloud if POIs are not given as input, and also achieve a good performance. Then, we propose algorithms for answering the $(1+\epsilon)$-approximate *kNN* and range query with the assistance of our oracle. Our experimental results show that when POIs are given (resp. not given) as input, our oracle is up to 390 times, 30 times, and 6 times (resp. 500 times, 140 times, and 50 times) better than the best-known oracle on a *TIN* in terms of the oracle construction time, oracle size, and shortest path query time, respectively. Our algorithms for the other two proximity queries are both up to 100 times faster than the best-known algorithms.

ACM Reference Format:
Anonymous and Anonymous. 2023. Proximity Queries on Point Clouds using Rapid Construction Path Oracle. In *Proceedings of 2024 International Conference on Management of Data (SIGMOD '24)*. ACM, New York, NY, USA, 33 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Conducting proximity queries, including (1) the *shortest path query*, (2) the *k-Nearest Neighbor* (*kNN*) *query* [54], and (3) the *range query* [46], on a 3D surface is a topic of widespread interest in both industry and academia [26, 61]. The shortest path query is the most fundamental type of the proximity query. In industry, numerous companies and applications, such as Google Earth [2] and a 3D computer game Cyberpunk 2077 [4], utilize the shortest path passing on a 3D surface (such as Earth) for route planning. In academia, the shortest path query on a 3D model is a prevalent research topic in the field of databases [20, 32, 33, 42, 58, 59, 62, 63]. There are different representations of a 3D surface, including a terrain surface represented by a *Triangular Irregular Network* (*TIN*) and a point cloud. While performing the shortest path query on a *TIN* has been extensively studied, answering the shortest path query on a point cloud is an emerging topic. For example, Tesla uses the shortest path passing on point clouds of the driving environment for autonomous driving [13, 19, 41, 45], and Metaverse uses the shortest path passing on point clouds of objects such as mountains to help users reach the destination faster in Virtual Reality [39, 40]. Applications of the other two proximity queries include rover path planning [15] and military tactical analysis [36].

Point cloud and *TIN*: (1) A point cloud is represented by a set of 3D *points* in space. Figure 1 (a) shows a satellite map of Mount Rainier [50] (a national park in the USA) in an area of 20km × 20km, and Figure 1 (b) shows the point cloud with 63 points of Mount Rainier. Given a point cloud, we create a *conceptual graph* of the point cloud, such that its *vertices* consist of the points in the point cloud, and its *edges* consist of a set of edges between each vertex and its 8 neighbour vertices in the 2D plane. Figure 1 (c) shows a conceptual graph of a point cloud. (2) A *TIN* contains a set of *faces* each of which is denoted by a triangle. Each face consists of three line segments called *edges* connected with each other at three *vertices*. The gray surface in Figure 1 (d) is a *TIN* of Mount Rainier, which consists of vertices, edges, and faces. We focus on three paths: (1) the path passing on (a conceptual graph of) a point cloud in Figure 1 (b) and (c), (2) the *surface path* [33] passing on (the faces of) a *TIN* in Figure 1 (d), and (3) the *network path* [33] passing on (the edges of a) a *TIN* in Figure 1 (e).

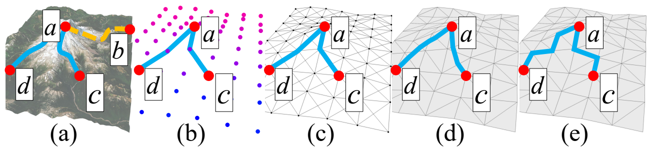


Figure 1: (a) A satellite map, (b) paths passing on a point cloud, (c) a conceptual graph of a point cloud, (d) surface paths passing on a *TIN*, and (e) network paths passing on a *TIN*

1.1 Motivation

1.1.1 ***Advantages of point cloud.*** (1) Points clouds have four advantages compared with *TIN*s.

(i) *More direct access to point cloud data.* We can use an iPhone 12/13 Pro LiDAR scanner to scan an object and generate a point cloud in 10s [57], or can use a satellite to obtain the elevation of a region in an area of 1km^2 and generate a point cloud in 144s ≈ 2.4 min [47]. But, in order to obtain a *TIN* of an object, typically, researchers need to transform a point cloud to a *TIN* [31]. Our experimental results show that it needs 210s ≈ 3.5 min to transform a point cloud with 25M points to a *TIN*.

(ii) *Lower memory consumption of a point cloud.* We only store the point information of a point cloud, but we need to store the vertex, edge, and face information of a *TIN*. Our experimental results show that storing a point cloud with 25M points needs 390MB, but storing a *TIN* generated by this point cloud needs 1.7GB.

(iii) *Faster shortest path query time on a point cloud.* After we transfer a point cloud to a *TIN*, calculating the shortest path passing on the point cloud is faster than calculating the shortest surface or network path passing on this *TIN*, since a *TIN* is more complicated than a point cloud. In addition, calculating the *shortest surface path* passing on a *TIN* is even slower since the search space is larger. Our experimental results show that calculating the shortest path passing on a point cloud with 2.5M points takes 3s, but calculating the shortest surface (resp. network) path passing on a *TIN* constructed by the point cloud takes 580s ≈ 10 min (resp. 17s).

(iv) *Small distance error of the shortest path passing on a point cloud.* In Figure 1 (b) and (d), the shortest path passing on a point cloud is similar to the shortest surface path passing on a *TIN*. But, in Figure 1 (d) and (e), the shortest surface path and the shortest network path passing on a *TIN* are very different. Because in Figure 2 (a), for the shortest path passing on a point cloud, a point q is connected with 8 neighbor points, i.e., 7 blue points and 1 red point s. But, in Figure 2 (b), for the shortest network path passing on a *TIN*, a vertex q is connected with only 6 blue neighbour vertices. Our experimental results show that given the shortest surface path passing on a *TIN*, the length of the shortest path passing on a point cloud is only 1.008 times larger than that of the given path, but the length of the shortest network path passing on a *TIN* is 1.1 times larger than that of the given path.

(2) Although calculating the shortest path passing on a point cloud can be regarded as on a conceptual graph of the point cloud, point clouds have two advantages compared with graphs, i.e., (i) there is no method to directly obtain a graph of an object, and (ii) we need to store the vertex and edge information of a graph. They are similar to (i) and (ii) in point (1). Our experimental results show that storing a point cloud with 25M points needs 390MB, but storing a graph generated by this point cloud needs 980MB.

1.1.2 ***Usages of POIs.*** (1) Given a set of *Points-Of-Interests* (*POIs*) on a point cloud or a *TIN*, conducting proximity queries between *pairs of POIs* on the point cloud or the *TIN*, i.e., *POI-to-POI* (*P2P*) *query*, is important. For example, POIs can be reference points used in measuring similarities between two different 3D objects [35, 55], and POIs can be residential locations used in studying migration patterns of the wildness animals [23, 43]. (2) If POIs are not given as input, we need to conduct proximity queries between (i) *pairs of any points* on the point cloud, i.e., *Any points-to-Any points* (*A2A*) *query*, or (ii) *pairs of arbitrary points* on the *TIN*, i.e., *ARbitrary points-to-ARbitrary points* (*AR2AR*) *query*. The AR2AR query on a *TIN* is more general than the A2A query on a point cloud since a point may lie on the face of a *TIN*.

1.1.3 ***Usage of oracles.*** Although answering the proximity query on a point cloud *on-the-fly* is fast, if we can pre-compute the pairwise P2P or A2A shortest paths by means of indexing (called an *oracle*) on a point cloud, then we can use the oracle to answer the proximity query more efficiently (the time taken to pre-compute the oracle is called the *oracle construction time*, the space complexity of the oracle is called the *oracle size*, and the time taken to return the shortest path is called the *shortest path query time*).

1.1.4 ***Example.*** We conducted a case study on an evacuation simulation in Mount Rainier due to snowfall [51]. In Figure 1 (a), we need to find the shortest paths (in blue and yellow lines) from one of the viewing platforms (e.g., POI a) on the mountain to its k-nearest hotels (e.g., POIs b to d) due to the limited capacity of each hotel. In Figure 1 (b) - (e), c and d are the k-nearest hotels to a where $k = 2$. Our experimental results show that we can construct an oracle on a point cloud with 5M points and 500 POIs (250 viewing platforms and 250 hotels) in 400s ≈ 6.6 min, but it needs 77,200s ≈ 21.4 hours on a *TIN* (constructed based on the same point cloud) to construct the same oracle. In addition, we can return the shortest paths from each viewing platform to its k-nearest hotels in 6s with the oracle, but it needs 4,400s ≈ 1.2 hours on a point cloud without the oracle. These show the usefulness of performing proximity queries on point clouds using oracles in real-life applications.

1.2 Challenges

1.2.1 ***Inefficiency of on-the-fly algorithms.*** All existing algorithms [48, 56, 64] for conducting proximity queries on a point cloud *on-the-fly* are very slow, since they (1) first construct a *TIN* using the given point cloud in $O(N)$ time, where N is the number of points in the point cloud, and (2) then calculate the shortest path passing on this *TIN*. For calculating the shortest surface path passing on a *TIN*, the best-known on-the-fly *exact* [16] and *approximate* [32] algorithm run in $O(N^2)$ and $O((N+N')\log(N+N'))$ time, respectively, where N' is the number of additional points introduced for bound guarantee. For calculating the shortest network path passing on a *TIN*, the best-known on-the-fly *approximate* algorithm [33] runs in $O(N \log N)$ time. Our experimental results show (1) algorithm [16] needs 290,000s ≈ 3.4 days, (2) algorithm [32] needs 161,000s ≈ 1.9 day, and (3) algorithm [33] needs 15,000s ≈ 4.2 hours to perform the *kNN* query for all 500 objects on a *TIN* with 0.5M vertices, which is very slow.

1.2.2 ***Non-existence of oracles.*** No existing oracle can answer proximity queries on a point cloud. The best-known oracle [58, 59] for the P2P query and the best-known oracle [30] for the AR2AR query only pre-compute the pairwise shortest surface paths passing on a *TIN*. Although we can first construct a *TIN* using the point cloud, then use works [30, 58, 59] for point cloud oracle construction, their oracle construction time is very large due to the *bad criterion for algorithm earlier termination*. This is because although they use the *Single-Source All-Destination* (*SSAD*) algorithm [16, 32, 33], i.e., a Dijkstra-based algorithm [24], to pre-compute the shortest surface path passing on the *TIN* from each POI (or point) to other POIs (or points), and provide a criterion to *terminate it earlier*, its criterion is very loose, and different POIs (or points) have the *same* earlier termination criterion. In our experiment, even after the *SSAD* algorithm has visited most of the POIs (or points), their earlier termination criterion are still not reached. After constructing a *TIN* using the given point cloud, the oracle construction time is $O(nN^2 + c_1 n)$ for the oracle [58, 59], and is $O(c_2 N^2)$ for the oracle [30], respectively, where n is the number of POIs on the point cloud and c_1, c_2 are constants depending on the point cloud ($c_1 \in [35, 80]$ on a point cloud with 2.5M points on average, $c_2 \in [75, 154]$ on a point cloud with 100k points on average). In our experiment, the oracle construction time for the oracle [58, 59] is 78,000s ≈ 21.7 hours on a point cloud with 2.5M points and 500 POIs, and for the oracle [30] is 50,000s ≈ 13.9 hours on a point cloud with 100k points.

1.3 Our Oracle and Proximity Query Algorithms

We propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the shortest path query for a set of POIs on a point cloud called Rapid Construction path Oracle, i.e., *RC-Oracle*, which has a good performance in terms of the oracle construction time, oracle size, and shortest path query time compared with the best-known oracle [58, 59] for the P2P query due to the concise information about the pairwise shortest paths between any pair of POIs stored in the oracle, where ϵ is a non-negative real user parameter called an *error parameter*. *RC-Oracle* can be easily adapted to answer the A2A query on the point cloud if POIs are not given as input, and also achieve a good performance compared with the best-known oracle [30] for the A2A query. Based on *RC-Oracle*, we develop efficient $(1+\epsilon)$-approximate proximity query algorithms. We introduce the key idea of the small oracle construction time of *RC-Oracle*.

(1) **Rapid point cloud on-the-fly shortest path query algorithm**: When constructing *RC-Oracle*, we propose algorithm *Fast on-the-Fly shortest path query*, i.e., *FastFly*, which is a Dijkstra-based algorithm [24] returning its calculated shortest path passing on a point cloud. It can significantly reduce the algorithm's running time, since computing the shortest path passing on a *TIN* is expensive.

(2) **Rapid oracle construction**: When constructing *RC-Oracle* for the P2P (resp. A2A) query, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, to calculate the shortest path passing on the point cloud from for each POI (resp. point) to other POIs (resp. points) *simultaneously*, and set *different* earlier termination criterion for different POIs (resp. points), i.e., this criterion is tight.

1.4 Contributions and Organization

We summarize our major contributions as follows.

(1) We propose *RC-Oracle*, which is the first oracle that efficiently answers the shortest path queries on a point cloud. We also propose algorithm *FastFly* used for constructing *RC-Oracle*, and develop efficient proximity query algorithms using *RC-Oracle*.

(2) We provide theoretical analysis on (i) the oracle construction time, oracle size, shortest path query time, and error bound of *RC-Oracle*, (ii) the shortest path query time and error bound of algorithm *FastFly*, (iii) the *kNN* query time, range query time, and error bound for proximity queries, and (iv) the distance relationships of the shortest path passing on a point cloud or a *TIN*.

(3) *RC-Oracle* performs much better than the best-known oracle [58, 59] for the P2P query and the best-known oracle [30] for the A2A query on a point cloud in terms of the oracle construction time, oracle size, and shortest path query time. The *kNN* and range query time with the assistance of *RC-Oracle* also performs much better than the best-known oracle. Our experimental results show that (i) for the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time and oracle size for *RC-Oracle* is 200s ≈ 3.2 min and 50MB, but is 78,000s ≈ 21.7 hours and 1.5GB for the best-known oracle [58, 59], (ii) under the same setting, the *kNN* and range query time of all 500 POIs for *RC-Oracle* are both 12.5s, but the best-known oracle [58, 59] needs 150s, and the best-known on-the-fly approximate shortest surface path query algorithm [32] on the *TIN* (constructed by the given cloud) needs 161,000s ≈ 1.9 days, and (iii) for the A2A query on a point cloud with 100k points and 5000 objects, the oracle construction time, oracle size, and *kNN* query time for *RC-Oracle* is 100s ≈ 1.6 min, 150M, and 0.25s, but is 50,000s ≈ 13.9 hours, 21GB, and 12.5s for the best-known oracle [30]. *RC-Oracle* also supports real-time responses, i.e., it can construct the oracle in 0.4s and answer the *kNN* query and range query in both 7ms on a point cloud with 10k points and 250 POIs.

The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 covers the related work. Section 4 presents the methodology. Section 5 covers the empirical studies and Section 6 concludes the paper.

2 PROBLEM DEFINITION

2.1 Notations and Definitions

2.1.1 ***Point cloud.*** Given a set of points, we let C be a point cloud of these points, and N be the number of points in C (i.e., $N = |C|$). Each point $p \in C$ has three coordinate values, denoted by x_p, y_p and z_p. We let x_{max} and x_{min} (resp. y_{max}, and y_{min}) be the maximum and minimum x (resp. y) coordinate value for all points in C. We define $L_x = x_{max} - x_{min}$ (resp. $L_y = y_{max} - y_{min}$) to be the side length of C along x-axis (resp. y-axis), and $L = \max(L_x, L_y)$. In Figure 2 (a), $L_x = L_y = 4$. In this paper, the point cloud C that we considered is a grid-based point cloud [12, 25], because a grid-based 3D object, e.g., a grid-based point cloud [12, 25] and a grid-based *TIN* [21, 42, 54, 58, 59], is commonly adopted in many papers. Given a point p in C, we define $N(p)$ to be a set of neighbor points of p, which denotes the closest top, bottom, left, right, top-left, top-right, bottom-left, and bottom-right points of p in the xy coordinate 2D plane. Figure 2 (a) shows an example of a point cloud C. In this figure, given a green point q, $N(q)$ is denoted as 7 blue points and 1 red point s. Let P be a set of POIs each of which is a point on the point cloud and n be the size of P (i.e., $n = |P|$). Since a POI can only be a point on C, $n \leq N$.

Given a pair of points p and p' in 3D space, we define $d_E(p, p')$ to be the Euclidean distance between p and p'. We define G to be a conceptual graph of C. Let $G.V$ and $G.E$ be the set of vertices and edges of G. Each point in C is denoted by a vertex in $G.V$. For each point $q \in C$, $G.E$ consists of a set of edges between q and $q' \in N(q)$. Figure 2 (c) shows a conceptual graph of a point cloud. Given a pair of points s and t on C, (1) let $\Pi^*(s, t|C) = (s = q_1, q_2, \ldots, q_l = t)$,

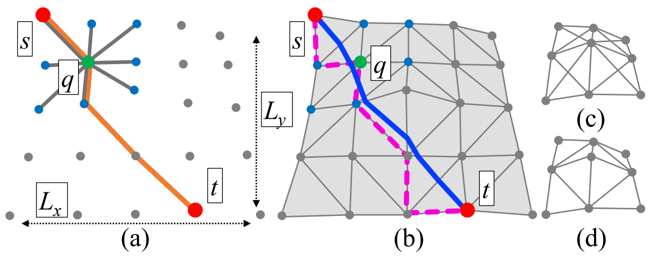


Figure 2: (a) Orange $\Pi^*(s,t|C)$, (b) blue $\Pi^*(s,t|T)$ and pink $\Pi_N(s,t|T)$, (c) a conceptual graph of a point cloud, and (d) a conceptual graph of a *TIN*

with $l \geq 2$, be the exact shortest path passing on (G of) C between s and t, such that (i) each q_i is a vertex in $G.V$, (ii) each (q_i, q_{i+1}) is an edge in $G.E$, and (iii) $\sum_{i=1}^{l-1} d_E(q_i, q_{i+1})$ is the minimum, and (2) let $\Pi(s,t|C)$ be the shortest path of returned by *RC-Oracle*. G is stored as a data structure and C can be cleared from the memory, so we do not need to construct G every time when we need to calculate the shortest path passing on C. Figure 2 (a) shows an example of $\Pi^*(s,t|C)$ in orange line. We define $|\cdot|$ to be the length of a path (e.g., $|\Pi^*(s,t|C)|$ is the length of $\Pi^*(s,t|C)$). *RC-Oracle* guarantees that $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any s and t in P. The differences between the points and POIs are that (1) we use points (from C) to construct G, and then calculate the shortest path passing on G, but (2) we use POIs as sources and destinations in *RC-Oracle*, and POIs are a subset of points.

We can easily extend our problem to the non-grid-based point cloud. Given a point p in a non-grid-based point cloud, we re-define $N(p)$ to be a set of neighbor points of p such that the Euclidean distance between p and all points in this non-grid-based point cloud is smaller than a user-defined parameter. Then, we can use the exact shortest path definition in the grid-based point cloud to calculate the exact shortest path passing on the non-grid-based point cloud.

2.1.2 ***TIN****.* Let T be a *TIN* triangulated [49] by the points in C. Similar to G, we define G' to be a conceptual graph of T. Let $G'.V$ and $G'.E$ be the set of vertices and edges of G', where each vertex in T is denoted by a vertex in $G'.V$, and each edge in T is denoted by an edge in $G'.E$. Figure 2 (d) shows a conceptual graph of a *TIN*. Given a pair of points s and t on C, (1) let $\Pi^*(s,t|T) = (s = m_1, m_2, \ldots, m_l = t)$ be the exact shortest surface path passing on T between s and t, such that (i) each m_i is a point along an edge of T, and (ii) $\sum_{i=1}^{l-1} d_E(m_i, m_{i+1})$ is the minimum, (2) let $\Pi_N(s,t|T) = (s = n_1, n_2, \ldots, n_l = t)$ be the shortest network path passing on (G' of) T between s and t, such that (i) each n_i is a vertex in $G'.V$, (ii) each (n_i, n_{i+1}) is an edge in $G'.E$, and (iii) $\sum_{i=1}^{l-1} d_E(n_i, n_{i+1})$ is the minimum. G' is also stored as a data structure and T can be cleared from the memory. Let θ be the minimum interior angle of a triangle in T. Figure 2 (b) shows an example of $\Pi^*(s,t|T)$ in blue line and $\Pi_N(s,t|T)$ in pink line. Table 1 shows a notation table.

2.1.3 ***Proximity queries****.* (1) In the *shortest path query*, given a source s and a destination t on a point cloud, it answers the shortest path passing on the point cloud between s and t. (2 & 3) In the *kNN query* (resp. the *range query*), given a user parameter k (resp. a range value r), a set of objects, and a query object q on the point cloud, it answers all the shortest paths passing on the point cloud from q to the k nearest objects of q (resp. from q to the objects whose distance to q are at most r) using the shortest path query.

Table 1: Summary of frequent used notations

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
$d_E(p,p')$	The Euclidean distance between point p and p'
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
T	The *TIN* constructed by C
$\Pi^*(s,t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s,t\|C)\|$	The length of $\Pi^*(s,t\|C)$
$\Pi(s,t\|C)$	The shortest path passing on C between s and t returned by *RC-Oracle*
$\Pi^*(s,t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_N(s,t\|T)$	The shortest network path passing on T between s and t

For the two types of proximity queries on a point cloud, i.e., *P2P query* and *A2A query*, by creating POIs which has the same coordinate values as all points in the point cloud, the A2A query can be regarded as one form of the P2P query. Furthermore, in the P2P query, there is no need to consider the case when a new POI is added or removed. In the case when a POI is added, we can create an oracle to answer the A2A query, which implies we have considered all possible POIs to be added. In the case when a POI is removed, we can still use the original oracle.

2.2 Problem

The problem is to (1) design an efficient $(1+\epsilon)$-approximate shortest path oracle on a point cloud with the state-of-the-art performance in terms of the oracle construction time, oracle size, and shortest path query time, and (2) use this oracle for efficiently answering the $(1+\epsilon)$-approximate *kNN* and range query.

3 RELATED WORK

3.1 On-the-fly Algorithms

All existing *on-the-fly* proximity query algorithms [48, 56, 64] on a point cloud are very slow. Given a point cloud, they first triangulate it into a *TIN* [49] in $O(N)$ time, then they calculate the shortest path passing on this *TIN*. To the best of our knowledge, no algorithm can answer proximity queries on a point cloud directly without converting it to a *TIN*. There are two types of *TIN* shortest path query algorithms, i.e., (1) the *shortest surface path* [16, 32, 38, 44, 60] and (2) the *shortest network path* [33] query algorithms.

3.1.1 ***Shortest surface path query algorithms****.* There are two more sub-types. (1) *Exact algorithms*: Algorithm [44] (resp. algorithm [60]) uses continuous Dijkstra (resp. checking window) algorithm to calculate the result in $O(N^2 \log N)$ (resp. $O(N^2 \log N)$) time, and the best-known exact shortest surface path query algorithm *C̲hen and H̲an*, i.e., algorithm *CH* [16] (as recognized by work [32, 33, 54, 61]) unfolds the 3D *TIN* into a 2D *TIN*, and then connects the source and destination using a line segment on this 2D *TIN* to calculate the result in $O(N^2)$ time. But, algorithm *CH* (without constructing a *TIN* first) cannot be directly adapted on the point cloud, because there is no face to be unfolded in a point cloud. (2) *Approximate algorithms*: All algorithms [32, 38] place discrete points (i.e., Steiner points) on edges of a *TIN*, and then construct a graph using these Steiner points together with the original vertices to calculate the result. The best-known $(1+\epsilon)$-approximate shortest surface path query algorithm, i.e., algorithm *Kaul* [32] (as recognized by work [58, 59]) runs in $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ time, where l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of the *TIN*, and θ is the minimum inner angle of any face in the *TIN*. If we let the path pass on the conceptual graph of the point cloud, algorithm *Kaul* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

3.1.2 ***Shortest network path query algorithm.*** Since the shortest network path does not cross the faces of a *TIN*, it is an approximate path. The best-known approximate shortest network path query algorithm *Dijkstra*, i.e., algorithm *Dijk* [33] runs in $O(N\log N)$ time. If we let the path pass on the conceptual graph of the point cloud, algorithm *Dijk* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

Drawbacks of the on-the-fly algorithms: Although we can pre-process the point cloud and store the generated *TIN* as a data structure, all these algorithms are still time-consuming. Since the time for calculating the shortest path passing on a *TIN* is much larger than (i.e., 10^2 to 10^5 times larger than) the time for converting a point cloud to a *TIN*. Thus, the latter time can be neglected. We denote algorithm (1) *CH-Adapt*, (2) *Kaul-Adapt*, and (3) *Dijk-Adapt*, to be the adapted algorithm [48, 56, 64], which first constructs a *TIN* using the given point cloud (i.e., we store the *TIN* as a data structure and clear the given point cloud from the memory), and then use algorithm (1) *CH* [16] to compute the exact shortest surface path, (2) *Kaul* [32] to compute the approximate shortest surface path, and (3) *Dijk* [33] to compute the approximate shortest network path passing on the *TIN*. But, since we regard the shortest path passing on a point cloud as the exact shortest path, algorithm *CH-Adapt*, *Kaul-Adapt*, and *Dijk-Adapt* return the approximate shortest path passing on a point cloud. Our experimental results show algorithm *CH-Adapt*, *Kaul-Adapt*, and *Dijk-Adapt* first needs to convert a point cloud with 0.5M points to a *TIN* in 4.2s, then perform the *kNN* query for all 2500 objects on this *TIN* in 290,000s ≈ 3.2 days, 90,000s ≈ 1 day, and 15,000s ≈ 4.2 hours, respectively.

3.2 Oracles for the shortest path query

No existing oracle can answer the shortest path query between pairs of POIs or pairs of any points on a point cloud. But, *Space Efficient Oracle* (*SE-Oracle*) [58, 59] (resp. *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [30]) uses an oracle to index the approximate pairwise P2P (resp. AR2AR) shortest surface paths passing on a *TIN*. We denote (1) *SE-Oracle-Adapt* to be the adapted oracle of *SE-Oracle* [58, 59] that first constructs a *TIN* from a point cloud (i.e., we store the *TIN* as a data structure and clear the given point cloud from the memory), then uses *SE-Oracle* on this *TIN*. Similarly, we denote (2) *EAR-Oracle-Adapt* as the adapted oracle of *EAR-Oracle* [30]. By performing a linear scan using the shortest path query results, they can answer other proximity queries.

3.2.1 ***SE-Oracle-Adapt.*** It uses a *compressed partition tree* [58, 59] and *well-separated node pair sets* [14] to index the $(1+\epsilon)$-approximate pairwise P2P shortest surface paths passing on a *TIN* (constructed by the given point cloud). Its oracle construction time, oracle size, and shortest path query time are $O(nN^2+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, where h is the height of the compressed partition tree and β is the largest capacity dimension [27, 34] ($\beta\in[1.5,2]$ according to work [58, 59]). It is regarded as the best-known oracle for the P2P query on a point cloud.

Drawbacks of *SE-Oracle-Adapt*: Its oracle construction time is large due to the *bad criterion for algorithm earlier termination*. For POIs in the same level of the compressed partition tree, they have the *same* earlier termination criteria. But, in *RC-Oracle*, we have *different* earlier termination criteria for each different POI, to minimize the running time of the *SSAD* algorithm. In the P2P query on a point cloud, our experimental results show that for a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt* is 78,000s ≈ 21.7 hours, while *RC-Oracle* just needs 200s ≈ 3.2 min.

3.2.2 ***EAR-Oracle-Adapt.*** It has the same idea of *SE-Oracle-Adapt*, i.e., well-separated node pair sets. Their differences are that *EAR-Oracle-Adapt* adapts *SE-Oracle-Adapt* from the P2P query on a point cloud to the A2A query on a point cloud by using Steiner points on the faces of the *TIN* (constructed by the given point cloud) and *highway nodes* as POIs in well-separated node pair sets construction. Its oracle construction time, oracle size, and shortest path query time are $O(\lambda\xi mN^2+\frac{N^2}{\epsilon^{2\beta}}+\frac{Nh}{\epsilon^{2\beta}}+Nh\log N)$, $O(\frac{\lambda mN}{\xi}+\frac{Nh}{\epsilon^{2\beta}})$, and $O(\lambda\xi\log(\lambda\xi))$, where λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face. It is regarded as the best-known oracle for the A2A query on a point cloud.

Drawbacks of *EAR-Oracle-Adapt*: It also has the *bad criterion for algorithm earlier termination* drawback. In the A2A query on a point cloud, our experimental results show that for a point cloud with 100k points, the oracle construction time of *EAR-Oracle-Adapt* is 50,0000s ≈ 13.9 hours, while *RC-Oracle* just needs 100s ≈ 1.6 min.

3.3 Oracles for other proximity queries

No existing oracle can answer proximity queries on a point cloud. But, works [21, 22, 54] build an oracle to answer proximity queries on a *TIN*. Specifically, works [21, 22] uses a multi-resolution terrain model (resp. *VOronoi diagram Oracle* (*VO-Oracle*) [54] uses a Voronoi diagram) to answer the *kNN* query on a *TIN* in $O(N^2)$ (resp. $O(N\log^2 N)$) time. We adapt *VO-Oracle* to be *VO-Oracle-Adapt* in a similar way of *SE-Oracle-Adapt*. Although *VO-Oracle-Adapt* is regarded as the best-known oracle to directly answer the *kNN* query, work [58, 59] shows the *kNN* query time of *VO-Oracle-Adapt* is up to 5 times larger than that of using *SE-Oracle-Adapt* with a linear scan of the shortest path query result. In addition, work [61] builds an oracle to answer the dynamic version of the *kNN* query, and work [62] builds an oracle to answer the reverse nearest neighbor query, but they are not our main focus.

4 METHODOLOGY

4.1 Overview of *RC-Oracle* for the P2P query

4.1.1 ***Components of RC-Oracle for the P2P query.*** There are two components, i.e., the *path map table* and the *POI map table*.

(1) **The path map table** M_{path} is a *hash table* [18] that stores a set of key-value pairs. For each key-value pair, it stores a pair of POIs u and v in P, as a key $\langle u, v\rangle$, and the corresponding exact shortest path $\Pi^*(u, v|C)$ passing on C, as a value. M_{path} needs linear space in terms of the number of paths to be stored. Given a pair of POIs u and v, M_{path} can return the associated exact shortest path $\Pi^*(u, v|C)$ passing on C in $O(1)$ time. In Figure 3 (e), M_{path} stores exact shortest paths passing on C, corresponding to the 7 paths in Figure 3 (d). For the exact shortest paths passing on C between b and c, M_{path} stores $\langle b, c\rangle$ as a key and $\Pi^*(b, c|C)$ as a value.

(2) **The POI map table** M_{POI} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores a POI u as a key (such that we do not store all the exact shortest paths passing on C in M_{path} from u to other non-processed POIs), and another POI v as a value (such that v is close to u, and we concatenate $\Pi^*(u, v|C)$ and the exact shortest paths passing on C with v as a source, to approximate the shortest paths passing on C with u as a source). The space consumption and query time of M_{POI} is similar to M_{path}. In Figure 3 (e), a is close to b, we concatenate $\Pi^*(b, a|C)$ and the exact shortest paths passing on C with a as a source, to approximate the shortest paths passing on C with b as a source, so we store b as a key, and a as a value.

4.1.2 *Phases of RC-Oracle for the P2P query*.

There are two phases, i.e., *construction phase* and *shortest path query phase* (see Figure 3). (1) In the construction phase, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{POI}. (2) In the shortest path query phase, given a pair of query POIs, M_{path}, and M_{POI}, we answer the path results between this pair of POIs efficiently.

4.2 Key Idea of *RC-Oracle* for the P2P query

4.2.1 *Small oracle construction time*.

We give the reason why *RC-Oracle* has a small oracle construction time for the P2P query.

(1) **Rapid point cloud on-the-fly shortest path querying by algorithm *FastFly***: When constructing *RC-Oracle*, given a point cloud C and a pair of points s and t on C, we use algorithm *FastFly* (a Dijkstra's algorithm [24]) to directly calculate the *exact* shortest path $\Pi^*(s, t|C)$ passing on the conceptual graph (see Figure 2 (c)) of C between s and t.

(2) **Rapid oracle construction**: When constructing *RC-Oracle*, we regard each POI as a source and use algorithm *FastFly*, i.e., a *SSAD* algorithm, for n times for oracle construction, and we assign a *different* earlier termination criteria for each POI to terminate the *SSAD* algorithm earlier for time-saving. There are two versions of a *SSAD* algorithm. (i) Given a source POI and a set of destination POIs, the *SSAD* algorithm can terminate earlier if it has visited all destination POIs. (ii) Given a source POI and a *termination distance* (denoted by D), the *SSAD* algorithm can terminate earlier if the searching distance from the source POI is larger than D. We use the first version. For each POI, by considering more geometry information of the point cloud, including the Euclidean distance and the length of the previously calculated shortest paths, we use *different* earlier termination criteria to calculate the corresponding destination POIs, such that the number of destination POIs is minimized, and these destination POIs are closer to the source POI compared with other POIs.

We use an example for illustration. In Figure 3 (b), for a, we use the *SSAD* algorithm (i.e., *FastFly*) to calculate the shortest path passing on C from a to all other POIs. In Figure 3 (c), for b, if b is close to a, i.e., judged using the previously calculated $|\Pi^*(a, b|C)|$, and b is far away from d (resp. e), i.e., judged using the Euclidean distance $d_E(b, d)$ (resp. $d_E(b, e)$), we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ (resp. $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$) to approximate $\Pi^*(b, d|C)$ (resp. $\Pi^*(b, e|C)$). Thus, we just need to use the *SSAD* algorithm with b as a source, and terminate earlier when it has visited c. In Figure 3 (d), we repeat it for c. Similarly, for d, we use $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ to determine whether we can terminate the *SSAD* algorithm earlier with d as a source. We found that there is even no need to use the *SSAD* algorithm with d as the source. For *different* POIs b and d, we use *different* termination criteria (i.e., $|\Pi^*(a, b|C)|$ and $d_E(b, d)$ for b, $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ for d) to calculate a different set of destination POIs for time-saving.

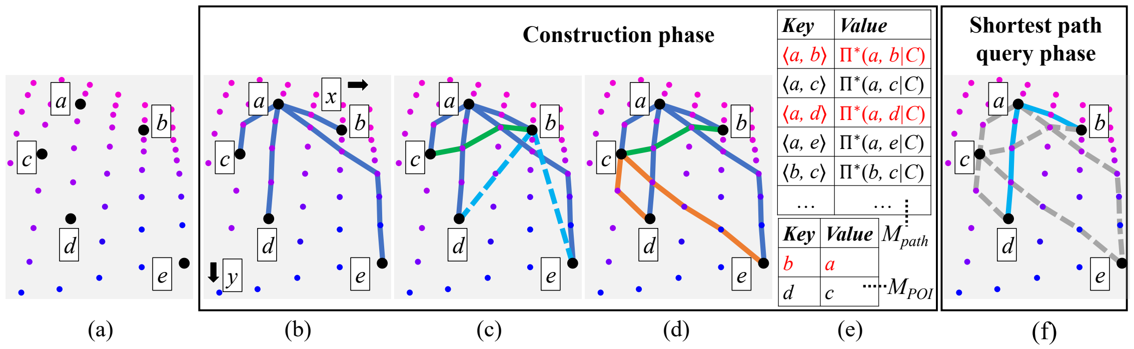
However, in *SE-Oracle-Adapt*, it has the *bad criterion for algorithm earlier termination* drawback. After the construction of the compressed partition tree, it pre-computes the shortest surface paths passing on T using the *SSAD* algorithm (i.e., *CH-Adapt*) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of the *SSAD* algorithm and sets termination distance $D = \frac{8r}{\epsilon} + 10r$, where r is the radius of the source POI in the compressed partition tree. Given two POIs a and b in the same level of the tree, their termination distances are the same (suppose that the value is d_1). However, for a, it is enough to terminate the *SSAD* algorithm when the searching distance from a is larger than d_2, where $d_2 < d_1$. This results in a large oracle construction time. In Figure 4, when processing d, suppose that b and d are in the same level of the tree, and they use the *same* termination criteria to get the *same* termination distance D. Since $|\Pi^*(d, e|C)| < D$, for d, it cannot terminate the *SSAD* algorithm earlier until e is visited. The two versions of the *SSAD* algorithm are similar, we achieve a small oracle construction time mainly by using *different* termination criteria for different POIs, unlike using the *same* termination criteria for different POIs in *SE-Oracle-Adapt*.

4.2.2 *Small oracle size*.

We introduce the reason why *RC-Oracle* has a small oracle size for the P2P query. We only store a small number of paths in *RC-Oracle*, i.e., we do not store the path between each pair of POIs. In Figure 3 (d), for a pair of POIs b and d, we use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ to approximate $\Pi^*(b, d|C)$, i.e., we will not store $\Pi^*(b, d|C)$ in M_{path} for memory saving.

4.2.3 *Small shortest path query time*.

We use an example to introduce the reason why *RC-Oracle* has a small shortest path query time for the P2P query. In Figure 3 (f), in the shortest path query phase of *RC-Oracle*, we need to query the shortest path passing on C (1) between a and d, and (2) between b and d. (1) For a and d, since $\langle a, d\rangle \in M_{path}.key$, we can directly return $\Pi^*(a, d|C)$. (2) For b and d, since $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{POI}, we retrieve the value a using the key that is processed first, i.e., b, in M_{POI}, then in M_{path}, we use $\langle b, a\rangle$ and $\langle a, d\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$.



Key	*Value*
$\langle a, b\rangle$	$\Pi^*(a, b\|C)$
$\langle a, c\rangle$	$\Pi^*(a, c\|C)$
$\langle a, d\rangle$	$\Pi^*(a, d\|C)$
$\langle a, e\rangle$	$\Pi^*(a, e\|C)$
$\langle b, c\rangle$	$\Pi^*(b, c\|C)$
…	…

Key	*Value*
b	a
d	c

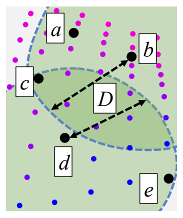


Figure 4: *SE-Oracle-Adapt*

Figure 3: Framework overview

4.3 Implementation Details of *RC-Oracle* for the P2P query

4.3.1 ***Construction Phase.*** Given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs, store them in M_{path}, and store the non-selected POIs and their corresponding POIs in M_{POI}.

Notation: Let $P_{remain} = \{p_1, p_2, \dots\}$ be a set of remaining POIs of P that we have not used algorithm *FastFly* to calculate the exact shortest paths passing on C with $p_i \in P_{remain}$ as a source. P_{remain} is initialized to be P. Given a POI q, let $P_{dest}(q) = \{p_1, p_2, \dots\}$ be a set of POIs of P that we need to use *FastFly* to calculate the exact shortest paths passing on C from q to $p_i \in P_{dest}(q)$. $P_{dest}(q)$ is empty at the beginning. In Figure 3 (c), $P_{remain} = \{c, d, e\}$ since we have not used *FastFly* to calculate the exact shortest paths with c, d, e as source. $P_{dest}(b) = \{c\}$ since we need to use *FastFly* to calculate the exact shortest path from b to c.

Detail and example: Algorithm 1 shows the construction phase in detail, and the following illustrates it with an example.

(1) *POIs sorting* (lines 2-3): In Figure 3 (b), since the side length of C along y-axis is longer than that of x-axis, the sorted POIs are a, b, c, d, e.

(2) *Shortest paths calculation* (lines 4-20): There are two cases.

(i) *Exact shortest paths calculation* (lines 5-9): In Figure 3 (b), a has the smallest y-coordinate based on the sorted POIs in P_{remain}, we delete a from P_{remain} (so $P_{remain} = \{b, c, d, e\}$), calculate the exact shortest paths passing on C from a to b, c, d, e (in purple lines) using algorithm *FastFly*, and store each POIs pair as a key and the corresponding path as a value in M_{path}.

(ii) *Shortest paths approximation* (lines 10-20): In Figure 3 (c), b is the POI in P_{remain} closest to a, c is the POI in P_{remain} second closest to a, so the following order is $b, c, \dots$. There are two cases:

- *Entering approximation looping* (lines 11-20): In Figure 3 (c), we first select a's closest POI in P_{remain}, i.e., b, since $d_E(a, b) \leq \epsilon L$, it means a and b are not far away, we enter approximation looping, delete b from P_{remain}, so $P_{remain} = \{c, d, e\}$. There are three steps:
  - *Far away POIs* (lines 14-15): In Figure 3 (c), $d_E(b, d) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$ and $d_E(b, e) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, it means d and e are far away from b, we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ that we have already calculated before to approximate $\Pi^*(b, d|C)$, and use $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$ that we have already calculated before to approximate $\Pi^*(b, e|C)$, so we get $\Pi(b, d|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi(b, e|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{POI}.
  - *Close POIs* (lines 16-17): In Figure 3 (c), $d_E(b, c) \leq \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, it means c is close to b, so we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(b, c|C)$, then we store c into $P_{dest}(b)$.
  - *Selected exact shortest paths calculation* (lines 18-20): In Figure 3 (c), when we have processed all POIs in P_{remain} with b as a source, we have $P_{dest}(b) = \{c\}$, we use algorithm *FastFly* to calculate the exact shortest path passing on C between b and c, i.e., $\Pi^*(b, c|C)$ (in green line), and store $\langle b, c\rangle$ as a key and $\Pi^*(u, c|C)$ as a value in M_{path}. Note that we can terminate

Algorithm 1 *Construction* (C, P)

Input: a point cloud C and a set of POIs P
Output: a path map table M_{path} and a POI map table M_{POI}

```
1:  $P_{remain} \leftarrow P$, $M_{path} \leftarrow \emptyset$, $M_{POI} \leftarrow \emptyset$
2:  if $L_x \geq L_y$ (resp. $L_x < L_y$) then
3:      sort POIs in $P_{remain}$ in ascending order using $x$-coordinate (resp. $y$-coordinate)
4:  while $P_{remain}$ is not empty do
5:      $u \leftarrow$ a POI in $P_{remain}$ with the smallest $x$-coordinate / $y$-coordinate
6:      $P_{remain} \leftarrow P_{remain} - \{u\}$
7:      calculate the exact shortest paths passing on $C$ from $u$ to each POI in $P_{remain}$ simultaneously using algorithm FastFly
8:      for each POI $v \in P_{remain}$ do
9:          $key \leftarrow \langle u, v\rangle$, $value \leftarrow \Pi^*(u, c|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
10:     sort POIs in $P_{remain}$ in ascending order using the exact distance on $C$ between $u$ and each $v \in P_{remain}$, i.e., $\Pi^*(u, v|C)$
11:     for each sorted POI $v \in P_{remain}$ such that $|\Pi^*(u, v|C)| \leq \epsilon L$ do
12:         $P_{remain} \leftarrow P_{remain} - \{v\}$, $P_{dest}(v) \leftarrow \emptyset$
13:         for each POI $w \in P_{remain}$ do
14:             if $d_E(v, w) > \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ and $v \notin M_{POI}.key$ then
15:                 $key \leftarrow v$, $value \leftarrow u$, $M_{POI} \leftarrow M_{POI} \cup \{key, value\}$
16:             else if $d_E(v, w) \leq \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ then
17:                 $P_{dest}(v) \leftarrow P_{dest}(v) \cup \{w\}$
18:         calculate the exact shortest paths passing on $C$ from $v$ to each POI in $P_{dest}(v)$ simultaneously using algorithm FastFly
19:         for each POI $w \in P_{dest}(v)$ do
20:             $key \leftarrow \langle v, w\rangle$, $value \leftarrow \Pi^*(v, w|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
21: return $M_{path}$ and $M_{POI}$
```

algorithm *FastFly* earlier since we just need to visit POIs that are close to b, and we do not need to visit d and e.

- *Leaving approximation looping* (line 11): In Figure 3 (c), since we have processed b, and $P_{remain} = \{c, d, e\}$, we select a's closest POI in P_{remain}, i.e., c, since $d_E(a, c) > \epsilon L$, it means a and c are far away, and it is unlikely to have a POI m that satisfies $d_E(c, m) > \frac{2}{\epsilon} \cdot |\Pi^*(a, c|C)|$, we leave approximation looping and terminate the iteration.

(3) *Shortest paths calculation iteration* (lines 4-20): In Figure 3 (d), we repeat the iteration, and calculate the exact shortest paths passing on C with c as a source (in orange lines).

4.3.2 ***Shortest Path Query Phase.*** Given a pair of POIs s and t in P, M_{path}, and M_{POI}, *RC-Oracle* can answer the associated shortest path $\Pi(s, t|C)$ passing on C, which is a $(1+\epsilon)$-approximated exact shortest path of $\Pi^*(s, t|C)$ in $O(1)$ time. Given a pair of POIs s and t, there are two cases (s and t are interchangeable, i.e., $\langle s, t\rangle = \langle t, s\rangle$):

- **Retrieve exact shortest path**: If $\langle s, t\rangle \in M_{path}.key$, we retrieve $\Pi^*(s, t|C)$ using $\langle s, t\rangle$ in $O(1)$ time (in Figure 3 (e), given a pair of POIs a and d, since $\langle a, d\rangle \in M_{path}.key$, we can retrieve $\Pi^*(a, d|C)$ in $O(1)$ time).
- **Retrieve approximate shortest path**: If $\langle s, t\rangle \notin M_{path}.key$, it means $\Pi^*(s, t|C)$ is approximated by two exact shortest paths passing on C in M_{path}, and (1) either s or t is a key in M_{POI}, or (2) both s and t are keys in M_{POI}. Without loss of generality, suppose that (1) s exists in M_{POI} if either s or t is a key in M_{POI}, or (2) s is processed a head of t during construction phase if both s and t are keys in M_{POI}. For both of two cases, we retrieve the value s' using the key s from M_{POI} in $O(1)$ time, then retrieve $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ from M_{path} using $\langle s, s'\rangle$ and $\langle s', t\rangle$ in $O(1)$ time, and use $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ to approximate $\Pi^*(s, t|C)$ (in Figure 3 (c), (1) given a pair of POIs b and e, since $\langle b, e\rangle \notin M_{path}.key$, b is a key in M_{POI}, so we retrieve the value a using the key b in M_{POI}, then in M_{path}, we use $\langle b, a\rangle$ and $\langle a, e\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, for approximating $\Pi^*(b, e|C)$, or (2) given a pair of POIs b and d, since $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{POI}, so we retrieve the value a using the key that is processed first, i.e., b, in M_{POI}, then in M_{path}, we use $\langle b, a\rangle$ and $\langle a, d\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$).

4.4 *RC-Oracle* for the A2A query

We can adapt *RC-Oracle* that answers the P2P query to answer the A2A query on a point cloud. Specifically, we simply create POIs which has the same coordinate values as all points in the point cloud, then *RC-Oracle* can answer the A2A query on a point cloud. We just need to pre-compute the exact shortest paths passing on the point cloud between some selected pairs of points on the point cloud (not all pairs of points on the point cloud), so *RC-Oracle* for the A2A query on the point cloud also has a small oracle construction time, small oracle size, and small shortest path query time.

4.5 Proximity Query Algorithms

Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k, and a range value r, we can answer other proximity queries, i.e., the *kNN* and range query using *RC-Oracle*. In the P2P (resp. A2A) query, these objects are POIs in P (resp. points on C). A naive algorithm is to perform a linear scan using the shortest path query results. We propose an efficient algorithm. Intuitively, when constructing *RC-Oracle*, we have used the *SSAD* algorithm to calculate shortest paths passing on C with q as a source and sorted these paths in ascending order based on their distance in M_{path}. For these paths, we do not need to perform a linear scan for all of them in proximity queries for time-saving.

Detail and example: There are two cases: (1) If $q \in M_{POI}.key$, we retrieve the value q' using the key q from M_{POI}. For the objects processed before q' during the construction phase, we perform a linear scan using the shortest path query result between q and these objects. For the objects (not including q) processed after q', the shortest paths passing on C between q' and these objects are sorted in order in M_{path}, we just need to use these distances plus $|\Pi^*(q, q'|C)|$, to get the approximate shortest paths between q and these objects in sorted order, so there is no need to perform a linear scan for all of them. In Figure 3 (c), $b \in M_{POI}.key$, we retrieve the value a using the key b from M_{POI}. There is no POI processed before a. For $\{c, d, e\}$ processed after a, $|\Pi^*(a, c|C)|$, $|\Pi^*(a, d|C)|$, and $|\Pi^*(a, e|C)|$ are sorted in order, so $|\Pi(b, c|C)|$, $|\Pi(b, d|C)|$, and $|\Pi(b, e|C)|$ are also sorted in order. (2) If $q \notin M_{POI}.key$, we just need to replace both q and q' in the first case as q, then perform the similar algorithm. For both two cases, we can then return the corresponding *kNN* and range query results. Our experimental results show that *kNN* query time of all 1000 objects using the *RC-Oracle* with the efficient algorithm is 25s, but is 50s for that of the naive algorithm.

The above description is just a simple application of *RC-Oracle*. We focus on key issues of point clouds, which have a lot of substantial materials. Designing a novel index particularly for the *kNN* and range query is left as the future work.

4.6 Theoretical Analysis

4.6.1 ***Algorithm FastFly and RC-Oracle.*** The analysis of them are in Theorem 4.1 and Theorem 4.2, respectively.

THEOREM 4.1. *The shortest path query time and memory consumption of algorithm FastFly are $O(N \log N)$ and $O(N)$. Algorithm FastFly returns the exact shortest path passing on the point cloud.*

PROOF. Since algorithm *FastFly* is a Dijkstra algorithm and there are total N points, we obtain the shortest path query time and memory consumption. Since Dijkstra algorithm is guaranteed to return the exact shortest path, algorithm *FastFly* returns the exact shortest path passing on the point cloud. □

THEOREM 4.2. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle for (1) the P2P query are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$, $O(1)$, respectively, and (2) the A2A query are $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$, $O(1)$, respectively. RC-Oracle always has $|\Pi(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ for each pair of (1) POIs s and t in P for the P2P query, and (2) points s and t on C for the A2A query.*

PROOF. We give the proof for the P2P query as follows.

Firstly, we prove the *oracle construction time*. (1) In *POIs sorting* step, it needs $O(n \log)$ time. Since there are n POIs, and we use the quick sort for sorting. (2) In *shortest paths calculation* step, it needs

$O(\frac{N \log N}{\epsilon} + n)$ time. (i) It needs to use $O(\frac{1}{\epsilon})$ POIs as a source to run algorithm *FastFly* for exact shortest paths calculation according to standard packing property [29], and each algorithm *FastFly* needs $O(N \log N)$ time. (ii) For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly*, we just calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for shortest paths approximation. (3) So the oracle construction time is $O(\frac{N \log N}{\epsilon} + n \log n)$.

Secondly, we prove the *oracle size*. (1) For M_{POI}, its size is $O(n)$ since there are n POIs. (2) For M_{path}, its size is $O(\frac{n}{\epsilon})$. We store (i) $O(\frac{n}{\epsilon})$ exact shortest paths passing on C from $O(\frac{1}{\epsilon})$ POIs (that uses algorithm *FastFly* as a source and cover all other POIs) to other $O(n)$ POIs, and (ii) $O(n)$ exact shortest paths passing on C from $O(n)$ POIs (that uses algorithm *FastFly* as a source and cover only some of POIs) to other $O(1)$ POIs. (3) So the oracle size is $O(\frac{n}{\epsilon})$.

Thirdly, we prove the *shortest path query time*. (1) If $\Pi^*(s, t|C) \in M_{path}$, the shortest path query time is $O(1)$. (2) If $\Pi^*(s, t|C) \notin M_{path}$, we need to retrieve s' from M_{POI} using s in $O(1)$ time, and retrieve $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ from M_{path} using $\langle s, s' \rangle$ and $\langle s', t \rangle$ in $O(1)$ time, so the shortest path query time is still $O(1)$. Thus, the shortest path query time of *RC-Oracle* is $O(1)$.

Fourthly, we prove the *error bound*. Given a pair of POIs s and t, if $\Pi^*(s, t|C)$ exists in M_{path}, then there is no error. Thus, we only consider the case that $\Pi^*(s, t|C)$ does not exist in M_{path}. Suppose that u is a POI close to s, such that $\Pi(s, t|C)$ is calculated by concatenating $\Pi^*(s, u|C)$ and $\Pi^*(u, t|C)$. This means that $d_E(s, t) > \frac{2}{\epsilon} \cdot \Pi^*(u, s|C)$. So we have $|\Pi^*(s, u|C)| + |\Pi^*(u, t|C)| < |\Pi^*(s, u|C)| + |\Pi^*(u, s|C)| + |\Pi^*(s, t|C)| = |\Pi^*(s, t|C)| + 2 \cdot |\Pi^*(u, s|C)| < |\Pi^*(s, t|C)| + \epsilon \cdot d_E(s, t) \leq |\Pi^*(s, t|C)| + \epsilon \cdot |\Pi^*(s, t|C)| = (1 + \epsilon)|\Pi^*(s, t|C)|$. The first inequality is due to triangle inequality. The second equation is because $|\Pi^*(u, s|C)| = |\Pi^*(s, u|C)|$. The third inequality is because we have $d_E(s, t) > \frac{2}{\epsilon} \cdot \Pi^*(u, s|C)$. The fourth inequality is because the Euclidean distance between two points is no larger than the distance of the shortest path passing on the point cloud between the same two points.

We give the proof for the A2A query as follows. We need to change (1) n to N in the oracle construction time and oracle size, and (2) pairs of POIs in P to pairs of points on C in the error bound. □

4.6.2 ***The shortest path passing on a point cloud and the shortest surface or network path passing on a TIN***.

We show the relationship of $|\Pi^*(s, t|C)|$ with $|\Pi_N(s, t|T)|$ and $|\Pi^*(s, t|T)|$ in Lemma 4.3

Lemma 4.3. *Given a pair of points s and t on C, we have (1) $|\Pi^*(s, t|C)| \leq |\Pi_N(s, t|T)|$ and (2) $|\Pi^*(s, t|C)| \leq k \cdot |\Pi^*(s, t|T)|$, where $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$.*

Proof. (1) In Figure 2 (a), given a green point q on C, it can connect with one of its 8 neighbor points (7 blue points and 1 red point s). In Figure 2 (b), given a green vertex q on T, it can only connect with one of its 6 blue neighbor vertices. So $|\Pi^*(s, t|C)| \leq |\Pi_N(s, t|T)|$. (2) We let $\Pi_E(s, t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s, t|T)$ passes) between s and t. According to left hand side equation in Lemma 2 of work [33], we have $|\Pi_E(s, t|T)| \leq k \cdot |\Pi^*(s, t|T)|$. Since $\Pi_N(s, t|T)$ considers all the edges on T, $|\Pi_N(s, t|T)| \leq |\Pi_E(s, t|T)|$. Thus, we finish the proof by combining these inequalities. □

4.6.3 ***Proximity query algorithms***.

We provide analysis on the proximity query algorithms using *RC-Oracle*. For the *kNN* range query, both of them return a set of objects. Given a query object q, we let v_f (resp. v'_f) be the furthest object to q among the returned objects calculated using the exact distance on C (resp. the approximated distance on C returned by *RC-Oracle*). In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d. And d is the furthest POI to a in these two POIs, i.e., $v_f = d$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c. And b is the furthest POI to a in these two POIs, i.e., $v'_f = b$. We define the error rate of the *kNN* and range query to be $\frac{|\Pi^*(q, v'_f|C)|}{|\Pi^*(q, v_f|C)|}$, which is a real number no smaller than 1. In Figure 1 (a), the error rate is $\frac{|\Pi^*(a,b|C)|}{|\Pi^*(a,d|C)|}$. Recall the error parameter of *RC-Oracle* is ϵ. Then, we show the query time and error rate of *kNN* and range query using *RC-Oracle* in Theorem 4.4.

Theorem 4.4. *The query time and error rate of both the kNN and range query by using RC-Oracle are $O(n')$ and $1 + \epsilon$, respectively.*

Proof Sketch. The *query time* is due to the usages of the shortest path query phase of *RC-Oracle* for n' times in the worst case. The *error rate* is due to its definition and the error of *RC-Oracle*. The detailed proof appears in the appendix. □

5 EMPIRICAL STUDIES

5.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.2 GHz CPU and 512GB memory. All algorithms were implemented in C++. Our experimental setup generally follows the setups in the literature [32, 33, 42, 58, 59]. We conducted experiments with point clouds and *TIN*s as input, separately.

Datasets: (1) Point cloud datasets: We conducted our experiment based on 34 real point cloud datasets in Table 2, where the subscript p means a point cloud. For BH_p and EP_p datasets, they are represented as a point cloud with 8km × 6km covered region. For GF_p, LM_p and RM_p, we first obtained the satellite map from Google Earth [2] with 8km × 6km covered region, and then used Blender [1] to generate the point cloud. These five original datasets have a resolution of 10m × 10m [21, 42, 54, 58, 59]. We extracted 500 POIs using OpenStreetMap [58, 59] for these datasets in the P2P query. For small-version datasets, we use the same region of the original datasets with a (lower) resolution of 70m × 70m and the dataset generation procedure in work [42, 58, 59] to generate them. This procedure can be found in the appendix. In addition, we have six sets of multi-resolution datasets with different numbers of points generated using the original and small-version datasets with the same procedure. (2) *TIN* datasets: Based on the 34 point cloud datasets, we triangulate [49] them and generate another 34 *TIN* datasets, and use t as the subscript. For example, BH_t means a *TIN* dataset generated using the BH_p point cloud dataset.

Algorithms: (1) Algorithms that support the shortest path query (and also other proximity queries) on a point cloud (i.e., algorithms for solving the problem studied in this paper): We adapted existing algorithms, originally designed for the problem on *TIN*s, for our problem on point clouds by performing the triangulation approach on the point cloud to obtain a *TIN* [49] (i.e., we store the *TIN* as a

Table 2: Point cloud datasets

Name	$\|N\|$
Original dataset	
BearHead (*BH$_p$*) [5, 58, 59]	0.5M
EaglePeak (*EP$_p$*) [5, 58, 59]	0.5M
GunnisonForest (*GF$_p$*) [7]	0.5M
LaramieMount (*LM$_p$*) [8]	0.5M
RobinsonMount (*RM$_p$*) [3]	0.5M
Small-version dataset	
BH$_p$-small	10k
EP$_p$-small	10k
GF$_p$-small	10k
LM$_p$-small	10k
RM$_p$-small	10k
Multi-resolution dataset	
BH$_p$ multi-resolution	1M, 1.5M, 2M, 2.5M
EP$_p$ multi-resolution	1M, 1.5M, 2M, 2.5M
GF$_p$ multi-resolution	1M, 1.5M, 2M, 2.5M
LM$_p$ multi-resolution	1M, 1.5M, 2M, 2.5M
RM$_p$ multi-resolution	1M, 1.5M, 2M, 2.5M
EP$_p$-small multi-resolution	20k, 30k, 40k, 50k

Table 3: Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error
Oracle-based algorithm							
SE-Oracle-Adapt [58, 59]	$O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
EAR-Oracle-Adapt [30]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium	Small
RC-Oracle-Naive	$O(nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Small
***RC-Oracle* (ours)**	$\boldsymbol{O(\frac{N\log N}{\epsilon} + n\log n)}$	**Small**	$\boldsymbol{O(\frac{n}{\epsilon})}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Small**
On-the-fly algorithm							
CH-Adapt [16]	-	N/A	-	N/A	$O(N^2)$	Large	Small
Kaul-Adapt [32]	-	N/A	-	N/A	$O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large	Small
Dijk-Adapt [33]	-	N/A	-	N/A	$O(N\log N)$	Medium	Medium
***FastFly* (ours)**	-	N/A	-	N/A	$\boldsymbol{O(N\log N)}$	**Medium**	**No error**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [27, 34], λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, θ is the minimum inner angle of any face in T, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T.

data structure and clear the given point cloud from the memory) so that the existing algorithm could be used. Their algorithm names are appended by "*-Adapt*". We have four on-the-fly algorithms, i.e., (i) *CH-Adapt* [16], (ii) *Kaul-Adapt* [32], (iii) *Dijk-Adapt* [33], and (iv) *FastFly*: our algorithm. We have four oracles, i.e., (v) *SE-Oracle-Adapt*: the best-known oracle [58, 59] for the P2P query on a point cloud, (vi) *EAR-Oracle-Adapt*: the best-known oracle [30] for the A2A query on a point cloud, (vii) *RC-Oracle-Naive*: the naive version of our oracle *RC-Oracle* without shortest paths approximation step, and (viii) *RC-Oracle*: our oracle. We compare them in Table 3.

(2) Algorithms that support the shortest path query (and also other proximity queries) on a *TIN* (i.e., algorithms for solving the problem studied by previous studies [30, 58, 59]): Similarly, we have four on-the-fly algorithms, i.e., (i) *CH* [16], (ii) *Kaul* [32], (iii) *Dijk* [33], (iv) *FastFly-Adapt*: our adapted algorithm (for the queries on a *TIN*) that calculates the shortest path passing on a conceptual graph of a *TIN*, where the vertices of this conceptual graph are formed by the vertices of the given *TIN*, and the edges of this graph are formed by adding edges between each vertex and its 8 neighbor vertices (this conceptual graph is similar to the one in Figure 2 (c), we store it as a data structure and clear the given *TIN* from the memory). We have four oracles, i.e., (v) *SE-Oracle* [58, 59], (vi) *EAR-Oracle* [30], (vii) *RC-Oracle-Naive-Adapt*: the adapted naive version of our oracle without shortest paths approximation step that calculates the shortest path passing on a conceptual graph of a *TIN*, and (viii) *RC-Oracle-Adapt*: our adapted oracle that calculates the shortest path passing on a conceptual graph of a *TIN*.

Query Generation: We conducted all proximity queries, i.e., (1) shortest path query, (2) all objects *kNN* query, and (3) all objects range query. (1) For the shortest path query, we issued 100 query instances where for each instance, we randomly chose two points (i) in P for the P2P query on a point cloud or a *TIN*, or (ii) on the point cloud (resp. *TIN*) for the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), one as a source and the other as a destination. The average, minimum, and maximum results were reported. In the experimental result figures, the vertical bar and the points mean the minimum, maximum, and average results. (2 & 3) For all objects *kNN* query and range query, we perform the proximity query algorithm for *RC-Oracle* in Section 4.5 and a linear scan for other baselines (as described in work [59]) using all objects as query objects. In the P2P query on a point cloud or a *TIN*, these objects are POIs in P. In the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), we randomly select 2500 points on the point cloud (resp. *TIN*) as objects. Since we perform linear scans or use the sorted distance stored in M_{path} for proximity query algorithms (because there is no index designed for the *kNN* and range query in this study), the value of k and r will not affect their query time, we set $k = 3$ and $r = 1$km.

Factors and Measurements: We studied three factors for the P2P query, namely (1) ϵ (i.e., the error parameter), (2) n (i.e., the number of POIs), and (3) N (i.e., the number of points in a point cloud dataset or the number of vertices in a *TIN* dataset). We studied one factor ϵ for the A2A query. In addition, we used nine measurements to evaluate the algorithm performance, namely (1) *oracle construction time*, (2) *memory consumption* (i.e., the space consumption when running the algorithm), (3) *oracle size*, (4) *query time* (i.e., the shortest path query time), (5) *kNN query time* (i.e., all objects kNN query time), (6) *range query time* (i.e., all objects range query time), (7) *distance error* (i.e., the error of the distance returned by the algorithm compared with the exact distance), (8) *kNN query error* (i.e., the error rate of the *kNN* query defined in Section 4.6.3), and (9) *range query error* (i.e., the error rate of the range query defined in Section 4.6.3).

5.2 Experimental Results for *TIN*s

We first study proximity queries on *TIN*s (studied by previous studies [30, 58, 59]) to justify why our proximity queries on *point clouds* are useful in practice. We have the following settings. (1) The distance of the path calculated by *CH* is used for distance error calculation since the path is the exact shortest surface path passing on the *TIN*. (2) *SE-Oracle*, *EAR-Oracle*, and *RC-Oracle-Naive-Adapt* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets (with

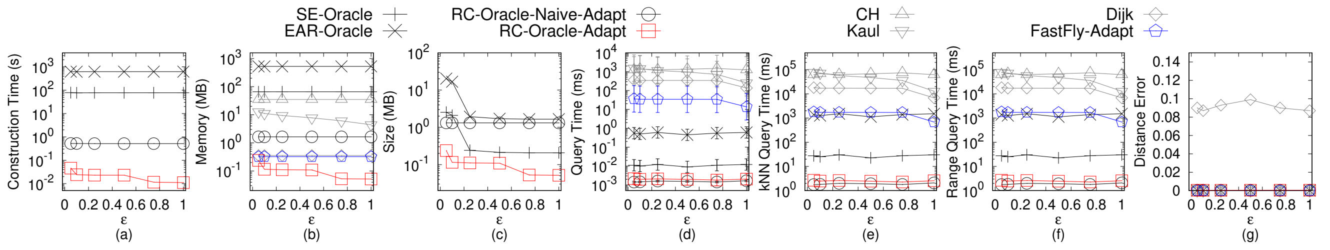


Figure 5: Baseline comparisons (effect of ϵ on *BH_t-small TIN* dataset for the P2P query)

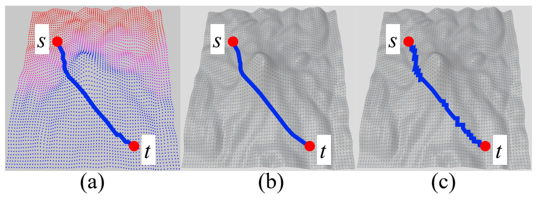


Figure 6: (a) The shortest path passing on a point cloud, (b) the shortest surface path passing on a *TIN*, and (c) the shortest network path passing on a *TIN*

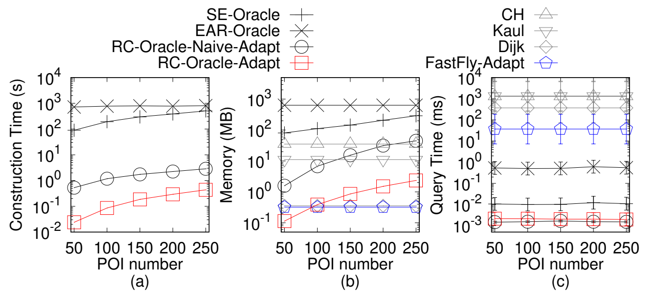


Figure 7: Baseline comparisons (effect of n on *EP_t-small TIN* dataset for the P2P query)

default 50 POIs for the P2P query), and (ii) compared *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets (with default 500 POIs for the P2P query). (3) The transformation time from a *TIN* to the conceptual graph of *FastFly-Adapt*, *RC-Oracle-Naive-Adapt*, and *RC-Oracle-Adapt* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly-Adapt*, and (ii) in the oracle construction time for *RC-Oracle-Adapt* and *RC-Oracle-Naive-Adapt*. (4) The transformation time from a *TIN* to the conceptual graph of *Dijk* is also only counted once in its shortest path query time, the *kNN* and range query time.

5.2.1 ***Baseline comparisons.*** We study the effect of ϵ and n for the P2P query on a *TIN* in this subsection. We study the effect of N for the P2P query, and the comparisons for the AR2AR query on a *TIN* in the appendix.

Effect of ϵ for the P2P query on a *TIN*. In Figure 5, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *BH_p-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. Although a *TIN* is given as input, *RC-Oracle-Adapt* performs better than *SE-Oracle*, *EAR-Oracle*, and *RC-Oracle-Naive-Adapt* in terms of the oracle construction time, oracle size, and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *CH* (although *FastFly-Adapt* needs to construct a conceptual graph from the given *TIN*, and there is no other additional steps for *CH*), and the distance error of *FastFly-Adapt* (with distance error 0.008) is much smaller than that of *Dijk* (with distance error 0.1). Figure 6 (a) shows the shortest path passing on a point cloud, and Figure 6 (b) (resp. Figure 6 (c)) shows the shortest surface (resp. network) path passing on a *TIN* of Mount Rainier in an area of 20km × 20km. The path in Figure 6 (a) and (b) are similar, but calculating the former path is much faster than the latter path, since the query region of the former path is smaller than the latter path. The path in Figure 6 (c) has a larger error than the path in Figure 6 (a). This motivates us to conduct experiments on point clouds. The *kNN* query error and range query error are all equal to 0 (since the distance error is very small), so their results are omitted.

Effect of n for the P2P query on a *TIN*. In Figure 7, we tested 5 values of n from {50, 100, 150, 200, 250} on *EP_t* dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons. In Figure 7 (b), when n increases, the memory consumption of *RC-Oracle-Adapt* exceeds that of *Dijk* and *FastFly-Adapt*. This is because (1) *RC-Oracle-Adapt* is an oracle which is affected by n, it needs more memory consumption during the oracle construction phase to calculate more shortest paths among these POIs when n increases, but (2) *Dijk* and *FastFly-Adapt* are on-the-fly algorithms which are not affected by n, their memory consumption only measure the space consumption for calculating one shortest path.

5.3 Experimental Results for Point Clouds

Now, we understand the effectiveness of proximity queries on *point clouds*. In this section, we then study proximity queries on *point clouds* using the algorithms in Table 3. We have the following setting. (1) The distance of the path calculated by *FastFly* is used for distance error calculation since the path is the exact shortest path passing on the point cloud. (2) *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*, and *RC-Oracle-Naive* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets (with default 50 POIs for the P2P query), and (ii) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets (with default 500 POIs for the P2P query). (3) Since algorithm *FastFly* uses the Dijkstra algorithm on the conceptual graph of a point cloud, it is the same as the shortest path algorithm on a general graph (constructed by the given point cloud), we do not (and there is no need to) compare them in the experiment. But, there is no existing work discussing how to build a conceptual graph from a point cloud. We fill this gap by proposing algorithm *FastFly*. (4) The transformation time from a point cloud to the conceptual graph of *FastFly*, *RC-Oracle-Naive*, and *RC-Oracle* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly*, and (ii) in the oracle construction time for *RC-Oracle* and *RC-Oracle-Naive*. (5) The transformation time from a point cloud to a *TIN* is also only counted once (i) in the shortest path query time, the *kNN* and range query time for *CH-Adapt*, *Kaul-Adapt*, and (ii) in the oracle construction time for *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*. (6) The transformation time from a point cloud to a *TIN*, and then to the conceptual graph of *Dijk-Adapt* is also only counted once in its shortest path query time, the *kNN* and range query time.

5.3.1 ***Baseline comparisons.***

We study the effect of ϵ, n, and N for the P2P query on a point cloud, and the comparisons for the A2A query on a point cloud in this subsection.

Effect of ϵ for the P2P query on a point cloud. In Figure 8, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on EP_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. (1) For *RC-Oracle* and the best-known oracle *SE-Oracle-Adapt*, (i) the oracle construction time and memory consumption, (ii) oracle size, and (iii) shortest path query time of *RC-Oracle* are all smaller than *SE-Oracle-Adapt*, since (i) *SE-Oracle-Adapt* has the *bad criterion for algorithm earlier termination* drawback, it cannot terminate the *SSAD* algorithm earlier, and thus requires more time and memory, (ii) *RC-Oracle* can terminate the *SSAD* algorithm earlier and store fewer paths, (iii) *RC-Oracle*'s shortest path query time is $O(1)$, while the time is $O(h^2)$ for *SE-Oracle-Adapt*. (2) *RC-Oracle* performs better than other on-the-fly algorithms in terms of the shortest path query time since it is an oracle. (3) Algorithm *FastFly* performs better than other on-the-fly algorithms in terms of the shortest path query time since it calculates the shortest path passing on a point cloud. (4) In Figure 8 (a) & (b), regarding the oracle construction time and memory consumption, the variation of ϵ (i) has a large effect on *RC-Oracle*, but due to the log scale used in the experimental figures, the effect is not obvious, (ii) has a small effect on *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*, because even when ϵ is large, they cannot terminate the *SSAD* algorithm earlier for most of the cases due to their *bad criterion for algorithm earlier termination* drawback, and (iii) has no effect on *RC-Oracle-Naive* since it is independent of ϵ. (5) The *kNN* and range query time of *RC-Oracle* are much smaller than the on-the-fly algorithms. (6) The distance error of *RC-Oracle* is close to 0.

Effect of n for the P2P query on a point cloud. In Figure 9, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on GF_p dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons. Since *RC-Oracle* is an oracle, its shortest path query time is smaller than on-the-fly algorithms, which shows the usefulness of using an oracle for faster querying.

Effect of N (scalability test) for the P2P query on a point cloud. In Figure 10, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on LM_p dataset by setting n to be 500 and ϵ to be 0.25 for baseline comparisons. *RC-Oracle* performs better than all the remaining algorithms in terms of the range query time. The oracle construction time of *RC-Oracle* is only 200s $\approx$ 3.2 min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of *RC-Oracle*. In addition, algorithm *FastFly* performs better than other on-the-fly algorithms in terms of the shortest path query time since it calculates the shortest path passing on a point cloud.

A2A query on a point cloud. In Figure 11, we tested the A2A query by varying ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ and setting N to be 10k on a small-version of EP_p dataset. Although *EAR-Oracle-Adapt* is regarded as the best-known oracle in the A2A query on a point cloud, *RC-Oracle* still performs much better than it due to the *bad criterion for algorithm earlier termination* drawback of *EAR-Oracle-Adapt*.

5.3.2 ***Ablation study for the P2P query on a point cloud.***

We denote *SE-Oracle-FastFly-Adapt* (resp. *EAR-Oracle-FastFly-Adapt*) to be another adapted oracle of *SE-Oracle-Adapt* (resp. *EAR-Oracle-Adapt*) that uses algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without constructing a *TIN*. In Figure 12, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on RM_p dataset by setting N to be 0.5M and n to be 500 for ablation study among *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt*, and *RC-Oracle*, such that they only differ by the oracle construction. The oracle construction time, oracle size, and shortest path query time of *RC-Oracle* perform better than the two baselines.

5.3.3 ***Comparisons with other proximity queries oracle and algorithm on a point cloud.***

We also compared *RC-Oracle* using our efficient proximity query algorithm with *VO-Oracle-Adapt* (i.e., the oracle designed for the *kNN* query), and *RC-Oracle* using the naive proximity query algorithm in the appendix. By using our efficient proximity query algorithm, we show that the oracle construction time, oracle size, and *kNN* query time of *RC-Oracle* using our efficient proximity query algorithm is the smallest.

5.3.4 ***Case study.***

We conducted a case study on an evacuation simulation in Mount Rainier [50] (with the highest seasonal total snowfall world record [10]) due to the frequent heavy snowfall [51]. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [11] and one may be dead due to asphyxiation [37] if s/he gets buried in the snow. In the case of snowfall, staffs will evacuate tourists in the mountain to the closest hotels immediately for tourists' safety. The time of a human being buried in the snow is expected to be 2.4 hours[1]. The average distance between the viewing platforms and hotels in Mount Rainier National Park is 11.2km [6], and the average human walking speed is 5.1 km/h [9], so the evacuation (i.e., the time of human's walking from the viewing platform to hotels) can be finished in 2.2 ($= \frac{11.2\text{km}}{5.1\text{km/h}}$) hours. Thus, the calculation of the shortest paths is expected to be finished within 12 min (= 2.4 − 2.2 hours). Our experimental results show that for a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels), (1) the oracle construction time for (i) *RC-Oracle* is 200s $\approx$ 3.2 min and (ii) the best-known oracle *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours, and (2) the query time

[1]The time of a human being buried is calculated as 2.4 hours which is computed by $\frac{10\text{centimeters} \times 24\text{hours}}{1\text{meter}}$, since the maximum snowfall rate (which is defined to be the maximum amount of snow accumulates in depth during a given time [17, 53]) in Mount Rainier is 1 meter per 24 hours [52], and it becomes difficult to walk, easy to lose the trail and get buried in the snow if the snow is deeper than 10 centimeters [28].

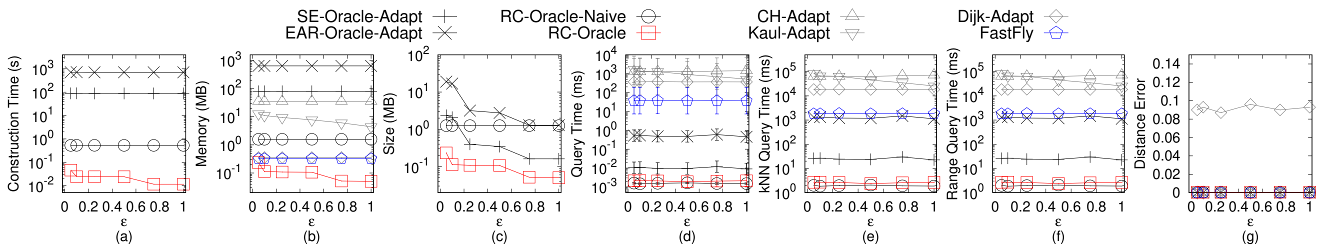


Figure 8: Baseline comparisons (effect of ϵ on *EP_p-small* point cloud dataset for the P2P query)

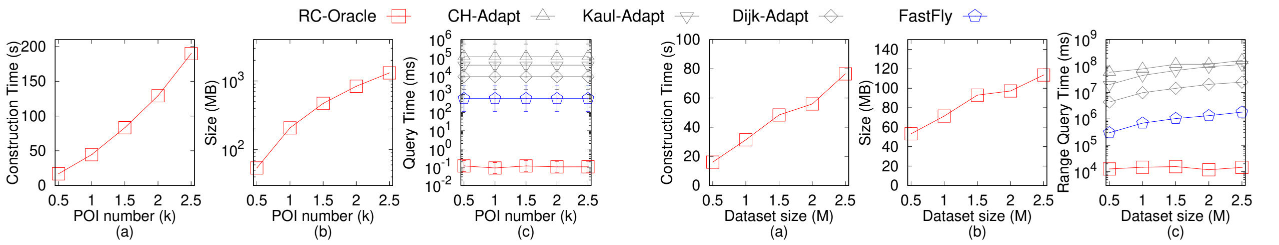


Figure 9: Baseline comparisons (effect of n on *GF_p* point cloud dataset for the P2P query)

Figure 10: Baseline comparisons (effect of N on *LM_p* point cloud dataset for the P2P query)

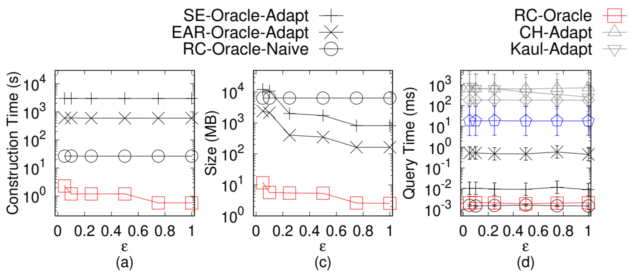


Figure 11: Baseline comparisons on *EP_p* point cloud dataset for the A2A query

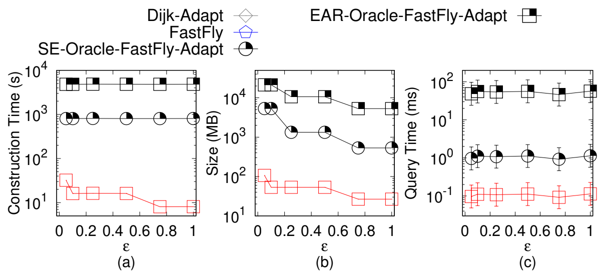


Figure 12: Ablation study on *RM_p* point cloud dataset for the P2P query

for calculating 10 nearest hotels of each viewing platform for (i) *RC-Oracle* is 6s, (ii) *SE-Oracle-Adapt* is 75s, and (iii) the best-known on-the-fly approximate shortest surface path query algorithm *Kaul-Adapt* is 80,500s ≈ 22.5 hours. Thus, *RC-Oracle* is the best one in the evacuation since 3.2 min + 6s ≤ 12 min. *RC-Oracle* also supports real-time responses, i.e., it can construct the oracle in 0.4s and answer the *kNN* query and range query in both 7 ms on a point cloud with 10k points and 250 POIs.

5.3.5 ***Summary***. In terms of the oracle construction time, oracle size, and shortest path query time, *RC-Oracle* is up to 390 times, 30 times, and 6 times better than the best-known oracle *SE-Oracle-Adapt* for the P2P query on a point cloud, and up to 500 times, 140 times, and 50 times better than the best-known oracle *EAR-Oracle-Adapt* for the A2A query on a point cloud. With the assistance of *RC-Oracle*, our algorithms for the *kNN* and range query are both up to 6 times faster than *SE-Oracle-Adapt* and up to 100 times faster than *EAR-Oracle-Adapt*. For the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time, oracle size, and all POIs *kNN* query time for *RC-Oracle* is 200s ≈ 3.2 min, 50MB, and 12.5s, but the values are 78,000s ≈ 21.7 hours, 1.5GB, and 150s for the best-known oracle *SE-Oracle-Adapt*, respectively. For the A2A query on a point cloud with 100k points and 5000 objects, the oracle construction time, oracle size, and all POIs *kNN* query time for *RC-Oracle* is 100s ≈ 1.6 min, 150MB, and 0.25s, but the values are 50,000s ≈ 13.9 hours, 21GB, and 25s for the best-known oracle *EAR-Oracle-Adapt*, respectively.

6 CONCLUSION

In our paper, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle on a point cloud called *RC-Oracle*, which has a good performance (in terms of the oracle construction time, oracle size, and shortest path query time) compared with the best-known oracle. With the assistance of *RC-Oracle*, we propose algorithms for answering other proximity queries, i.e., the *kNN* and range query. For the future work, we can explore how to build an index designed for the *kNN* and range query for better performance.

REFERENCES

[1] 2022. *Blender.* https://www.blender.org

[2] 2022. *Google Earth.* https://earth.google.com/web

[3] 2022. *Robinson Mountain.* https://www.mountaineers.org/activities/routes-places/robinson-mountain

[4] 2023. *Cyberpunk 2077.* https://www.cyberpunk.net

[5] 2023. *Data Geocomm.* http://data.geocomm.com/

[6] 2023. *Google Map.* https://www.google.com/maps

[7] 2023. *Gunnison National Forest.* https://gunnisoncrestedbutte.com/visit/places-to-go/parks-and-outdoors/gunnison-national-forest/

[8] 2023. *Laramie Mountain.* https://www.britannica.com/place/Laramie-Mountains

[9] 2023. *Preferred walking speed.* https://en.wikipedia.org/wiki/Preferred_walking_speed

[10] 2023. *Snow.* https://en.wikipedia.org/wiki/Snow

[11] Mithil Aggarwal. 2022. *More than 60 killed in blizzard wreaking havoc across U.S.* https://www.cnbc.com/2022/12/26/death-toll-rises-to-at-least-55-as-freezing-temperatures-and-heavy-snow-wallop-swaths-of-us.html

[12] Gergana Antova. 2019. Application of areal change detection methods using point clouds data. In *IOP Conference Series: Earth and Environmental Science*, Vol. 221. IOP Publishing, 012082.

[13] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M Paixao, Filipe Mutz, et al. 2021. Self-driving cars: A survey. *Expert Systems with Applications* 165 (2021), 113816.

[14] Paul B Callahan and S Rao Kosaraju. 1995. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)* 42, 1 (1995), 67–90.

[15] Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. 2007. Global path planning on board the mars exploration rovers. In *2007 IEEE Aerospace Conference.* IEEE, 1–11.

[16] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG.* New York, NY, USA, 360–369.

[17] The Conversation. 2022. *How is snowfall measured? A meteorologist explains how volunteers tally up winter storms.* https://theconversation.com/how-is-snowfall-measured-a-meteorologist-explains-how-volunteers-tally-up-winter-storms-175628

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.

[19] Yaodong Cui, Ren Chen, Wenbo Chu, Long Chen, Daxin Tian, Ying Li, and Dongpu Cao. 2021. Deep learning for image and point cloud fusion in autonomous driving: A review. *IEEE Transactions on Intelligent Transportation Systems* 23, 2 (2021), 722–739.

[20] Ke Deng, Heng Tao Shen, Kai Xu, and Xuemin Lin. 2006. Surface k-NN query processing. In *22nd International Conference on Data Engineering (ICDE'06).* IEEE, 78–78.

[21] Ke Deng and Xiaofang Zhou. 2004. Expansion-based algorithms for finding single pair shortest path on surface. In *International Workshop on Web and Wireless Geographical Information Systems.* Springer, 151–166.

[22] Ke Deng, Xiaofang Zhou, Heng Tao Shen, Qing Liu, Kai Xu, and Xuemin Lin. 2008. A multi-resolution surface distance model for k-nn query processing. *The VLDB Journal* 17, 5 (2008), 1101–1119.

[23] Brett G Dickson and P Beier. 2007. Quantifying the influence of topographic position on cougar (Puma concolor) movement in southern California, USA. *Journal of Zoology* 271, 3 (2007), 270–277.

[24] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[25] David Eriksson and Evan Shellshear. 2014. Approximate distance queries for path-planning in massive point clouds. In *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, Vol. 2. IEEE, 20–28.

[26] David Eriksson and Evan Shellshear. 2016. Fast exact shortest distance queries for massive point clouds. *Graphical Models* 84 (2016), 28–37.

[27] Mingyu Fan, Hong Qiao, and Bo Zhang. 2009. Intrinsic dimension estimation of manifolds by incising balls. *Pattern Recognition* 42, 5 (2009), 780–787.

[28] Fresh Off The Grid. 2022. *Winter Hiking 101: Everything you need to know about hiking in snow.* https://www.freshoffthegrid.com/winter-hiking-101-hiking-in-snow/

[29] Anupam Gupta, Robert Krauthgamer, and James R Lee. 2003. Bounded geometries, fractals, and low-distortion embeddings. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 534–543.

[30] Bo Huang, Victor Junqiu Wei, Raymond Chi-Wing Wong, and Bo Tang. 2023. EAR-Oracle: on efficient indexing for distance queries between arbitrary points on terrain surface. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[31] GreenValley International. 2023. *3D Point Cloud Data and the Production of Digital Terrain Models.* https://geo-matching.com/content/3d-point-cloud-data-and-the-production-of-digital-terrain-models

[32] Manohar Kaul, Raymond Chi-Wing Wong, and Christian S Jensen. 2015. New lower and upper bounds for shortest distance queries on terrains. *Proceedings of the VLDB Endowment* 9, 3 (2015), 168–179.

[33] Manohar Kaul, Raymond Chi-Wing Wong, Bin Yang, and Christian S Jensen. 2013. Finding shortest paths on terrains by killing two birds with one stone. *Proceedings of the VLDB Endowment* 7, 1 (2013), 73–84.

[34] Balázs Kégl. 2002. Intrinsic dimension estimation using packing numbers. *Advances in neural information processing systems* 15 (2002).

[35] Marcel Körtgen, Gil-Joo Park, Marcin Novotni, and Reinhard Klein. 2003. 3D shape matching with 3D shape contexts. In *The 7th central European seminar on computer graphics*, Vol. 3. Citeseer, 5–17.

[36] Baki Koyuncu and Erkan Bostancı. 2009. 3D battlefield modeling and simulation of war games. *Communications and Information Technology proceedings* (2009).

[37] Russell LaDuca. 2020. *What would happen to me if I was buried under snow?* https://qr.ae/prt6zQ

[38] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.

[39] Lik-Hang Lee, Tristan Braud, Pengyuan Zhou, Lin Wang, Dianlei Xu, Zijun Lin, Abhishek Kumar, Carlos Bermejo, and Pan Hui. 2021. All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda. *arXiv preprint arXiv:2110.05352* (2021).

[40] Lik-Hang Lee, Zijun Lin, Rui Hu, Zhengya Gong, Abhishek Kumar, Tangyao Li, Sijia Li, and Pan Hui. 2021. When creators meet the metaverse: A survey on computational arts. *arXiv preprint arXiv:2111.13486* (2021).

[41] Ying Li, Lingfei Ma, Zilong Zhong, Fei Liu, Michael A Chapman, Dongpu Cao, and Jonathan Li. 2020. Deep learning for lidar point clouds in autonomous driving: A review. *IEEE Transactions on Neural Networks and Learning Systems* 32, 8 (2020), 3412–3432.

[42] Lian Liu and Raymond Chi-Wing Wong. 2011. Finding shortest path on land surface. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data.* 433–444.

[43] Anders Mårell, John P Ball, and Annika Hofgaard. 2002. Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights. *Canadian Journal of Zoology* 80, 5 (2002), 854–865.

[44] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (1987), 647–668.

[45] Geo Week News. 2022. *Tesla using radar to generate point clouds for autonomous driving.* https://www.geoweeknews.com/news/tesla-using-radar-generate-point-clouds-autonomous-driving

[46] Hoong Kee Ng, Hon Wai Leong, and Ngai Lam Ho. 2004. Efficient algorithm for path-based range query in spatial databases. In *Proceedings. International Database Engineering and Applications Symposium, 2004. IDEAS'04.* IEEE, 334–343.

[47] Janet E Nichol, Ahmed Shaker, and Man-Sing Wong. 2006. Application of high-resolution stereo satellite images to detailed landslide hazard assessment. *Geomorphology* 76, 1-2 (2006), 68–75.

[48] Sebastian Pütz, Thomas Wiemann, Jochen Sprickerhof, and Joachim Hertzberg. 2016. 3d navigation mesh generation for path planning in uneven terrain. *IFAC-PapersOnLine* 49, 15 (2016), 212–217.

[49] Fabio Remondino. 2003. From point cloud to surface: the modeling and visualization problem. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 34 (2003).

[50] National Park Service. 2022. *Mount Rainier.* https://www.nps.gov/mora/index.htm

[51] National Park Service. 2022. *Mount Rainier Annual Snowfall Totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm

[52] National Park Service. 2022. *Mount Rainier Frequently Asked Questionss.* https://www.nps.gov/mora/faqs.htm

[53] National Weather Service. 2023. *Measuring Snow.* https://www.weather.gov/dvn/snowmeasure

[54] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1020–1031.

[55] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. 2006. Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. In *European conference on computer vision.* Springer, 1–15.

[56] Barak Sober, Robert Ravier, and Ingrid Daubechies. 2020. Approximating the riemannian metric from point clouds via manifold moving least squares. *arXiv preprint arXiv:2007.09885* (2020).

[57] Spatial. 2022. *LiDAR Scanning with Spatial's iOS App.* https://support.spatial.io/hc/en-us/articles/360057387631-LiDAR-Scanning-with-Spatial-s-iOS-App

[58] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *SIGMOD/PODS'17.* New York, NY, USA, 1211–1226.

[59] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2022. Proximity queries on terrain surface. *ACM Transactions on Database Systems (TODS)* (2022).

[60] Shi-Qing Xin and Guo-Jin Wang. 2009. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics* 28, 4 (2009), 1–8.

[61] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1114–1125.

[62] Da Yan, Zhou Zhao, and Wilfred Ng. 2012. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *Proceedings of the 21st ACM international conference on Information and knowledge management.* 942–951.
[63] Yinzhao Yan and Raymond Chi-Wing Wong. 2021. Path Advisor: a multi-functional campus map tool for shortest path. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2683–2686.
[64] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering* 2014 (2014).

A SUMMARY OF ALL NOTATIONS

Table 4 shows a summary of all notations.

Table 4: Summary of all notations

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
$N(p)$	A set of neighbor points of p
$d_E(p, p')$	The Euclidean distance between point p and p'
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
M_{path}	The path map table
M_{POI}	The POI map table
P_{remain}	A set of remaining POIs of P on C that we have not used algorithm *FastFly* to calculate the exact shortest path passing on C with $p_i \in P_{remain}$ as source
$P_{dest}(q)$	A set of POIs of P on C that we need to use algorithm *FastFly* to calculate the exact shortest path passing on C from q to $p_i \in P_{dest}(q)$ as destinations
T	The *TIN* constructed by C
h	The height of the compressed partition tree
β	The largest capacity dimension
θ	The minimum inner angle of any face in T
l_{max}/l_{min}	The length of the longest / shortest edge of T
λ	The number of highway nodes covered by a minimum square
ξ	The square root of the number of boxes
m	The number of Steiner points per face
$\Pi^*(s, t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s, t\|C)\|$	The length of $\Pi^*(s, t\|C)$
$\Pi(s, t\|C)$	The shortest path passing on C between s and t returned by *RC-Oracle*
$\Pi^*(s, t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_N(s, t\|T)$	The shortest network path passing on T between s and t
$\Pi_E(s, t\|T)$	The shortest path passing on the edges of T between s and t where these edges belongs to the faces that $\Pi^*(s, t\|T)$ passes

B COMPARISON OF ALL ALGORITHMS

Table 5 shows a comparison of all algorithms (support the shortest path query) in terms of the oracle construction time, oracle size, and shortest path query time, and Table 6 shows a comparison of *RC-Oracle* and oracle designed for other proximity queries in terms of the oracle construction time, oracle size, and *kNN* query time.

C AR2AR QUERY ON *TINS*

Apart from the P2P query on *TIN*s that we discussed in the main body of this paper, we also present an oracle to answer the AR2AR query on *TIN*s based on *RC-Oracle-Adapt*. This adapted oracle is similar to the one presented in Section 4, but there are two differences. The first difference is that we need to create POIs which has the same coordinate values as all vertices in the *TIN*. The second difference is that the source point s or the destination point t may lie on the faces of a *TIN*. There are three cases: (1) both s and t lie on the vertices of the *TIN*, (2) both s and t lie on the faces of the *TIN*, and (3) either s or t lies on the faces of the *TIN*. (1) For the first case, after creating POIs which has the same coordinate values as all vertices in the *TIN*, *RC-Oracle-Adapt* can answer the AR2AR query. (2) For the second case, we denote the face that s lies in to be f_s and the face that t lies in to be f_t. We denote the set of three vertices of f_s to be V_s, and the set of three vertices of f_t to be V_t. After creating POIs which has the same coordinate values as all vertices in the *TIN*, we need to find the shortest path between each vertex $u \in V_s$ and each vertex $v \in V_t$, then concatenate the line segment (s, u) and (v, t) with the path. After calculating nine paths, we select the path with the smallest distance as the result path. (3) For the third case, it is similar to the second case. When s lies on the vertices of the *TIN* and t lies on the faces of the *TIN*, we set $V_s = \{s\}$. When t lies on the vertices of the *TIN* and s lies on the faces of the *TIN*, we set $V_t = \{t\}$. Then, we can use the second case to answer the shortest path between s and t.

In general, the number of POI becomes N. Thus, for the AR2AR query on *TIN*s, the oracle construction time, oracle size, and shortest path query time of *RC-Oracle-Adapt* are $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$, and $O(1)$, respectively. For the AR2AR query on *TIN*s, *RC-Oracle-Adapt* always has $|\Pi(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ for each pair of vertices s and t on the faces T, where $\Pi_{adapt}(s, t|T)$ is the calculated shortest path of *RC-Oracle-Adapt* passing on a conceptual graph of T between s and t, where the vertices of this conceptual graph are formed by the vertices of T, and the edges of this graph are formed by adding edges between each vertex and its 8 neighbor vertices, and $\Pi^*_{adapt}(s, t|T)$ is the exact shortest path passing on this conceptual graph between s and t. This is because we let $p \in V_s$ and $q \in V_t$ be two vertices that lie on the path $\Pi_{adapt}(s, t|T)$, so $|\Pi_{adapt}(s, t|T)| = |(s, p)| + |\Pi_{adapt}(p, q|T)| + |(q, t)| \leq |(s, p')| + |\Pi_{adapt}(p', q'|T)| + |(q', t)|$. We let $p' \in V_s$ and $q' \in V_t$ be two vertices that lie on the path $\Pi^*_{adapt}(s, t|T)$, so $|\Pi^*_{adapt}(s, t|T)| = |(s, p')| + |\Pi^*_{adapt}(p', q'|T)| + |(q', t)|$. Since *RC-Oracle-Adapt* always has $|\Pi_{adapt}(p', q'|T)| \leq (1 + \epsilon)|\Pi^*_{adapt}(p', q'|T)|$, we obtain $|\Pi_{adapt}(s, t|T)| = |(s, p)| + |\Pi_{adapt}(p, q|T)| + |(q, t)| \leq |(s, p')| + |\Pi_{adapt}(p', q'|T)| + |(q', t)| \leq |(s, p')| + (1 + \epsilon)|\Pi^*_{adapt}(p', q'|T)| + |(q', t)| \leq (1+\epsilon)|(s, p')| + (1+\epsilon)|\Pi^*_{adapt}(p', q'|T)| + (1+\epsilon)|(q', t)| = (1 + \epsilon)|\Pi^*_{adapt}(s, t|T)|$. The query time and error rate of both the AR2AR *kNN* and range query by using *RC-Oracle-Adapt* are $O(N)$ and $1 + \epsilon$, respectively.

Table 5: Comparison of all algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error
Oracle-based algorithm							
SE-Oracle-Adapt [58, 59]	$O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
SE-Oracle-FastFly-Adapt [58, 59]	$O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Medium	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
EAR-Oracle-Adapt [30]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi \log(\lambda\xi))$	Medium	Small
EAR-Oracle-FastFly-Adapt [30]	$O(\lambda\xi mN \log N + \frac{N \log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$	Medium	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi \log(\lambda\xi))$	Medium	Small
RC-Oracle-Naive	$O(nN \log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Small
RC-Oracle **(ours)**	$\boldsymbol{O(\frac{N \log N}{\epsilon} + n \log n)}$	**Small**	$\boldsymbol{O(\frac{n}{\epsilon})}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Small**
On-the-fly algorithm							
CH-Adapt [16]	-	N/A	-	N/A	$O(N^2)$	Large	Small
Kaul-Adapt [32]	-	N/A	-	N/A	$O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large	Small
Dijk-Adapt [33]	-	N/A	-	N/A	$O(N \log N)$	Medium	Medium
FastFly **(ours)**	-	N/A	-	N/A	$\boldsymbol{O(N \log N)}$	**Medium**	**No error**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [27, 34], λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, θ is the minimum inner angle of any face in T, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T.

Table 6: Comparison of *RC-Oracle* and oracle designed for other proximity queries on a point cloud

Algorithm	Oracle construction time		Oracle size		*kNN* query time		Error
VO-Oracle-Adapt [54]	$O(N^2 \log N)$	Large	$O(N)$	Medium	$O(N \log^2 N)$	Small	No error
RC-Oracle **(ours)**	$\boldsymbol{O(\frac{N \log N}{\epsilon} + n \log n)}$	**Small**	$\boldsymbol{O(\frac{n}{\epsilon})}$	**Small**	$\boldsymbol{O(n')}$	**Tiny**	**Small**

Remark: n' is the number of query objects.

D EMPIRICAL STUDIES

D.1 Experimental Results for *TIN*s

D.1.1 ***Baseline comparisons for the P2P query.*** We study the P2P query on *TIN*s. We (1) compared *SE-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets with default 500 POIs. The *kNN* query error and range query error are all equal to 0 for all experiments (since the distance error is very small), so their results are omitted.

Effect of ϵ. In Figure 5, Figure 14, Figure 17, Figure 19 and Figure 21, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting N to be 10k and n to be 50. In Figure 23, Figure 26, Figure 29, Figure 32 and Figure 35, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle-Adapt* a lot, the oracle construction time, memory consumption, oracle size, shortest path query time, all POIs *kNN* query time, and all POIs range query time of *RC-Oracle-Adapt* still perform much better than the best-known oracle *SE-Oracle*, and other algorithms / oracles.

Effect of n. In Figure 13, Figure 15, Figure 18, Figure 20 and Figure 22, we tested 5 values of n from {50, 100, 150, 200, 250} on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 24, Figure 27, Figure 30, Figure 33 and Figure 36, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time for *SE-Oracle* is large compared with *RC-Oracle-Adapt*, which shows the superior performance of *RC-Oracle-Adapt* in terms of the oracle construction and shortest path querying.

Effect of N **(scalability test)**. In Figure 16, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_t*-small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 25, Figure 28, Figure 31, Figure 34 and Figure 37, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle-Adapt* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size, and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *CH*, and the distance error of *FastFly-Adapt* (with distance error close to 0) is much smaller than that of *Dijk* (with distance error 0.03).

D.1.2 ***Baseline comparisons for the AR2AR query.*** We study the AR2AR query on *TIN*s. We compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt*.

In Figure 38, we tested the AR2AR query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a small-version of EP_t dataset. We selected 50 points as reference points for the *kNN* and range query. Even though varying ϵ will not affect *RC-Oracle-Adapt* a lot, the oracle construction time, memory consumption, oracle size, shortest path query time, all POIs *kNN* query time, and all POIs range query time of *RC-Oracle-Adapt* still perform much better than the best-known oracle *SE-Oracle*, and other adapted algorithms / oracles. The oracle construction time of *EAR-Oracle* is

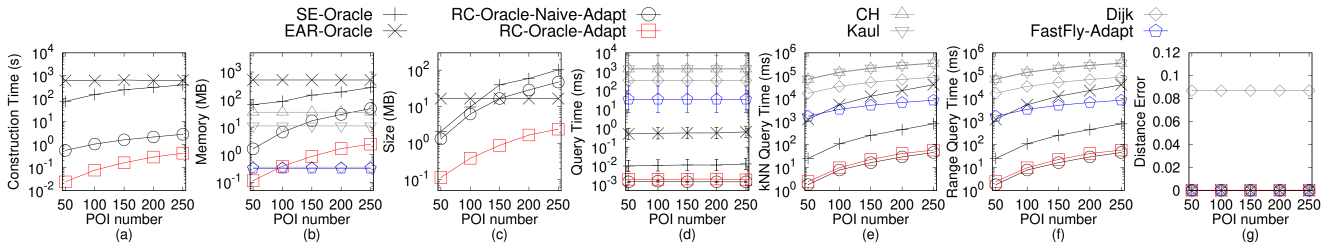


Figure 13: Baseline comparisons (effect of n on BH_t-small TIN dataset for the P2P query)

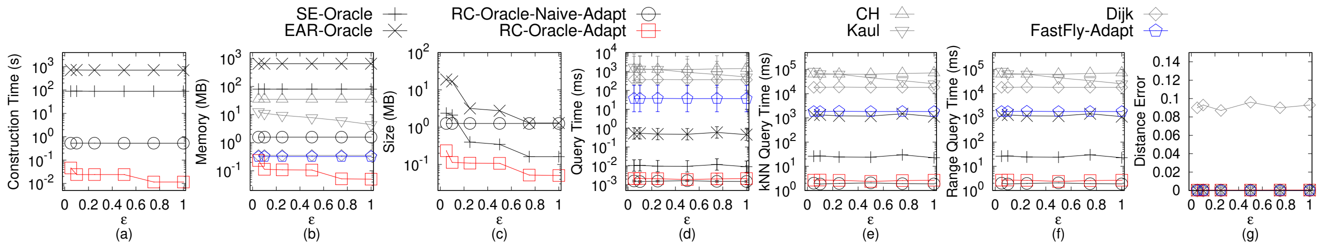


Figure 14: Baseline comparisons (effect of ϵ on EP_t-small TIN dataset for the P2P query)

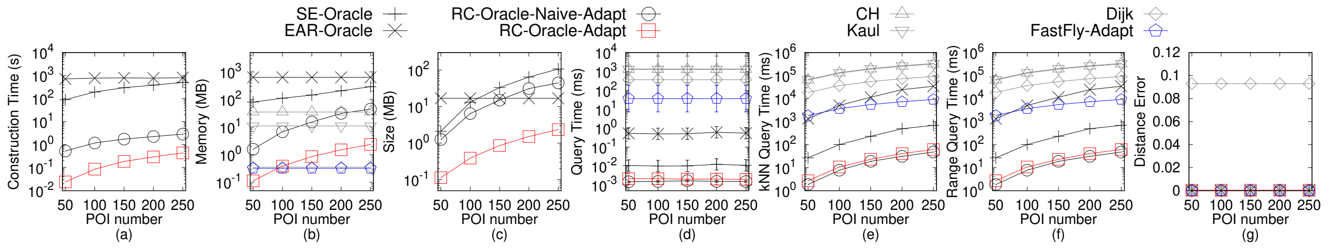


Figure 15: Baseline comparisons (effect of n on EP_t-small TIN dataset for the P2P query)

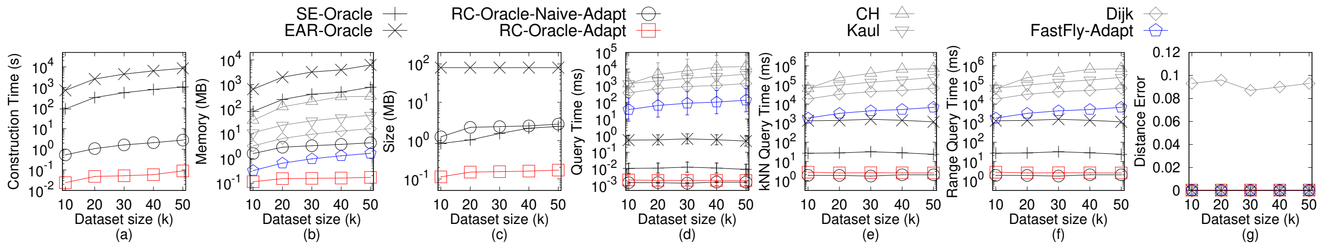


Figure 16: Baseline comparisons (effect of N on EP_t-small TIN dataset for the P2P query)

up to 10^4 times larger than that of *RC-Oracle-Adapt*, even though *EAR-Oracle* is deliberately designed for the AR2AR query on *TINs*, its performance is not well compared with *RC-Oracle-Adapt*.

D.2 Experimental Results for Point Clouds

D.2.1 ***Baseline comparisons for the P2P query.*** We study the P2P query on *point clouds* for baseline comparisons. We (1) compared *SE-Oracle-Adapt*, *SE-Oracle-FastFly-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets with default 500 POIs.

Effect of ϵ. In Figure 39, Figure 8, Figure 43, Figure 45 and Figure 47, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p-*small*, EP_p-*small*, GF_p-*small*, LM_p-*small* and RM_p-*small* dataset by setting N to be 10k and n to be 50. In Figure 49, Figure 52, Figure 55, Figure 58 and Figure 61, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle* a lot, the oracle construction time, memory consumption, oracle size, shortest path query time, all POIs *kNN* query time, and all POIs range query time of *RC-Oracle* still perform

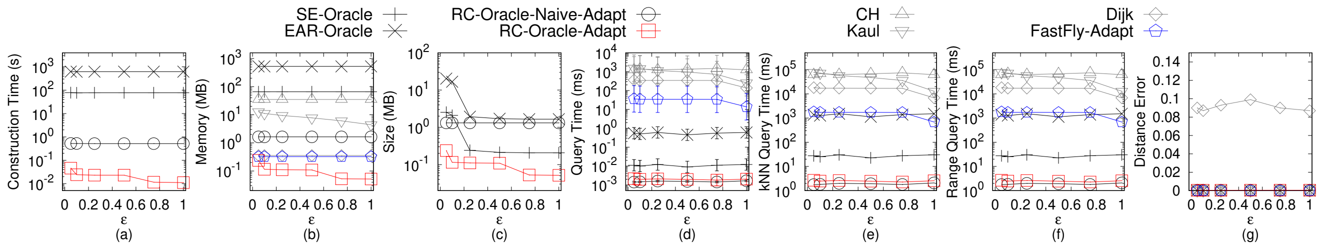


Figure 17: Baseline comparisons (effect of ϵ on GF_t-*small TIN* dataset for the P2P query)

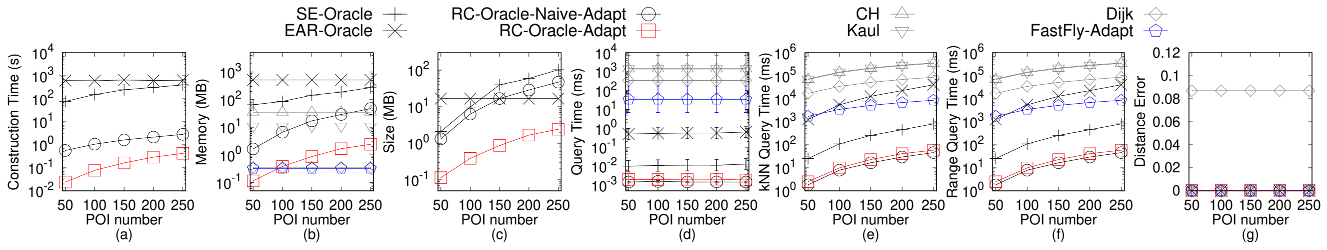


Figure 18: Baseline comparisons (effect of n on GF_t-*small TIN* dataset for the P2P query)

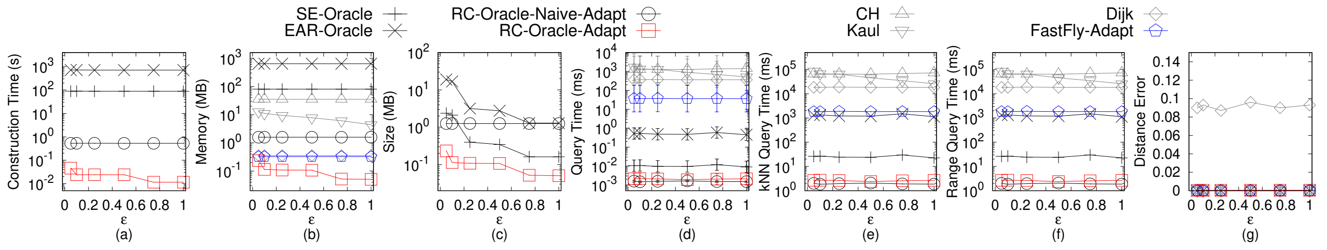


Figure 19: Baseline comparisons (effect of ϵ on LM_t-*small TIN* dataset for the P2P query)

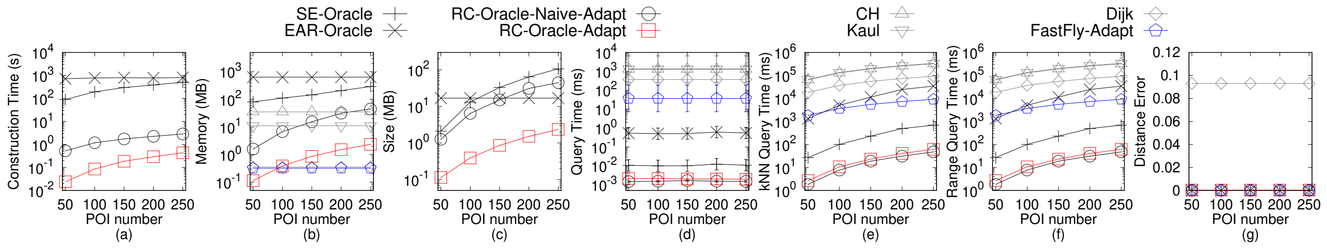


Figure 20: Baseline comparisons (effect of n on LM_t-*small TIN* dataset for the P2P query)

much better than the best-known oracle *SE-Oracle-Adapt*, and other algorithms / oracles.

Effect of n. In Figure 40, Figure 41, Figure 44, Figure 46 and Figure 48, we tested 5 values of n from {50, 100, 150, 200, 250} on BH_p-*small*, EP_p-*small*, GF_p-*small*, LM_p-*small* and RM_p-*small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 50, Figure 53, Figure 56, Figure 59 and Figure 62, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time for *SE-Oracle* is large compared with *RC-Oracle*, which shows the superior performance of *RC-Oracle* in terms of the oracle construction and shortest path querying.

Effect of N (scalability test). In Figure 42, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_p-*small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 51, Figure 54, Figure 57, Figure 60 and Figure 63, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_p-*small*, EP_p-*small*, GF_p-*small*, LM_p-*small* and RM_p-*small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size, and shortest path query time.

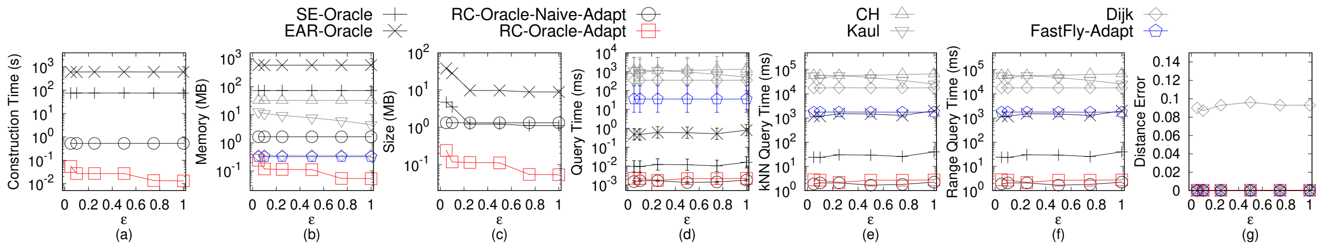


Figure 21: Baseline comparisons (effect of ϵ on *RM_t-small TIN* dataset for the P2P query)

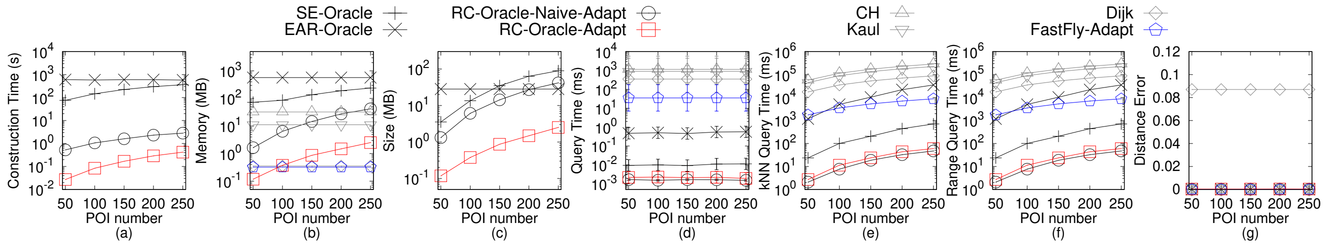


Figure 22: Baseline comparisons (effect of n on *RM_t-small TIN* dataset for the P2P query)

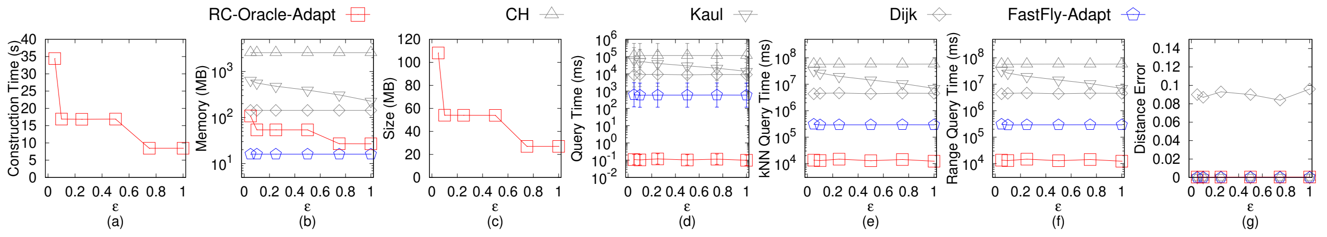


Figure 23: Baseline comparisons (effect of ϵ on *BH_t TIN* dataset for the P2P query)

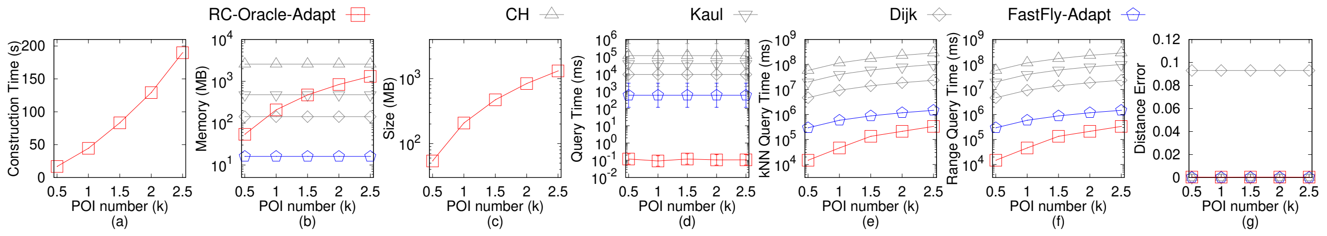


Figure 24: Baseline comparisons (effect of n on *BH_t TIN* dataset for the P2P query)

D.2.2 *Ablation study for the P2P query.*

We study the P2P query on *point clouds* for ablation study. We compared *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt*, and *RC-Oracle*.

In Figure 64, Figure 65, Figure 66, Figure 67 and Figure 68, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size, and shortest path query time of *RC-Oracle* perform better than *SE-Oracle-FastFly-Adapt* and *EAR-Oracle-FastFly-Adapt*, which shows the usefulness of the oracle part of *RC-Oracle*.

D.2.3 *Comparisons with other proximity queries oracle and algorithm for the P2P query.*

We study the P2P query on *point clouds* for comparisons with other proximity queries oracle and algorithm. We denote *RC-Oracle-NaiveProx* to be *RC-Oracle* using the naive proximity query algorithm. We compared *VO-Oracle-Adapt*, *RC-Oracle-NaiveProx*, and *RC-Oracle* with the efficient proximity query algorithm.

In Figure 69, Figure 70, Figure 71, Figure 72 and Figure 73, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size, and *kNN* query time of *RC-Oracle* perform better than *VO-Oracle-Adapt* and *RC-Oracle-NaiveProx*. Specifically, the *kNN* query time of *RC-Oracle* is 200 times smaller than that of *VO-Oracle-Adapt*. This is because the

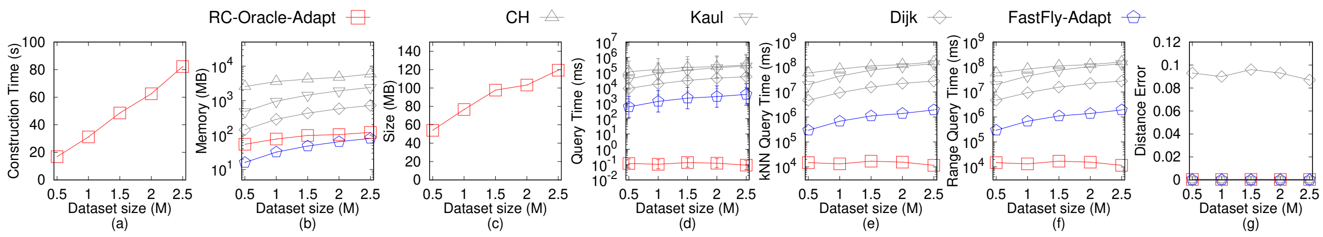


Figure 25: Baseline comparisons (effect of N on BH_t *TIN* dataset for the P2P query)

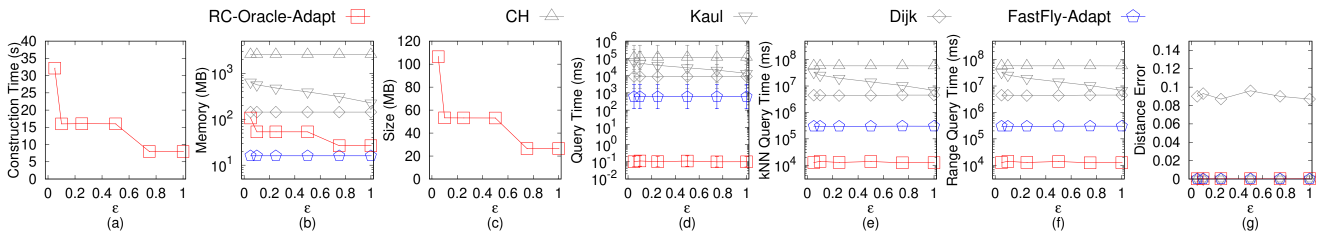


Figure 26: Baseline comparisons (effect of ϵ on EP_t *TIN* dataset for the P2P query)

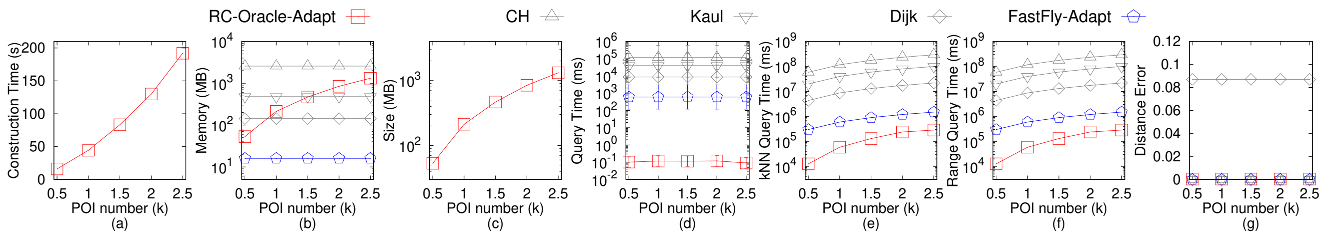


Figure 27: Baseline comparisons (effect of n on EP_t *TIN* dataset for the P2P query)

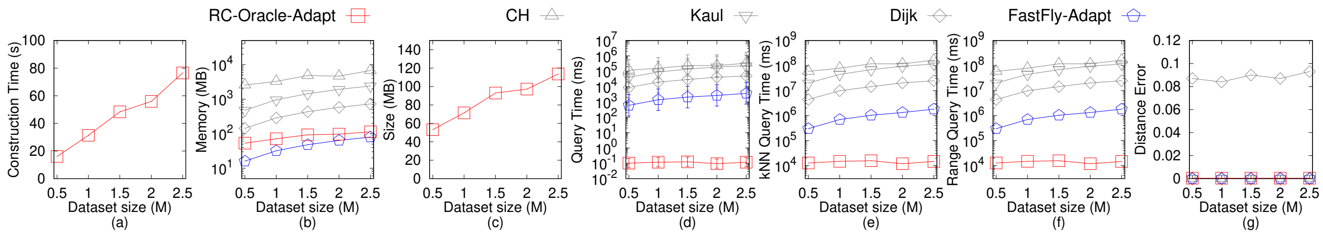


Figure 28: Baseline comparisons (effect of N on EP_t *TIN* dataset for the P2P query)

shortest path query time of *RC-Oracle* is $O(1)$, so even with the linear scan of the proximity query algorithm (in the worst case), the *kNN* query time of *RC-Oracle* is still fast.

D.2.4 ***Baseline comparisons for the A2A query***.

We study the A2A query on point clouds for baseline comparisons. We compared *SE-Oracle-Adapt*, *SE-Oracle-FastFly-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly*.

In Figure 74, we tested the A2A query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a small-version of EP_p dataset. We selected 50 points as reference points for the *kNN* and range query. Although *EAR-Oracle-Adapt* is regarded as the best-known oracle in the A2A query on a point cloud, *RC-Oracle* still performs much better than it.

D.2.5 ***Ablation study for the A2A query for ablation study***.

We study the A2A query on point clouds for ablation study. We compared *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt*, and *RC-Oracle*.

In Figure 75, we tested the A2A query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a small-version of EP_p dataset. We selected 50 points as reference points for the *kNN* and range query. *RC-Oracle* can still perform much better than *SE-Oracle-FastFly-Adapt* and *EAR-Oracle-FastFly-Adapt*.

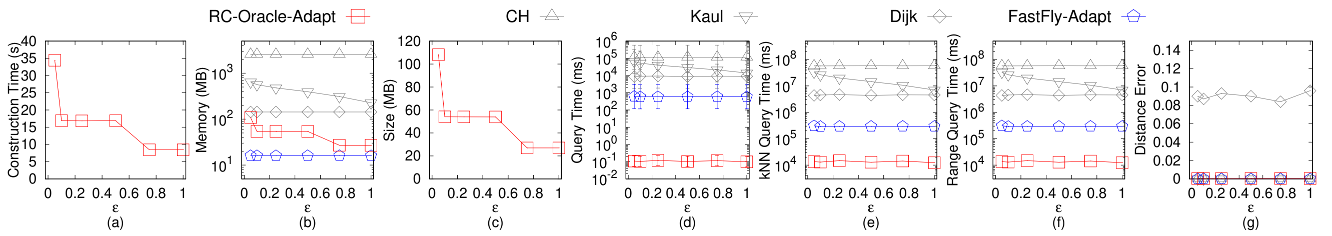


Figure 29: Baseline comparisons (effect of ϵ on GF_t *TIN* dataset for the P2P query)

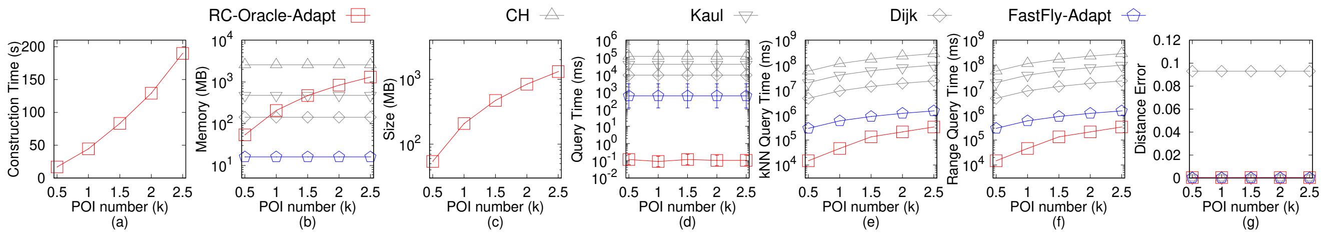


Figure 30: Baseline comparisons (effect of n on GF_t *TIN* dataset for the P2P query)

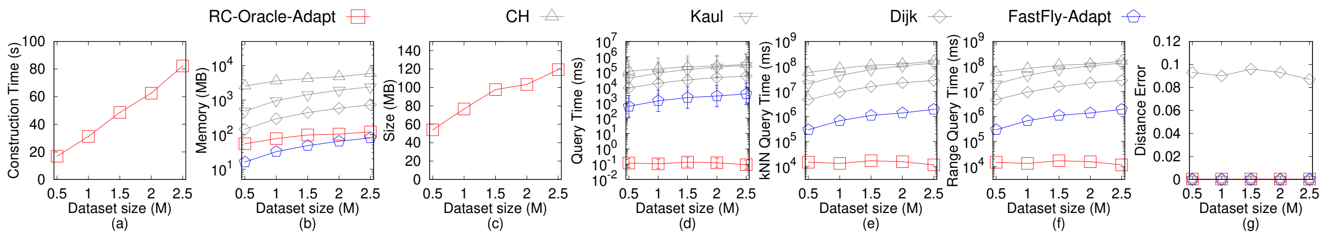


Figure 31: Baseline comparisons (effect of N on GF_t *TIN* dataset for the P2P query)

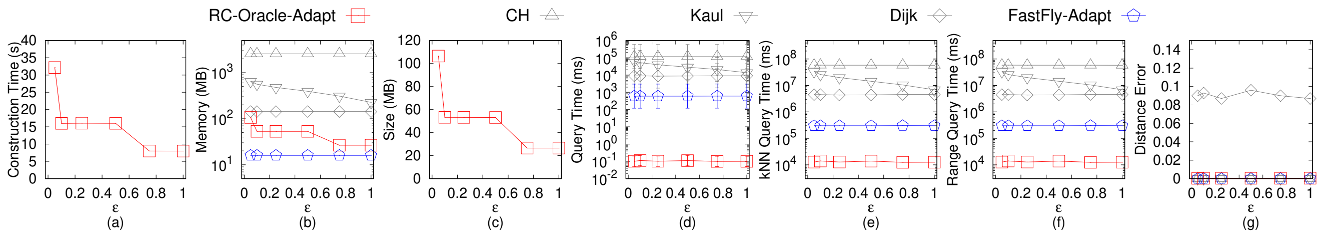


Figure 32: Baseline comparisons (effect of ϵ on LM_t *TIN* dataset for the P2P query)

D.2.6 ***Comparisons with other proximity queries oracle and algorithm for the A2A query.*** We study the A2A query on *point clouds* for comparisons with other proximity queries oracle. We compared *VO-Oracle-Adapt*, *RC-Oracle-NaiveProx*, and *RC-Oracle*.

In Figure 76, we tested the A2A query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a small-version of EP_p dataset. We selected 50 points as reference points for the *kNN* and range query. The oracle construction time, oracle size, and *kNN* query time of *RC-Oracle* perform better than *VO-Oracle-Adapt* and *RC-Oracle-NaiveProx*.

D.3 Generating Datasets with Different Dataset Sizes

The procedure for generating the point cloud datasets with different dataset sizes is as follows. We mainly follow the procedure for generating datasets with different dataset sizes in the work [42, 58, 59]. Let C_t be our target point cloud that we want to generate with qx_t points along x-coordinate, qy_t points along y-coordinate and N_t points, where $N_t = qx_t \cdot qy_t$. Let C_o be the original point cloud that we currently have with qx_o edges along x-coordinate, qy_o edges along y-coordinate and N_o points, where $N_o = qx_o \cdot qy_o$. We then generate $qx_t \cdot qy_t$ 2D points (x, y) based on a Normal distribution $N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{q_o \in C_o} x_{q_o}}{qx_o \cdot qy_o}, \overline{y} = \frac{\sum_{q_o \in C_o} y_{q_o}}{qx_o \cdot qy_o})$ and

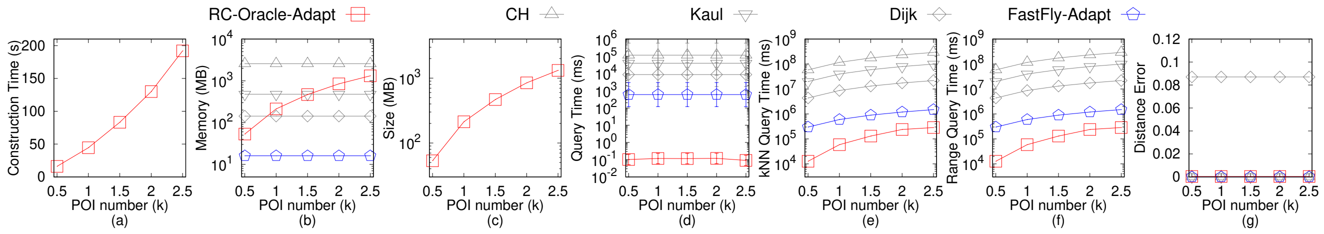


Figure 33: Baseline comparisons (effect of n on LM_t *TIN* dataset for the P2P query)

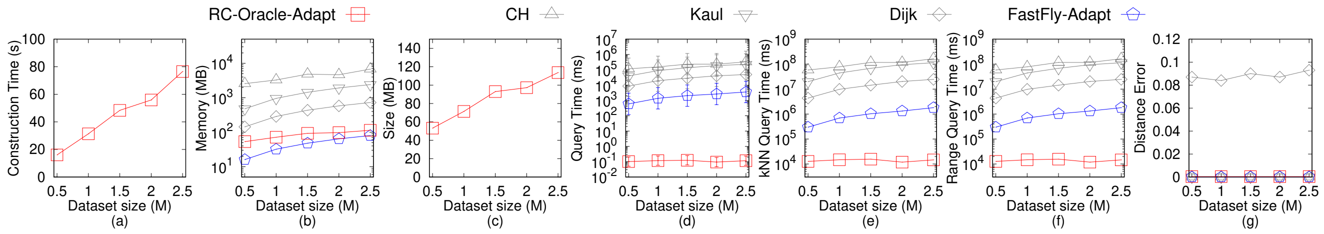


Figure 34: Baseline comparisons (effect of N on LM_t *TIN* dataset for the P2P query)

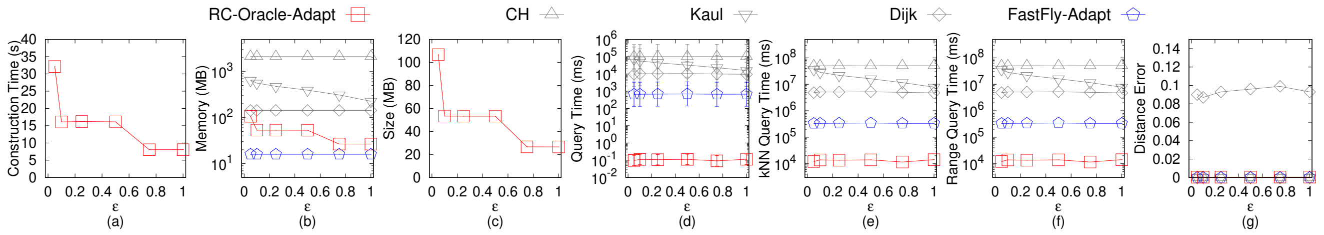


Figure 35: Baseline comparisons (effect of ϵ on RM_t *TIN* dataset for the P2P query)

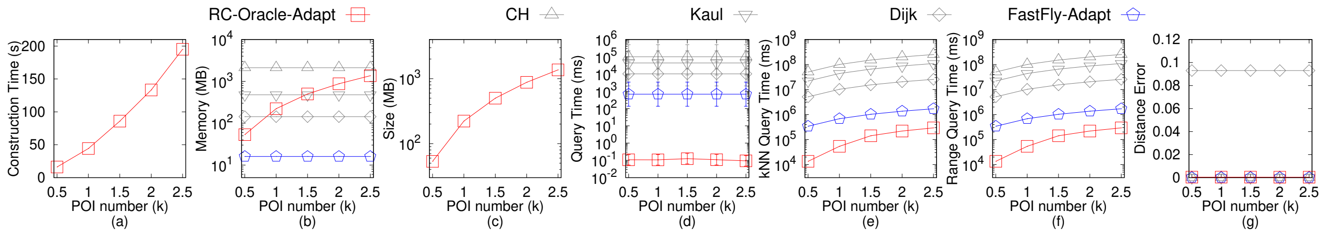


Figure 36: Baseline comparisons (effect of n on RM_t *TIN* dataset for the P2P query)

$\sigma_N^2 = (\frac{\sum_{q_O \in C_O}(x_{q_O}-\overline{x})^2}{qx_O \cdot qy_O}, \frac{\sum_{q_O \in C_O}(y_{q_O}-\overline{y})^2}{qx_O \cdot qy_O})$. In the end, we project each generated point (x, y) to the implicit surface of C_o and take the projected point as the newly generate C_t.

E PROOF

Proof of Lemma 4.4. Firstly, we prove the query time of both the *kNN* and range query algorithm. Given a query object, when we need to perform the *kNN* query or the range query, the worst case is that we need to perform a linear scan to check the distance between this query object to all other objects using the shortest path query phase of *RC-Oracle* in $O(1)$ time. This happens only if the query object is the last processed object during the construction phase of *RC-Oracle*. Since there are total n' objects, the query time is $O(n')$. However, the real query time is smaller than $O(n')$. This is because for most of the cases, we do not need to perform a linear scan (since we have already sorted some distances in order during the construction phase of *RC-Oracle*).

Secondly, we prove the error rate of both the *kNN* and range query algorithm. We give some definitions first. For simplicity, given a query POI $q \in P$, (1) we let X be a set of POIs containing the *exact* (i) k nearest POIs of q or (ii) POIs whose distance to q are at most r, calculated using the exact distance on C. Furthermore, given a query POI $q \in P$, (2) we let X' be a set of POIs containing (i) k nearest POIs of q or (ii) POIs whose distance to

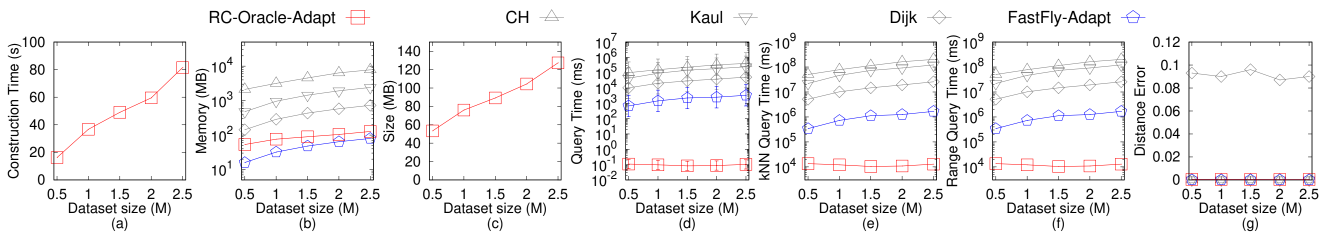


Figure 37: Baseline comparisons (effect of N on RM_t *TIN* dataset for the P2P query)

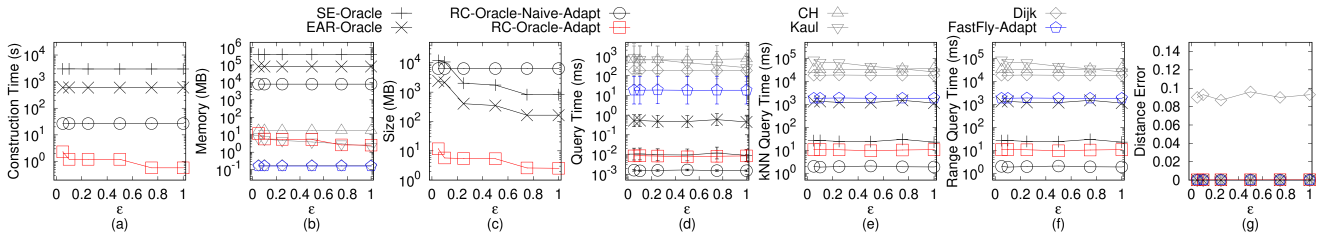


Figure 38: Baseline comparisons on EP_t *TIN* dataset for the AR2AR query

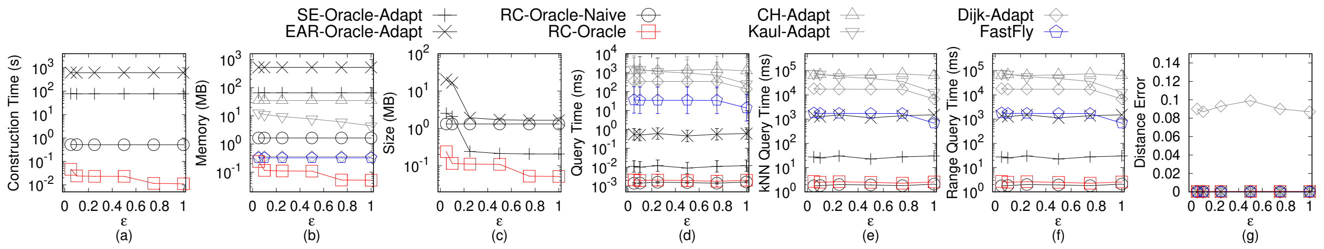


Figure 39: Baseline comparisons (effect of ϵ on BH_p*-small* point cloud dataset for the P2P query)

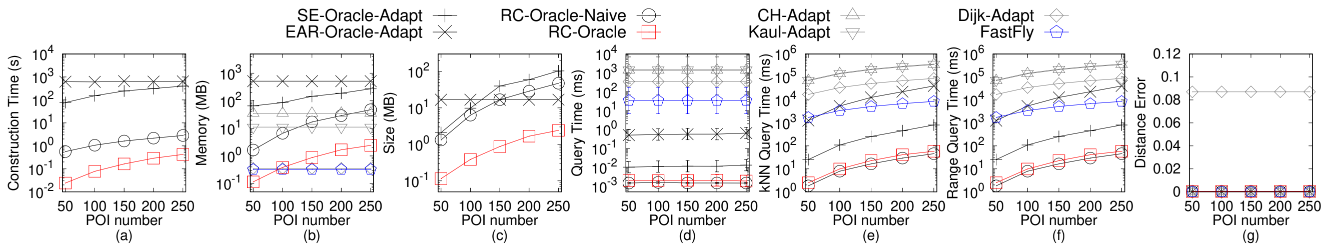


Figure 40: Baseline comparisons (effect of n on BH_p*-small* point cloud dataset for the P2P query)

q are at most r, calculated using the approximated distance on C returned by *RC-Oracle*. In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d, i.e., $X = \{c, d\}$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c, i.e., $X' = \{b, c\}$. Recall that we let v_f (resp. v'_f) be the furthest object to q in X (resp. X'), i.e., $|\Pi^*(q, v_f|C)| \leq \max_{\forall v \in X} |\Pi^*(q, v|C)|$ (resp. $|\Pi^*(q, v'_f|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q, v'|C)|$). We further let w_f (resp. w'_f) be the furthest object to q in X (resp. X') based on the approximated distance on C returned by *RC-Oracle*, i.e., $|\Pi(q, w_f|C)| \leq \max_{\forall w \in X} |\Pi(q, w|C)|$ (resp. $|\Pi(q, w'_f|C)| \leq \max_{\forall w' \in X'} |\Pi(q, w'|C)|$). Recall the error rate of the *kNN* and range query is $\alpha = \frac{|\Pi^*(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$. Since the approximated distance on C returned by *RC-Oracle* is always longer than the exact distance on C, we have $|\Pi(q, v'_f|C)| \geq |\Pi^*(q, v'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$. By the definition of v_f and w_f, we have $|\Pi^*(q, v_f|C)| \geq |\Pi^*(q, w_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v'_f|C)|}{|\Pi^*(q,w_f|C)|}$. By the definition of v'_f and w'_f, we have $|\Pi(q, v'_f|C)| \leq |\Pi(q, w'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,w'_f|C)|}{|\Pi^*(q,w_f|C)|}$. Since the error ratio of the approximated distance on C returned by *RC-Oracle* is $1 + \epsilon$, we have $|\Pi(q, w_f|C)| \leq (1 + \epsilon)|\Pi^*(q, w_f|C)|$. Then, we have $\alpha \leq$

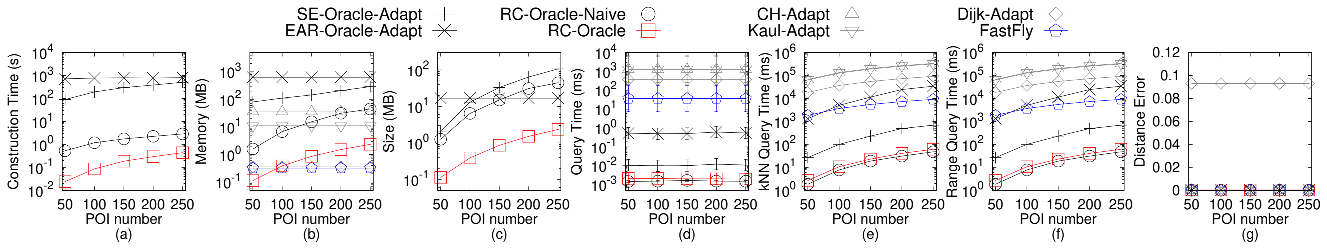


Figure 41: Baseline comparisons (effect of n on *EP_p-small* point cloud dataset for the P2P query)

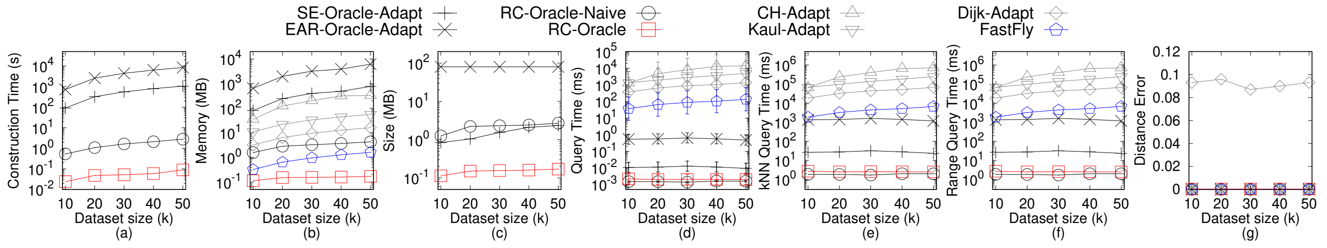


Figure 42: Baseline comparisons (effect of N on *EP_p-small* point cloud dataset for the P2P query)

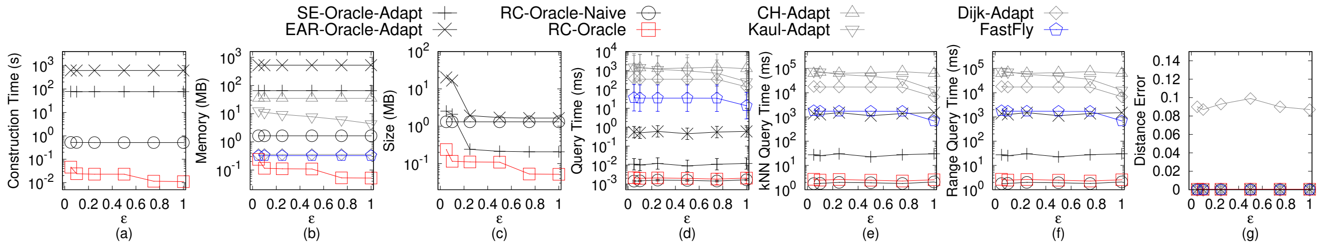


Figure 43: Baseline comparisons (effect of ϵ on *GF_p-small* point cloud dataset for the P2P query)

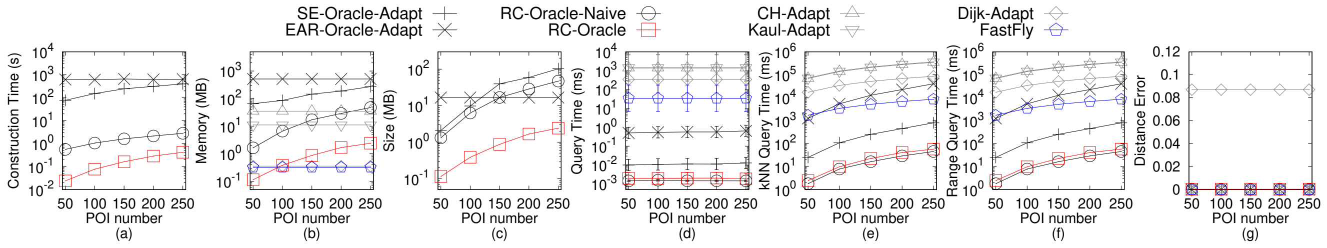


Figure 44: Baseline comparisons (effect of n on *GF_p-small* point cloud dataset for the P2P query)

$\frac{|\Pi(q,w'_f|C)|(1+\epsilon)}{|\Pi(q,w_f|C)|}$. By our *kNN* and range query algorithm, we have $|\Pi(q, w'_f|C)| \leq |\Pi(q, w_f|C)|$. Thus, we have $\alpha \leq 1 + \epsilon$. □

THEOREM E.1. *The shortest path query time and memory consumption of algorithm CH-Adapt are $O(N^2)$ and $O(N)$. Compared with $\Pi^*(s,t|T)$, algorithm CH-Adapt returns the exact shortest surface path passing on a TIN (that is constructed by the point cloud). Compared with $\Pi^*(s,t|C)$, algorithm CH-Adapt returns the approximate shortest path passing on a point cloud.*

PROOF. Firstly, we prove the *shortest path query time* of algorithm *CH-Adapt*. The proof of the shortest path query time of algorithm *CH-Adapt* is in work [16]. But since algorithm *CH-Adapt* first needs to construct the *TIN* using the point cloud, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *CH-Adapt* is $O(N + N^2) = O(N^2)$.

Secondly, we prove the *memory consumption* of algorithm *CH-Adapt*. The proof of the memory consumption of algorithm *CH-Adapt* is in work [16]. Thus, the memory consumption of algorithm *CH-Adapt* is $O(N)$.

Thirdly, we prove the *error bound* of algorithm *CH-Adapt*. Compared with $\Pi^*(s,t|T)$, the proof that algorithm *CH-Adapt* returns

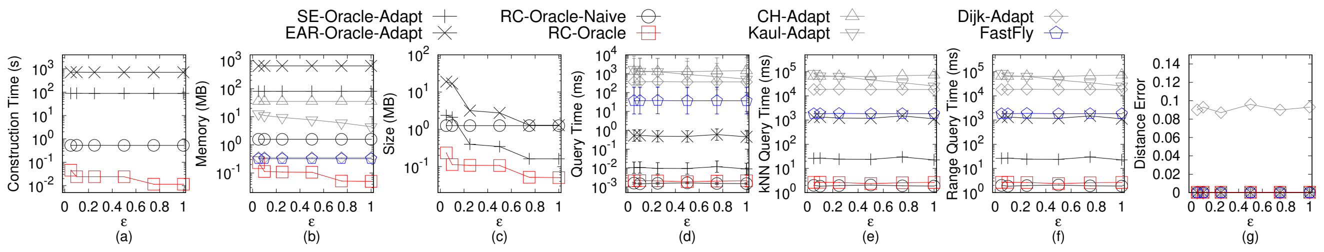


Figure 45: Baseline comparisons (effect of ϵ on *LM_p-small* point cloud dataset for the P2P query)

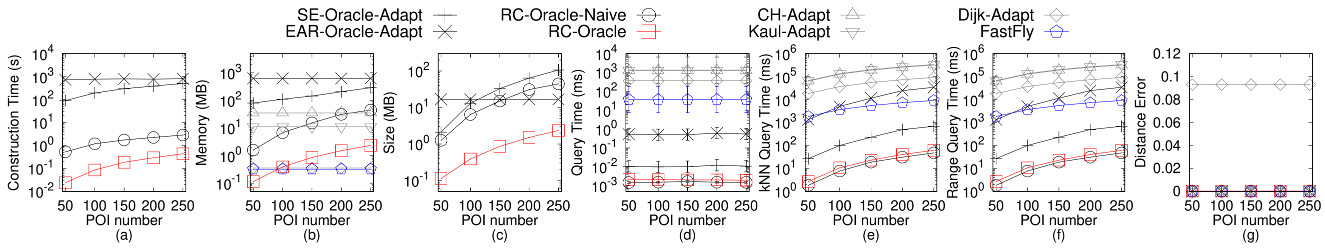


Figure 46: Baseline comparisons (effect of n on *LM_p-small* point cloud dataset for the P2P query)

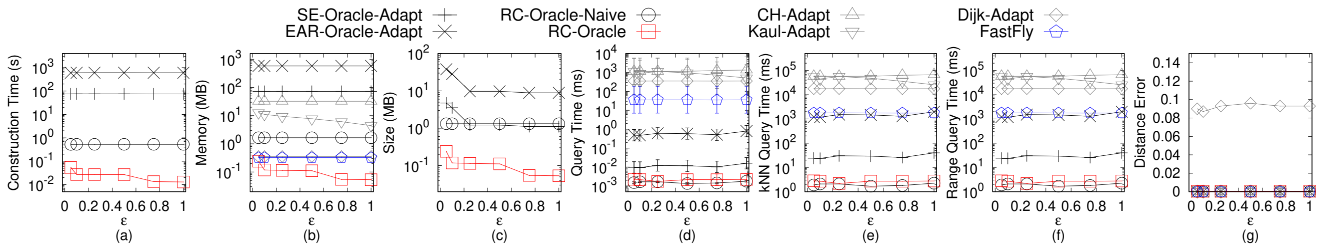


Figure 47: Baseline comparisons (effect of ϵ on *RM_p-small* point cloud dataset for the P2P query)

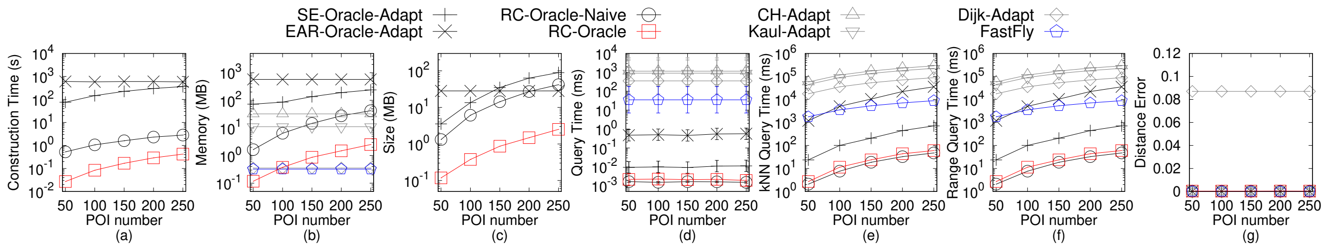


Figure 48: Baseline comparisons (effect of n on *RM_p-small* point cloud dataset for the P2P query)

the exact shortest path passing on a *TIN* is in work [16]. Since the *TIN* is constructed by the point cloud, so algorithm *CH-Adapt* returns the exact shortest surface path passing on a *TIN* (that is constructed by the point cloud). Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *CH-Adapt* returns the approximate shortest path passing on a point cloud. □

THEOREM E.2. *The shortest path query time and memory consumption of algorithm Kaul-Adapt are $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ and $O(N)$. Compared with $\Pi^*(s,t|T)$, algorithm Kaul-Adapt always has $|\Pi_{Kaul\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of vertices s and t on T, where $\Pi_{Kaul\text{-}Adapt}(s,t|T)$ is the shortest surface path of algorithm Kaul-Adapt passing on a TIN T (that is constructed by the point cloud) between s and t. Compared with $\Pi^*(s,t|C)$, algorithm Kaul-Adapt returns the approximate shortest path passing on a point cloud.*

PROOF. Firstly, we prove the *shortest path query time* of algorithm *Kaul-Adapt*. The proof of the shortest path query time of algorithm *Kaul-Adapt* is in work [32]. Note that in Section 4.2 of [32], the shortest path query time of algorithm *Kaul-Adapt* is $O((N+N')(\log(N+N')+(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where $N' =$

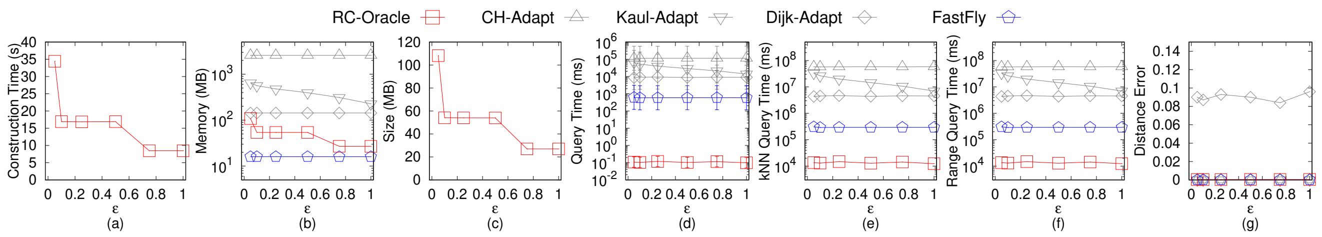


Figure 49: Baseline comparisons (effect of ϵ on BH_p point cloud dataset for the P2P query)

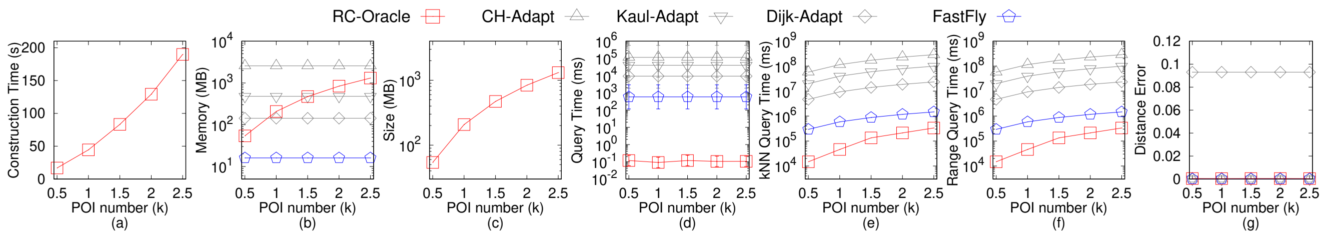


Figure 50: Baseline comparisons (effect of n on BH_p point cloud dataset for the P2P query)

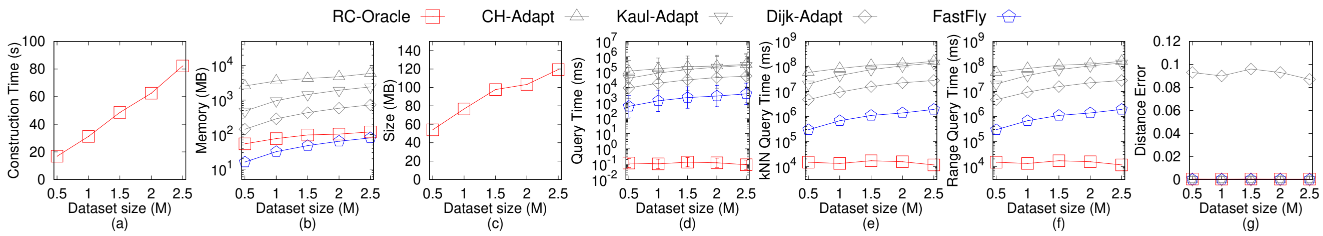


Figure 51: Baseline comparisons (effect of N on BH_p point cloud dataset for the P2P query)

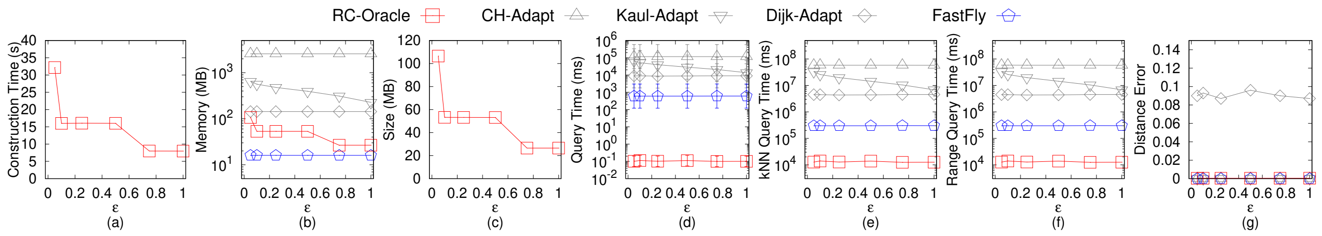


Figure 52: Baseline comparisons (effect of ϵ on EP_p point cloud dataset for the P2P query)

$O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ and K is a parameter which is a positive number at least 1. By Theorem 1 of [32], we obtain that its error bound ϵ is equal to $\frac{1}{K-1}$. Thus, we can derive that the shortest path query time of algorithm *Kaul-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + \frac{l_{max}^2}{(\epsilon l_{min}\sqrt{1-\cos\theta})^2})$. Since for N, the first term is larger than the second term, so we obtain the shortest path query time of algorithm *Kaul-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. But since algorithm *Kaul-Adapt* first needs to construct a *TIN* using the point cloud, so it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Kaul-Adapt* is $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}})) = O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$.

Secondly, we prove the *memory consumption* of algorithm *Kaul-Adapt*. Since algorithm *Kaul-Adapt* is a Dijkstra algorithm and there are total N vertices on the *TIN*, the memory consumption is $O(N)$. Thus, the memory consumption of algorithm *Kaul-Adapt* is $O(N)$.

Thirdly, we prove the *error bound* of algorithm *Kaul-Adapt*. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of algorithm *Kaul-Adapt* is in work [32]. Since the *TIN* is constructed by the point cloud, so algorithm *Kaul-Adapt* always has $|\Pi_{Kaul\text{-}Adapt}(s,t|T)| \leq$

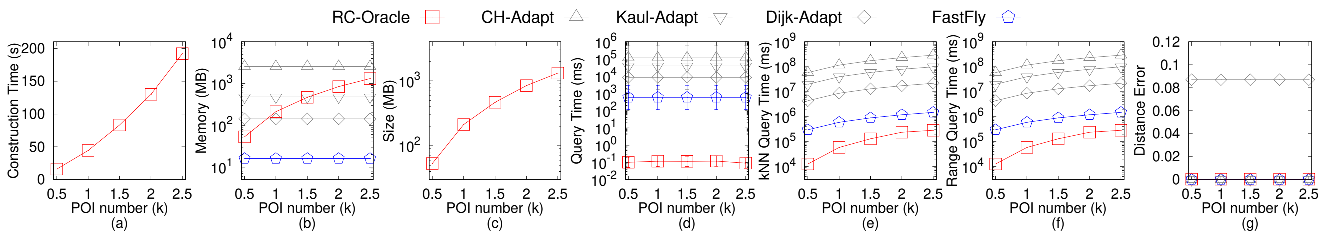


Figure 53: Baseline comparisons (effect of n on EP_p point cloud dataset for the P2P query)

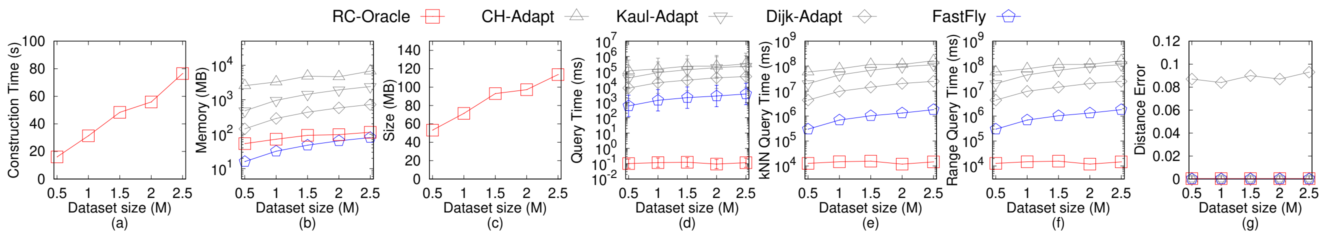


Figure 54: Baseline comparisons (effect of N on EP_p point cloud dataset for the P2P query)

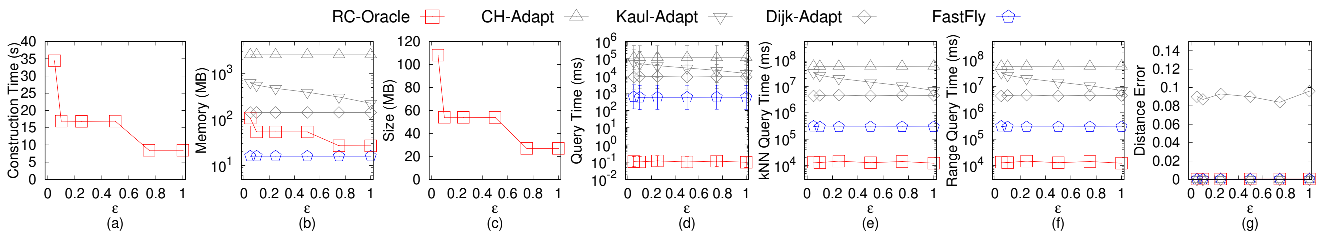


Figure 55: Baseline comparisons (effect of ϵ on GF_p point cloud dataset for the P2P query)

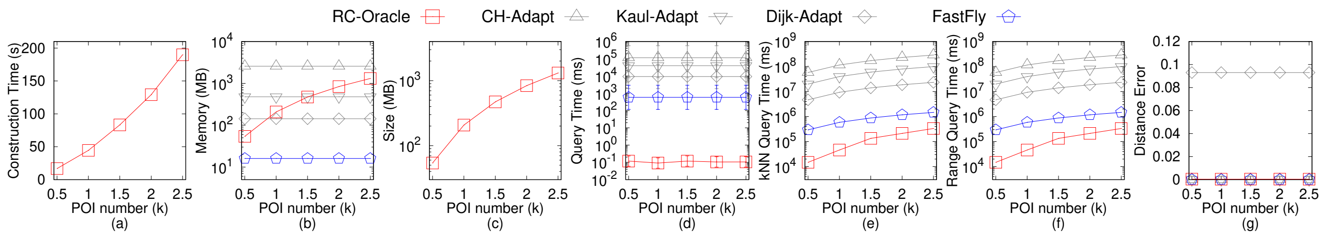


Figure 56: Baseline comparisons (effect of n on GF_p point cloud dataset for the P2P query)

$(1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of vertices s and t on T. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *Kaul-Adapt* returns the approximate shortest path passing on a point cloud. □

Theorem E.3. *The shortest path query time and memory consumption of algorithm Dijk-Adapt are $O(N \log N)$ and $O(N)$. Compared with $\Pi^*(s,t|T)$, algorithm Dijk-Adapt always has $|\Pi_{Dijk\text{-}Adapt}(s,t|T)| \leq k \cdot |\Pi^*(s,t|T)|$ for each pair of vertices s and t on T, where $\Pi_{Dijk\text{-}Adapt}(s,t|T)$ is the shortest network path of algorithm Dijk-Adapt passing on a TIN T (that is constructed by the point cloud) between s and t, $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$. Compared with $\Pi^*(s,t|C)$, algorithm Dijk-Adapt returns the approximate shortest path passing on a point cloud.*

Proof. Firstly, we prove the *shortest path query time* of algorithm *Dijk-Adapt*. Since algorithm *Dijk-Adapt* only calculates the shortest network path passing on T (that is constructed by the point cloud), it is a Dijkstra algorithm and there are total N points, so the shortest path query time is $O(N \log N)$. But since algorithm *Dijk-Adapt* first needs to construct a *TIN* using the point cloud, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Dijk-Adapt* is $O(N + N\log N) = O(N \log N)$.

Secondly, we prove the *memory consumption* of algorithm *Dijk-Adapt*. Since algorithm *Dijk-Adapt* is a Dijkstra algorithm and there

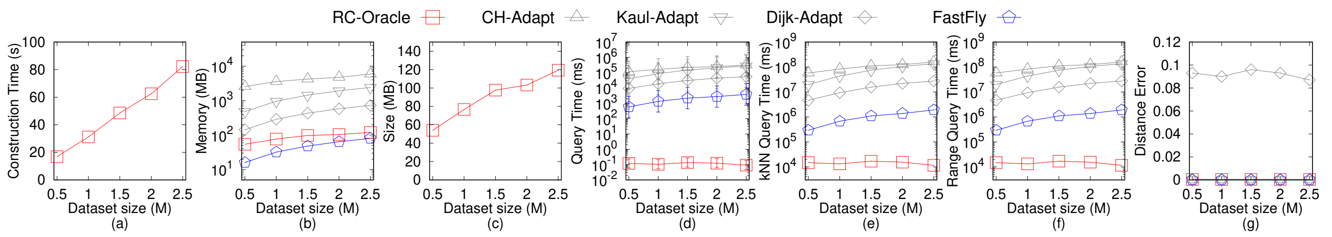


Figure 57: Baseline comparisons (effect of N on GF_p point cloud dataset for the P2P query)

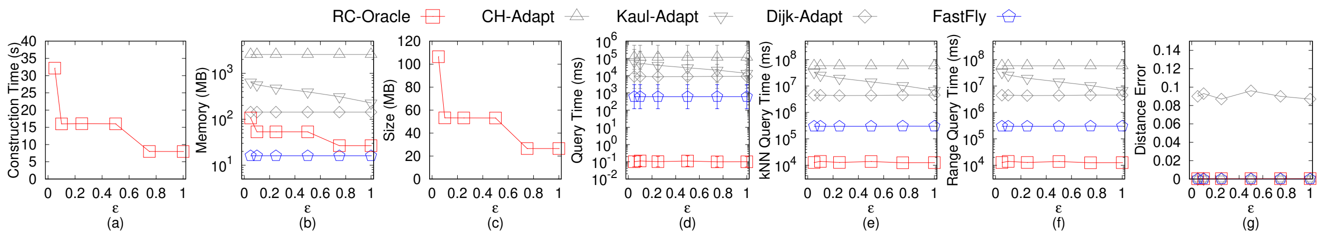


Figure 58: Baseline comparisons (effect of ϵ on LM_p point cloud dataset for the P2P query)

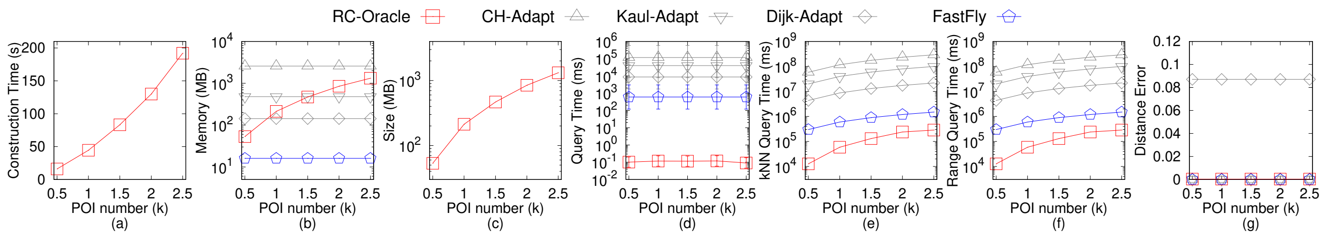


Figure 59: Baseline comparisons (effect of n on LM_p point cloud dataset for the P2P query)

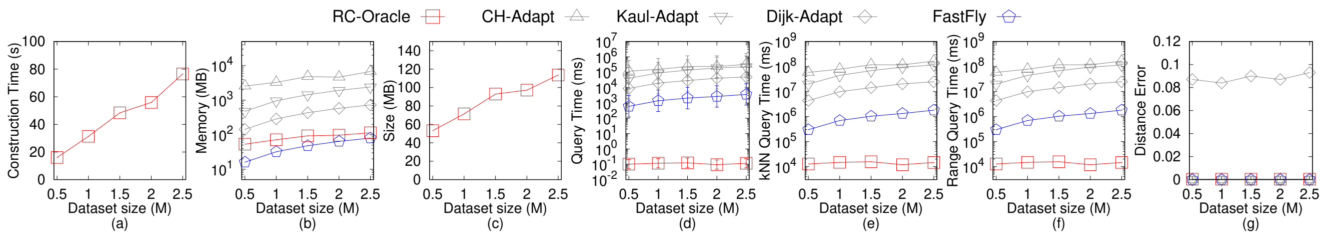


Figure 60: Baseline comparisons (effect of N on LM_p point cloud dataset for the P2P query)

are total N vertices on the *TIN*, the memory consumption is $O(N)$. Thus, the memory consumption of algorithm *Dijk-Adapt* is $O(N)$.

Thirdly, we prove the *error bound* of algorithm *Dijk-Adapt*. Recall that $\Pi_N(s,t|T)$ is the shortest network path passing on T (that is constructed by the point cloud) between s and t, so actually $\Pi_N(s,t|T)$ is the same as $\Pi_{\textit{Dijk-Adapt}}(s,t|T)$. We let $\Pi_E(s,t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s,t|T)$ passes) between s and t. Compared with $\Pi^*(s,t|T)$, we know $|\Pi_E(s,t|T)| \leq k \cdot |\Pi^*(s,t|T)|$ (according to left hand side equation in Lemma 2 of work [33]) and $|\Pi_N(s,t|T)| \leq |\Pi_E(s,t|T)|$ (since $\Pi_N(s,t|T)$ considers all the edges on T), so we have algorithm *Dijk-Adapt* always has $|\Pi_{\textit{Dijk-Adapt}}(s,t|T)| \leq k \cdot |\Pi^*(s,t|T)|$ for each pair of vertices s and t on T. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *Dijk-Adapt* returns the approximate shortest path passing on a point cloud. □

Theorem E.4. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt are* $O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, *and* $O(h^2)$. *Compared with* $\Pi^*(s,t|T)$, *SE-Oracle-Adapt always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{\textit{SE-Oracle-Adapt}}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{\textit{SE-Oracle-Adapt}}(s,t|T)$ *is the shortest surface path of SE-Oracle-Adapt*

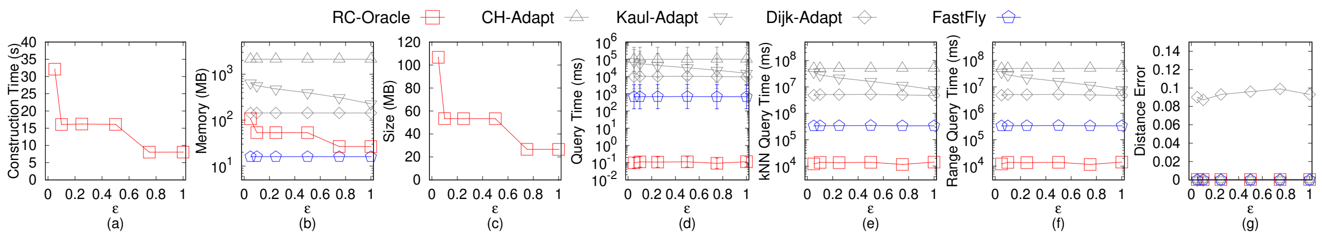


Figure 61: Baseline comparisons (effect of ϵ on RM_p point cloud dataset for the P2P query)

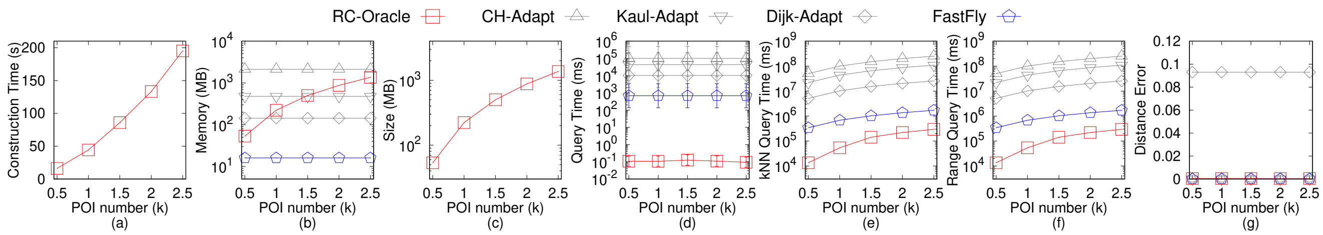


Figure 62: Baseline comparisons (effect of n on RM_p point cloud dataset for the P2P query)

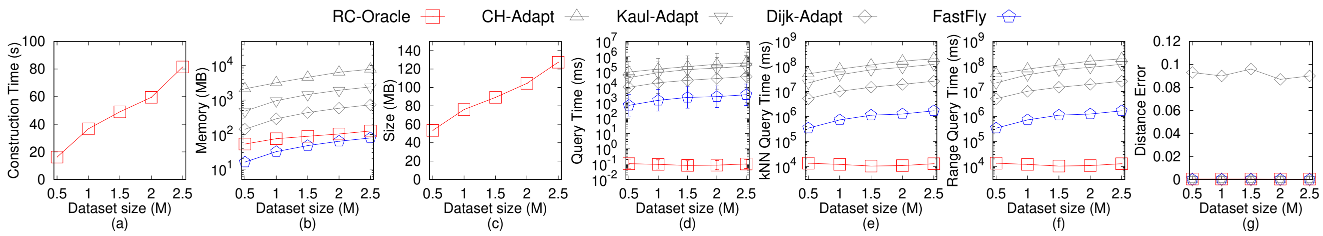


Figure 63: Baseline comparisons (effect of N on RM_p point cloud dataset for the P2P query)

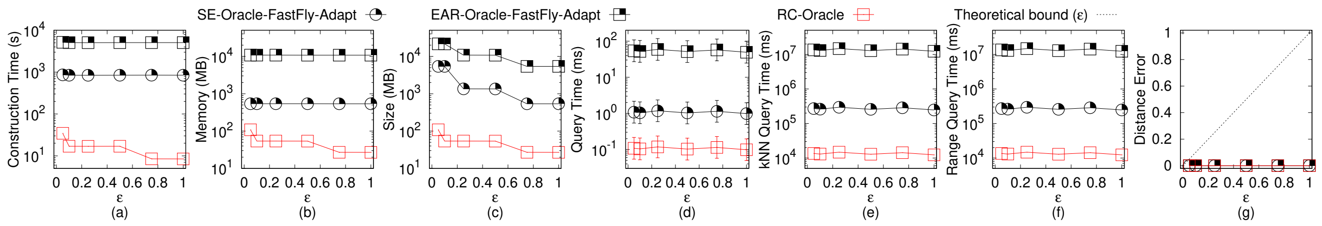


Figure 64: Ablation study on BH_p point cloud dataset for the P2P query

passing on a TIN T (that is constructed by the point cloud) between s and t. Compared with $\Pi^*(s,t|C)$, algorithm SE-Oracle-Adapt returns the approximate shortest path passing on a point cloud.

Proof. Firstly, we prove the *oracle construction time* of *SE-Oracle-Adapt*. The oracle construction time of the original oracle in work [58, 59] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-Adapt*, we use algorithm *CH* for the *TIN* shortest path query, which has shortest path query time $O(N^2)$ according to Theorem E.1. But, we also need to construct the *TIN* using the point cloud at the beginning, so we substitute u with N^2, and *SE-Oracle-Adapt* needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt* is $O(N + nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n) = O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$.

Secondly, we prove the *oracle size* of *SE-Oracle-Adapt*. The proof of the oracle size of *SE-Oracle-Adapt* is in work [58, 59]. Thus, the oracle size of *SE-Oracle-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the *shortest path query time* of *SE-Oracle-Adapt*. The proof of the shortest path query time of *SE-Oracle-Adapt* is in work [58, 59]. Thus, the shortest path query time of *SE-Oracle-Adapt* is $O(h^2)$.

Fourthly, we prove the *error bound* of *SE-Oracle-Adapt*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt* is algorithm *CH*, which returns the exact surface shortest path passing on T (that is constructed by the point cloud) according to

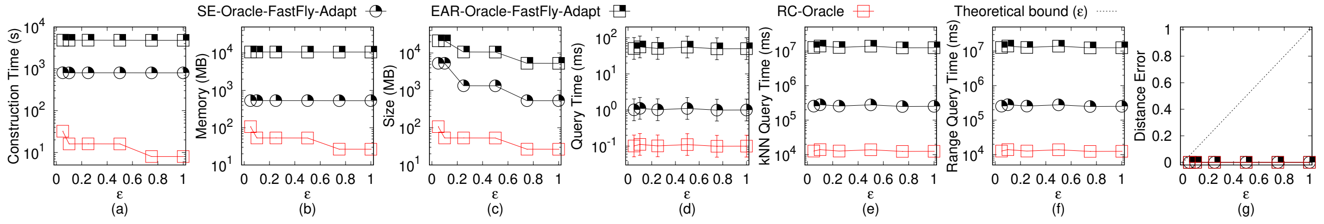


Figure 65: Ablation study on EP_p point cloud dataset for the P2P query

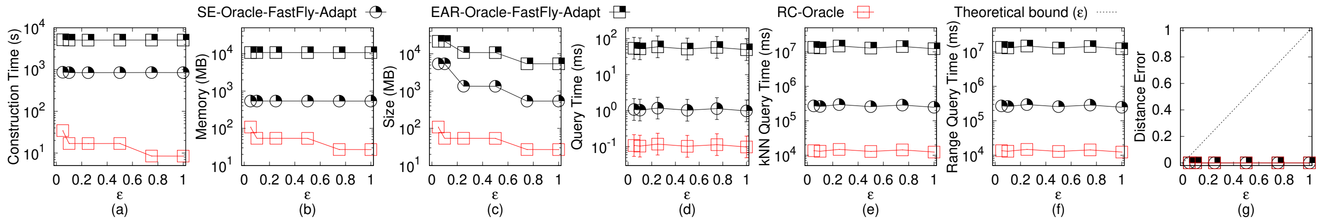


Figure 66: Ablation study on GF_p point cloud dataset for the P2P query

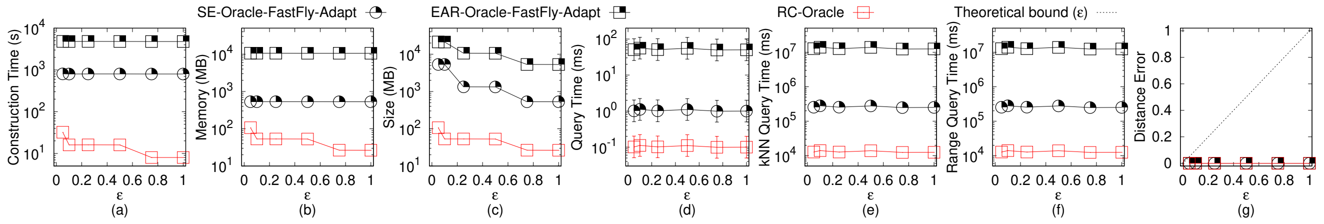


Figure 67: Ablation study on LM_p point cloud dataset for the P2P query

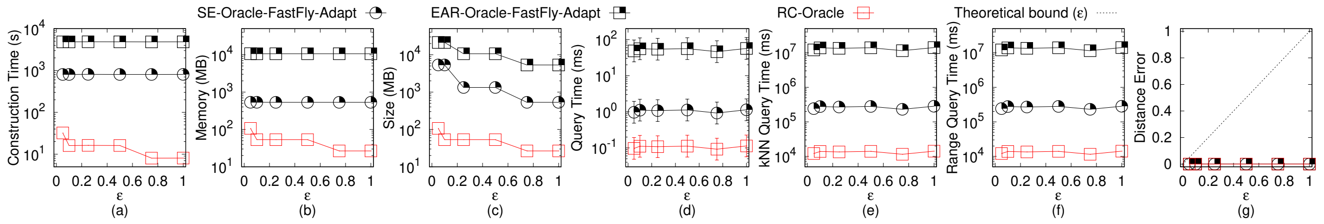


Figure 68: Ablation study on RM_p point cloud dataset for the P2P query

Theorem E.1, so the error of *SE-Oracle-Adapt* is due to the oracle itself. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt* is in work [58, 59]. Since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{\textit{SE-Oracle-Adapt}}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *SE-Oracle-Adapt* returns the approximate shortest path passing on a point cloud. □

Theorem E.5. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-FastFly-Adapt are $O(nN\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$. SE-Oracle-FastFly-Adapt always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{\textit{SE-Oracle-FastFly-Adapt}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P, where $\Pi_{\textit{SE-Oracle-FastFly-Adapt}}(s,t|C)$ is the shortest path of SE-Oracle-FastFly-Adapt passing on C between s and t.*

Proof. Firstly, we prove the *oracle construction time* of *SE-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle in work [58, 59] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. Thus, the oracle construction time of *SE-Oracle-FastFly-Adapt* is $O(nN\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$.

Secondly, we prove the *oracle size* of *SE-Oracle-FastFly-Adapt*. The proof of the oracle size of *SE-Oracle-FastFly-Adapt* is in work [58, 59]. Thus, the oracle size of *SE-Oracle-FastFly-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

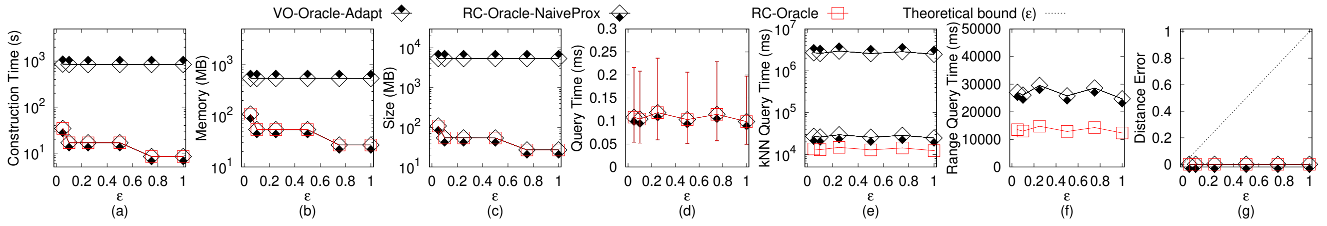


Figure 69: Comparisons with other proximity queries oracle and algorithm on BH_p point cloud dataset for the P2P query

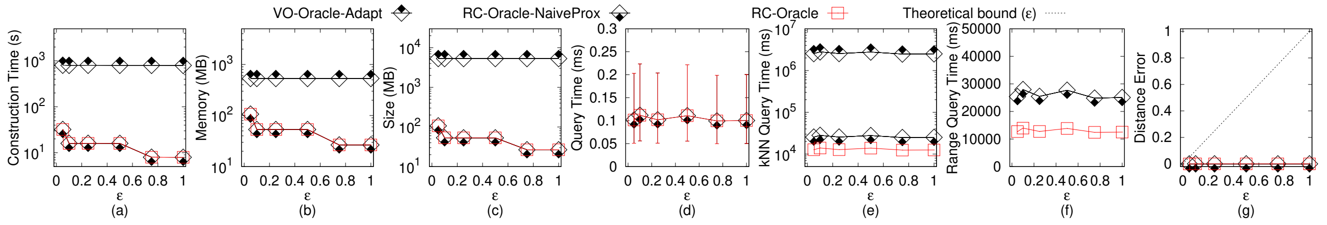


Figure 70: Comparisons with other proximity queries oracle and algorithm on EP_p point cloud dataset for the P2P query

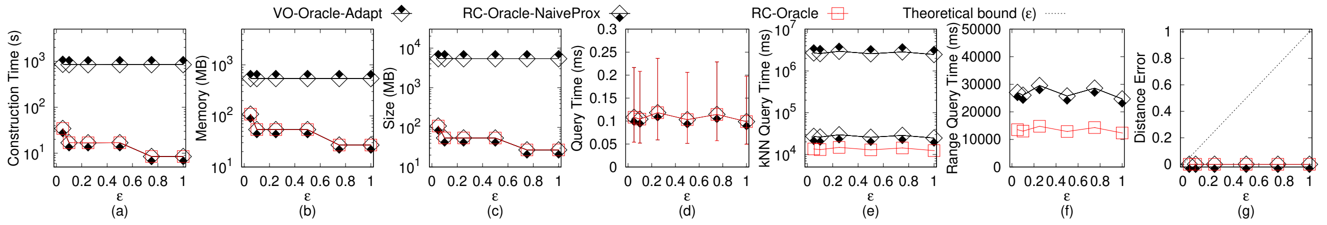


Figure 71: Comparisons with other proximity queries oracle and algorithm on GF_p point cloud dataset for the P2P query

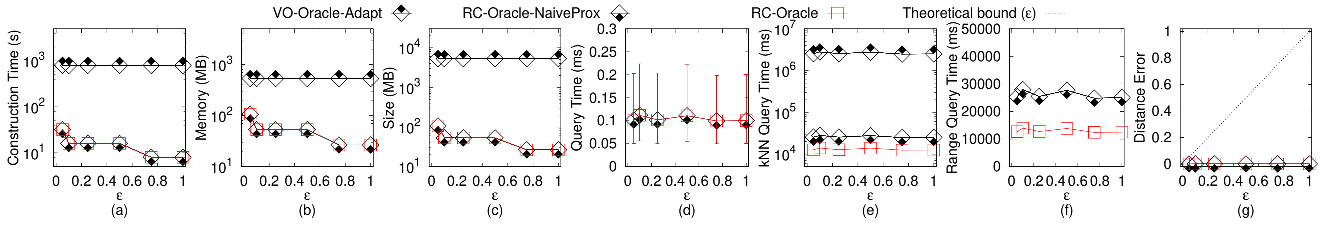


Figure 72: Comparisons with other proximity queries oracle and algorithm on LM_p point cloud dataset for the P2P query

Thirdly, we prove the *shortest path query time* of *SE-Oracle-FastFly-Adapt*. The proof of the shortest path query time of *SE-Oracle-FastFly-Adapt* is in work [58, 59]. Thus, the shortest path query time of *SE-Oracle-FastFly-Adapt* is $O(h^2)$.

Fourthly, we prove the *error bound* of *SE-Oracle-FastFly-Adapt*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest path passing on the point cloud according to Theorem 4.1, the error of *SE-Oracle-FastFly-Adapt* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-FastFly-Adapt* is in work [58, 59]. So we obtain that *SE-Oracle-FastFly-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P □

Theorem E.6. *The oracle construction time, oracle size, and shortest path query time of EAR-Oracle-Adapt are* $O(\lambda\xi mN^2 + \frac{nN^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, *and* $O(\lambda\xi\log(\lambda\xi))$. *Compared with* $\Pi^*(s,t|T)$, *EAR-Oracle-Adapt always has* $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)+2\delta|$ *for each pair of POIs s and t in P, where* $\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)$ *is the shortest surface path of EAR-Oracle-Adapt passing on a TIN T (that is constructed by the point cloud) between s and t, and δ is an error parameter [30]. Compared with* $\Pi^*(s,t|C)$, *algorithm EAR-Oracle-Adapt returns the approximate shortest path passing on a point cloud.*

Proof. Firstly, we prove the *oracle construction time* of *EAR-Oracle-Adapt*. The oracle construction time of the original oracle in work [30] is $O(\lambda\xi mu + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, where u is the on-the-fly shortest path query time. In *EAR-Oracle-Adapt*, we use algorithm *CH* for the *TIN* shortest path query, which has the shortest path query time $O(N^2)$ according to Theorem E.1.

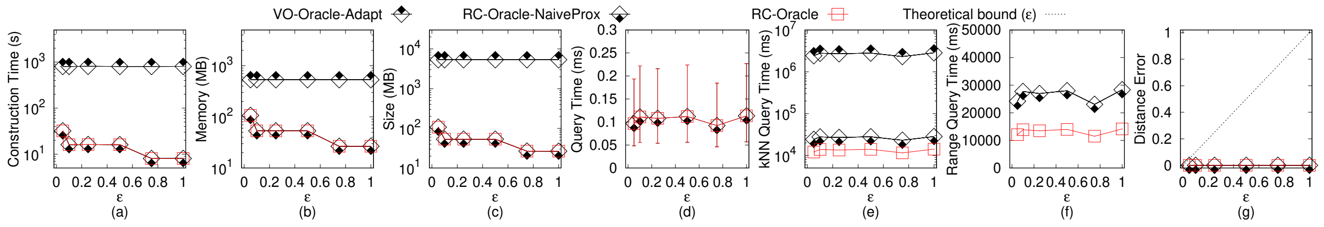


Figure 73: Comparisons with other proximity queries oracle and algorithm on RM_p point cloud dataset for the P2P query

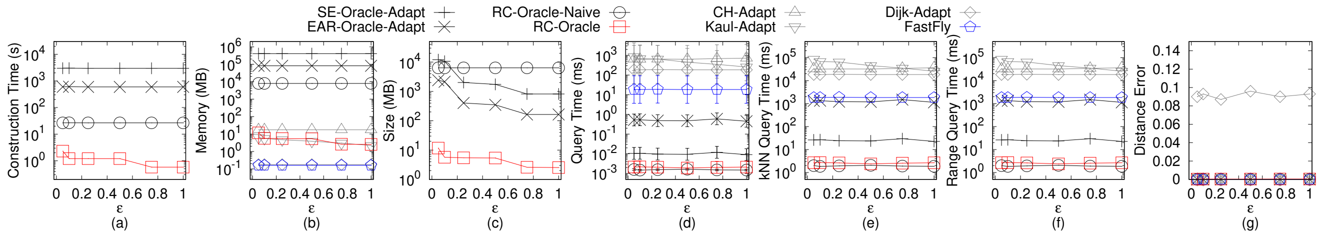


Figure 74: Baseline comparisons on EP_p point cloud dataset for the A2A query

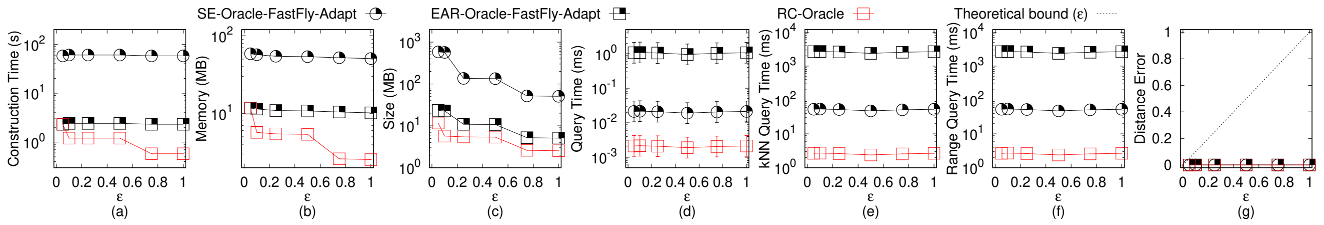


Figure 75: Ablation study on EP_p point cloud dataset for the A2A query

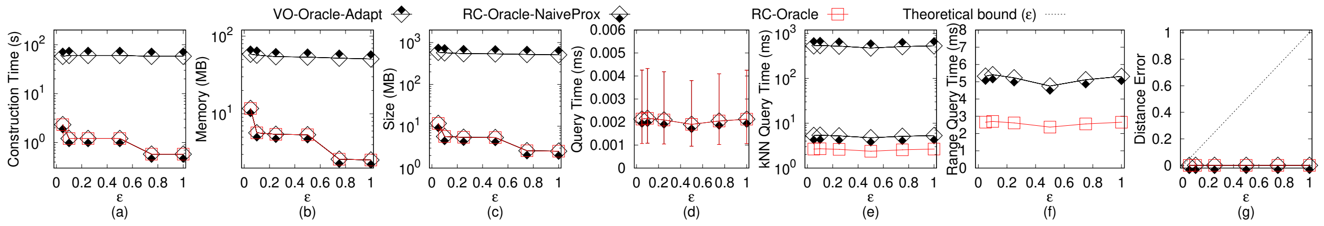


Figure 76: Comparisons with other proximity queries oracle and algorithm on EP_p point cloud dataset for the A2A query

But, we also need to construct the *TIN* using the point cloud at the beginning, so we substitute u with N^2, and *EAR-Oracle-Adapt* needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *EAR-Oracle-Adapt* is $O(N + \lambda\xi mu + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N) = O(\lambda\xi mu + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$.

Secondly, we prove the *oracle size* of *EAR-Oracle-Adapt*. The proof of the oracle size of *EAR-Oracle-Adapt* is in work [30]. Thus, the oracle size of *EAR-Oracle-Adapt* is $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the *shortest path query time* of *EAR-Oracle-Adapt*. The proof of the shortest path query time of *EAR-Oracle-Adapt* is in work [30]. Thus, the shortest path query time of *EAR-Oracle-Adapt* is $O(\lambda\xi\log(\lambda\xi))$.

Fourthly, we prove the *error bound* of *EAR-Oracle-Adapt*. Since the on-the-fly shortest path query algorithm in *EAR-Oracle-Adapt* is algorithm *CH*, which returns the exact surface shortest path passing on T (that is constructed by the point cloud) according to Theorem E.1, so the error of *EAR-Oracle-Adapt* is due to the oracle itself. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of the oracle itself regarding *EAR-Oracle-Adapt* is in work [30]. Since the *TIN* is constructed by the point cloud, we obtain that *EAR-Oracle-Adapt* always has $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T) + 2\delta|$ for each pair of POIs s and t in P. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *EAR-Oracle-Adapt* returns the approximate shortest path passing on a point cloud. □

Theorem E.7. *The oracle construction time, oracle size, and shortest path query time of EAR-Oracle-FastFly-Adapt are* $O(\lambda\xi mN\log N + \frac{N\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, *and* $O(\lambda\xi\log(\lambda\xi))$. *EAR-Oracle-FastFly-Adapt always has*

$|\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)+2\delta|$ *for each pair of POIs s and t in P, where* $\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)$ *is the shortest path of EAR-Oracle-FastFly-Adapt passing on C between s and t, and δ is an error parameter [30].*

Proof. Firstly, we prove the *oracle construction time* of *EAR-Oracle-FastFly-Adapt.* The oracle construction time of the original oracle in work [30] is $O(\lambda\xi mu + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, where u is the on-the-fly shortest path query time. In *EAR-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. Thus, the oracle construction time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda\xi mN\log N + \frac{N\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$.

Secondly, we prove the *oracle size* of *EAR-Oracle-FastFly-Adapt.* The proof of the oracle size of *EAR-Oracle-FastFly-Adapt* is in work [30]. Thus, the oracle size of *EAR-Oracle-FastFly-Adapt* is $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the *shortest path query time* of *EAR-Oracle-FastFly-Adapt.* The proof of the shortest path query time of *EAR-Oracle-FastFly-Adapt* is in work [30]. Thus, the shortest path query time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda\xi\log(\lambda\xi))$.

Fourthly, we prove the *error bound* of *EAR-Oracle-FastFly-Adapt.* Since the on-the-fly shortest path query algorithm in *EAR-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest path passing on the point cloud according to Theorem 4.1, the error of *EAR-Oracle-FastFly-Adapt* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *EAR-Oracle-FastFly-Adapt* is in work [30]. So we obtain that *EAR-Oracle-FastFly-Adapt* always has $|\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)+2\delta|$ for each pair of POIs s and t in P. □

Theorem E.8. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle-Naive are* $O(nN\log N + n^2)$, $O(n^2)$, *and* $O(1)$. *RC-Oracle-Naive returns the exact shortest path passing on the point cloud.*

Proof. Firstly, we prove the *oracle construction time* of *RC-Oracle-Naive.* Since there are total n POIs, *RC-Oracle-Naive* first needs $O(nm)$ time to calculate the shortest path passing on the point cloud from each POI to all other remaining POIs using on-the-fly shortest path query algorithm (which is a *SSAD* algorithm), where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store pairwise P2P shortest paths passing on the point cloud into a hash table. In *RC-Oracle-Naive*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute m with $N\log N$. Thus, the oracle construction time of *RC-Oracle-Naive* is $O(nN\log N + n^2)$.

Secondly, we prove the *oracle size* of *RC-Oracle-Naive. RC-Oracle-Naive* stores $O(n^2)$ pairwise P2P shortest paths passing on the point cloud. Thus, the oracle size of *RC-Oracle-Naive* is $O(n^2)$.

Thirdly, we prove the *shortest path query time* of *RC-Oracle-Naive. RC-Oracle-Naive* has a hash table to store the pairwise P2P shortest paths passing on the point cloud. Thus, the shortest path query time of *RC-Oracle-Naive* is $O(1)$.

Fourthly, we prove the *error bound* of *RC-Oracle-Naive.* Since the on-the-fly shortest path query algorithm in *RC-Oracle-Naive* is algorithm *FastFly*, which returns the exact shortest path passing on the point cloud according to Theorem 4.1, and the oracle itself regarding *RC-Oracle-Naive* also computes the pairwise P2P exact shortest paths passing on the point cloud, so *RC-Oracle-Naive* returns the exact shortest path passing on the point cloud. □

Theorem E.9. *The oracle construction time, oracle size, and kNN query time of VO-Oracle-Adapt are* $O(N^2\log N)$, $O(N)$, *and* $O(N\log^2 N)$. *EVO-Oracle-Adapt returns the exact kNN result.*

Proof. The proof of the oracle construction time, oracle size, *kNN* query time, and error analysis of *VO-Oracle-Adapt* is in work [54]. □