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 the point cloud in recent years, and researchers started to utilize its advantages over the terrain surface (represented by *T*riangular *I*rregular *N*etwork, i.e., *TIN*) in the proximity queries, including the *shortest path query*, the k-*N*earest *N*eighbor (*kNN*) *query*, and the *range query*. As can be observed from the existing studies, the on-the-fly and oracle-based shortest path algorithms on a *TIN* are very expensive. All existing on-the-fly shortest path algorithms on a point cloud are still not efficient, and there are no oracle-based shortest path algorithms on a point cloud. Motivated by this, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the proximity queries 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 path between any pair of POIs stored in the oracle. Then, we propose algorithms for answering the *kNN* query and the range query with the assistance of our path oracle. Our experimental results show that our oracle is up to 390 times, 2 times, and 6 times better than the best-known oracle-based algorithm 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 6 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, 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Conducting proximity queries, including (1) the *shortest path query*, (2) the k-*N*earest *N*eighbor (*kNN*) *query* [8], and (3) the *range query* [11], on a 3D surface has become a topic of widespread interest in both industry and academia [29, 64]. The shortest path query is the most fundamental type of proximity query. Numerous well-known companies and applications, such as Google Earth [3] and the renowned 3D computer game Cyberpunk 2077 [5], utilize the shortest path 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 [23, 35, 36, 45, 61, 62, 65, 66]. There are different representations of a 3D surface, including terrain surfaces represented by a *T*riangular *I*rregular *N*etwork (*TIN*) and point clouds. 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 on point clouds of the driving environment for autonomous driving [16, 22, 44, 48], and Metaverse uses the shortest path on point clouds of objects such as mountains and hills to help users reach the destination faster in Virtual Reality [42, 43]. Applications of the other two proximity queries include rover path planning [18] and military tactical analysis [39].

Point cloud and *TIN*: A point cloud is represented by a set of 3D *points* in space. Figure 1 (a) shows a satellite map of Mount Rainier [52] (a renowned national park in the USA) in an area of 20km × 20km, and Figure 1 (b) shows the point cloud with 81 points of the Mount Rainier. 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 (c) is an example of a *TIN*, which consists of vertices, edges, and faces. We focus on three types of paths, i.e., paths passing on (1) a point cloud (Figure 1 (b)), (2) the faces on a *TIN* (Figure 1 (c), or (3) the edges on a *TIN* Figure 1 (d).

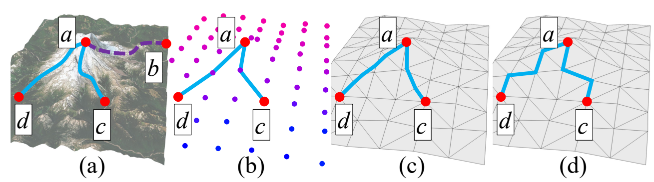


Figure 1: Paths passing on (a) a satellite map, (b) a point cloud, (c) the faces of a *TIN*, and (d) the edges of a *TIN*

1.1 Motivation

1.1.1 ***Advantages of point cloud***. Answering proximity queries on a point cloud has four advantages compared with on a *TIN*.

(1) *More direct access to point cloud data.* For example, we can use an iPhone 12/13 Pro LiDAR scanner [59] to scan an object and generate a point cloud in 10s, 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 [49]. But, in order to obtain a *TIN*, typically, researchers need to transform a point cloud to a *TIN* [34]. Our experimental result shows that it needs 210s ≈ 3.5 min to transform a point cloud with 25M points to a *TIN*.

(2) *Lower memory consumption of a point cloud.* We only store the point information of a point cloud, but we need to store the vertex,

SIGMOD '24, June 11–16, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

edge, and face information of a *TIN*, and our experimental result shows that storing a point cloud with 25M points needs 390MB, but storing a *TIN* generated by this point cloud needs 1.7GB.

(3) *Faster proximity query time on a point cloud.* Compared with calculating the paths passing on a point cloud, calculating the paths passing on both of the faces or edges of a *TIN* is slow, since a *TIN* is more complicated than a point cloud, it takes more time to pre-process a *TIN*. In addition, calculating the paths passing on the faces of a *TIN* is even slower since the search space is larger. Our experimental result shows that calculating one shortest path passing on a point cloud with 2.5M points needs 3s, but calculating one path passing on the faces of a *TIN* generated by this point cloud needs 580s $\approx$ 10 min, and calculating one path passing on the edges of the *TIN* needs 17s.

(4) *Small distance error of the shortest path passing on a point cloud.* In Figure 1 (b) and (c), the shortest path passing on a point cloud is similar to the shortest path passing on the faces of a *TIN* (because in Figure 2 (a), each point q is connected with 8 neighbor points, i.e., 7 blue points and 1 red point s). But, in Figure 1 (c) and (d), the path passing on the faces and edges of a *TIN* are very different (because in Figure 2 (b), each vertex q is connected with only 6 blue neighbour vertices). Our experimental result shows that the length of the shortest path passing on a point cloud is only 1.04 times larger than that of the shortest path passing on the faces of a *TIN*, but the length of the shortest path passing on the edges of a *TIN* is 1.3 times larger than that of the shortest path passing on the faces of a *TIN*.

1.1.2 ***Usages of POIs.*** Given a set of *points-of-interest (POIs)* on a point cloud, conducting proximity queries between *pairs of POIs* on the point cloud, i.e., *points of interest-to-points of interest (P2P) proximity query*, is important. For example, POIs can be reference points used in measuring similarities between two different 3D objects [38, 57], and POIs can be residential locations used in conducting proximity queries of the wildness animals when studying their migration patterns [26, 46].

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 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 result is called the *shortest path query time*). Applications of using an oracle include network routing and social network analysis [62].

1.1.4 ***Real-life example.*** We conducted a case study on an evacuation simulation in Mount Rainier due to snowfall [53]. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [13], and one may be dead due to asphyxiation [40] if s/he gets buried in the snow. In Figure 1 (a), we would like to find the shortest paths (in blue and purple lines) from one of the viewing platforms (POIs) on the mountain to its k-nearest hotels (POIs) due to the limited capacity of each hotel (where a is the viewing platform, and b to d are the hotels). In Figure 1 (b) - (d), c and d are the k-nearest hotels to this viewing platform where $k = 2$. We can also find the shortest path from one of the viewing platforms to all the hotels that are not further than r km using the range query. Our experimental result shows that we can construct an oracle on a point cloud with 5M points and 500 POIs (250 viewing platforms and 250 hotels) in 400s $\approx$ 6.6 min, but it needs 77,200s $\approx$ 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 12.5s with the oracle, but it needs 4,400s $\approx$ 1.2 hours on a point cloud without the oracle. These show the usefulness to perform proximity queries on point cloud with POIs using oracles in real-life applications.

1.2 Challenges

1.2.1 ***Inefficiency for on-the-fly algorithm.*** All existing algorithms [50, 58, 67] for conducting the 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 on this *TIN* on-the-fly (which is time-consuming). The best-known on-the-fly *exact* [19] and *approximate* [35] algorithm that calculates a path passing on the faces of a *TIN* 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. The best-known on-the-fly *approximate* algorithm that calculates a path passing on the edges of a *TIN* [36] runs in $O(N\log N)$ time. Our experimental result shows (1) algorithm [19] needs 290,000s $\approx$ 3.4 days, (2) algorithm [35] needs 90,000s $\approx$ 1 day, and (3) algorithm [36] needs 15,000s $\approx$ 4.2 hours to perform the *kNN* query for all 2500 POIs on a *TIN* with 0.5M vertices, which is very slow.

1.2.2 ***Non-existence of oracle.*** There is no existing work that answers the proximity queries on a point cloud using an oracle. The best-known existing work [61, 62] only build an oracle on a *TIN*. Although we can first construct a *TIN* using the point cloud, then use oracle [61, 62] for pairwise P2P *point cloud* shortest path oracle construction, its oracle construction time is still very large due to two reasons. (1) *Bad criterion for algorithm earlier termination*: Although it uses *<u>S</u>ingle-<u>S</u>ource <u>A</u>ll-<u>D</u>estination* (*SSAD*) algorithm [19, 35, 36], i.e., a Dijkstra-based algorithm [27], to pre-compute the shortest path from each POI to other POIs, and provide a criterion to *terminate it earlier*, its criterion is very loose, and different POIs have the *same* earlier termination criterion. In our experiment, even after *SSAD* algorithm has visited most of the POIs, this earlier termination criterion is still not reached. (2) *Additional heavy data structure construction*: It always constructs the oracle using of two additional time-consuming constructed data structures, called *compressed partition tree* [61, 62] and *well-separated node pair set* [17]. The oracle construction time and oracle size of the oracle [61, 62] (after adaption on the point cloud) are $O(nN^2 + cn)$ and $O(cn)$, respectively, where n is the number of POIs on the point cloud and c is a constant depending on the point cloud (where $c \in [35, 80]$ on a point cloud with 2.5M points on average). In our experiment, its oracle construction time and oracle size are 78,000s $\approx$ 21.7 hours and 1.5GB for a point cloud with 2.5M points and 500 POIs.

1.3 Our Oracle and Proximity Query Algorithms

Motivated by these, we propose an efficient $(1+\epsilon)$-approximate path oracle that answers the proximity queries for a set of POIs on a point cloud called *Rapid Construction path Oracle on point cloud*, 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 adapted point cloud oracle [61, 62] due to the concise information about the pairwise shortest path between any pair of POIs stored in the oracle, where ϵ is a non-negative real user parameter called an *error parameter*. Based on *RC-Oracle*, we develop efficient proximity query algorithms for the *kNN* and range queries. 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 on point cloud*, i.e., *FastFly*, which is a Dijkstra-based algorithm [27] returning its calculated shortest path passing on the *points* of the point cloud. This can significantly reduce the algorithm's running time, since computing the shortest path on a *TIN* is expensive.

(2) **Rapid oracle construction**: When constructing the oracle part of *RC-Oracle*, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, to calculate the shortest path from for each POI to other POIs *simultaneously*, and set *different* earlier termination criterion for each different POI, so that this criterion is tight. Furthermore, we directly construct *RC-Oracle* on the point cloud, without any other additional data structures.

1.4 Contribution and Organization

We summarize our major contributions as follows.

(1) We propose *RC-Oracle*, which is the first oracle that efficiently answers the P2P shortest path query on a point cloud. We also propose algorithm *FastFly* used for constructing *RC-Oracle*. We also develop efficient proximity query algorithms with the assistance of *RC-Oracle*.

(2) We provide thorough theoretical analysis on the oracle construction time, oracle size, shortest path query time, and error bound of *RC-Oracle*, and on the *kNN* query time, range query time, and error bound for proximity queries. We also provide theoretical analysis on the relationships between the shortest path passing on a point cloud, and passing on the faces or edges of a *TIN*.

(3) *RC-Oracle* performs much better than the best-known oracle [61, 62] in terms of the oracle construction time, oracle size, and shortest path query time, and *RC-Oracle* support real-time response. The *kNN* query time and range query time with the assistance of *RC-Oracle* also performs much better than the best-known oracle. Our experimental results show that the *RC-Oracle*'s oracle construction time is 200s ≈ 3.2 min and output size is 50MB, but the best-known oracle [61, 62] needs more than 78,000s ≈ 21.7 hours and 1.5GB for a point cloud with 2.5M points and 500 POIs. Under the same setting, the *kNN* query time and range query time of all 500 POIs for *RC-Oracle* are both 25s, but the best-known exact on-the-fly algorithm [19] needs 290,000s ≈ 3.4 days, the best-known approximate on-the-fly algorithm [35] needs 161,000s ≈ 1.9 days, and the best-known oracle [61, 62] needs 150s. Our case study also shows *RC-Oracle* 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 and POI.*** 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. $L_y = y_{max} - y_{min}$) 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 [15, 28], because a grid-based 3D object, e.g., a grid-based point cloud [15, 28] and a grid-based *TIN* [24, 45, 56, 61, 62], 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 eight blue points. We can easily extend our problem to the non-grid-based point cloud. The only difference is that we need to re-define $N(p)$. Given a point p in a non-grid-based point cloud, we 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, e.g., r. 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, $n \leq N$. With the new definition of neighbor points, we can calculate the exact shortest path in the non-grid-based point cloud (using the same exact shortest path definition in the grid-based point cloud).

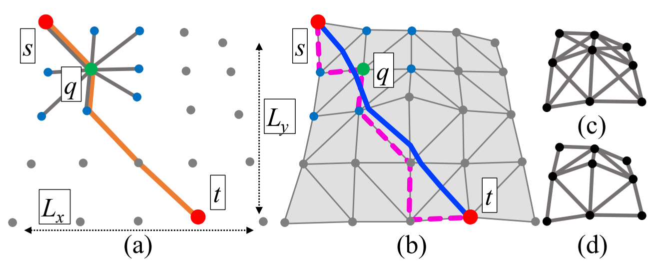


Figure 2: (a) Orange $\Pi^*(s,t|C)$, (b) blue $\Pi^*(s,t|T)$ and pink $\Pi_E(s,t|T)$, (c) a conceptual graph for a point cloud, and (d) a conceptual graph for a *TIN*

2.1.2 ***Path.*** Given a pair of neighbour points p and p' in C, we define $d_E(p,p')$ to be the Euclidean distance between point p and p'. Given a pair of points s and t in P, we define $\Pi^*(s,t|C) = (s = q_1, q_2, \ldots, q_l = t)$, with $l \geq 2$, to be the exact shortest path between s and t passing on the *points* of a point cloud C, such that the total Euclidean distance $\sum_{i=1}^{l-1} d_E(q_i, q_{i+1})$ is the minimum, where for $i \in \{1, \ldots, l\}$, q_i is a point in C and $q_{i+1} \in N(q_i)$. We further define

$|\cdot|$ to be the length of a path on C (e.g., $|\Pi^*(s,t|C)|$ is the length of the exact shortest path $\Pi^*(s,t|C)$ on C). The orange line in Figure 2 (a) shows an exact shortest path $\Pi^*(s,t|C)$ on a point cloud C. Let $\Pi(s,t|C)$ be the shortest path of returned by *RC-Oracle*. *RC-Oracle* guarantees that $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any s and t in P. Performing the shortest path query on C can be regarded as performing the shortest path query on a conceptual graph G. Let $G.V$ and $G.E$ be the set of vertices and edges of G, where each point in C is denoted by a vertex in $G.V$, and $G.E$ consists of eight edges connecting each vertex $v \in G.V$ to its eight closest neighboring vertices. Figure 2 (c) is a conceptual graph used for calculating the shortest path on a point cloud.

Let T be a *TIN* triangulated [51] by the points in C. Given a pair of points s and t in P, let $\Pi^*(s,t|T)$ be the exact shortest path between s and t passing on the faces of T, let $\Pi_E(s,t|T)$ be the shortest path between s and t passing on the edges of T. 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 $\Pi_E(s,t|T)$ in pink line. Similar to G, performing the shortest path query on the edges of T can be regarded as performing the shortest path query on a conceptual graph G'. 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) is a conceptual graph used for calculating the shortest path on the edges of a *TIN*.

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

There are two types of proximity queries on a point cloud, including (1) *P2P proximity query*, and (2) *any points-to-any points (A2A) proximity query*, i.e., given a point cloud, conducting proximity queries between *pairs of any points* on the point cloud. By creating POIs which has the same coordinate values as all points in the point cloud, the A2A proximity query can be regarded as one form of the P2P proximity query. In the main body of this paper, we focus on the P2P proximity query. We study the A2A proximity query in the appendix. Furthermore, in the P2P proximity 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 proximity 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 after removing the POI. A notation table can be found in the appendix of Table 3.

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 *kNN* query and the range query.

3 RELATED WORK

3.1 On-the-fly Algorithm

Most (if not all) existing algorithms [50, 58, 67] for conducting the proximity queries on a point cloud *on-the-fly* are very slow, since they calculate the shortest path on an implicit structure (e.g., a *TIN*). Given a point cloud, they first triangulate it into a *TIN* [51] in $O(N)$ time, then they calculate the shortest path on this *TIN* on-the-fly. There are two types of algorithms for computing the shortest path on a *TIN*, which are (1) *exact* algorithm [19, 47, 63] and (2) *approximate* algorithm [35, 36, 41].

Exact algorithm: The algorithm [47] (resp. algorithm [63]) uses continuous Dijkstra (resp. checking window) algorithm to calculate the exact shortest path on a *TIN* on-the-fly in $O(N^2 \log N)$ (resp. $O(N^2 \log N)$) time, and the best-known exact algorithm [19] (as recognized by work [35, 36, 56, 64]) 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, the best-known exact algorithm [19] (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. We denote algorithm *Chen and Han-Adaption*, i.e., *CH-Adapt*, to be the adapted algorithm in work [50, 58, 67], which first constructs a *TIN* using the given point cloud, and then uses algorithm [19] for computing the exact shortest path on the faces of the *TIN*. It is a *SSAD* algorithm.

Approximate algorithm: All algorithms [35, 36, 41] 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 $(1+\epsilon)$-approximate shortest path on the *TIN* on-the-fly. The best-known approximate algorithm [35] (as recognized by work [61, 62]) that calculates the path on the face of a *TIN* 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*. The best-known approximate algorithm [36] that calculates the path on the edges of a *TIN* runs in $O(N \log N)$ time. If we let the path pass on the point of the point cloud, both algorithms [35, 36] (without constructing a *TIN* first) can be adapted on the point cloud, and this adaption is similar as algorithm *FastFly*. We denote algorithm *Kaul-Adaption*, i.e., *Kaul-Adapt* (resp. *Dijkstra-Adaption*, i.e., *Dijk-Adapt*), to be the adapted algorithm in work [50, 58, 67], which first constructs a *TIN* using the given point cloud, and then uses algorithm [35] (resp. algorithm [36]) for computing the approximate shortest path passing on the faces (resp. edges) of the *TIN*. They are *SSAD* algorithms.

Drawbacks of the on-the-fly algorithms: All these algorithms are very slow even on a moderate-size point cloud. Our experimental result show algorithm (1) *CH-Adapt* needs 290,000s ≈ 3.2 days, (2) *Kaul-Adapt* needs 90,000s ≈ 1 day, and (3) *Dijk-Adapt* needs 15,000s ≈ 4.2 hours to perform the *kNN* query for all 2500 POIs on a point cloud with 0.5M points.

3.2 Oracle

There is no existing work for answering the proximity queries between pairs of POIs (i.e., calculating the pairwise P2P shortest path) on a point cloud in the form of an oracle. *Sterint Point Oracle*

(*SP-Oracle*) [14] and Space Efficient Oracle (*SE-Oracle*) [61, 62] only pre-compute the approximate pairwise P2P shortest path in the form of oracle on a *TIN*. We can first construct a *TIN* using the point cloud, then use them for pairwise P2P *point cloud* shortest path oracle construction. We denote *SP-Oracle-Adapt* to be the adapted oracle of *SP-Oracle* [14] that uses first construct a *TIN* from a point cloud, then uses *SP-Oracle* on this *TIN*. Similarly, we denote *SE-Oracle-Adapt* as the adapted oracle of *SE-Oracle* [61, 62].

SP-Oracle-Adapt uses a Steiner graph to index the $(1+\epsilon)$-approximation pairwise P2P shortest path. The oracle construction time, oracle size, and shortest path query time of *SP-Oracle-Adapt* is $O(\frac{N}{\epsilon^2 \sin\theta}\log^3\frac{N}{\epsilon}\log^2\frac{1}{\epsilon})$, $O(\frac{N}{\epsilon^{1.5}\sin\theta}\log^2\frac{N}{\epsilon}\log^2\frac{1}{\epsilon})$, and $O(\frac{1}{\epsilon\sin\theta}\log\frac{1}{\epsilon}+\log\log(N+n))$, respectively. *SE-Oracle-Adapt* first constructs a *compressed partition tree* [61, 62], then partitions the POIs into several levels of *well-separated node pair sets* [17] using the compressed partition tree, and finally uses the node pair set to index the $(1+\epsilon)$-approximation pairwise P2P shortest path. The oracle construction time, oracle size, and shortest path query time of *SE-Oracle-Adapt* is $O(nN^2+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [30, 37] ($\beta\in[1.5,2]$ in practice according to work [61, 62]).

Drawback of *SP-Oracle-Adapt*: The oracle construction time for *SP-Oracle-Adapt* is very large since there are *many Steiner points in the Steiner graph construction.* The experimental result in work [61, 62] shows the oracle construction time of *SP-Oracle-Adapt* is up to 25,000 times larger than that of *SE-Oracle-Adapt*. Thus, we do not focus on this oracle in this paper.

Drawbacks of *SE-Oracle-Adapt*: The oracle construction time for *SE-Oracle-Adapt* is still large due to two reasons (as mentioned in Section 1.2.2). (1) *Bad criterion for algorithm earlier termination*: Its earlier termination criterion for each *SSAD* algorithm is not well-designed, because 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 *SSAD* algorithm. (2) *Additional heavy data structure construction*: It always needs to construct the oracle using the compressed partition tree and the well-separated node pair set. *RC-Oracle* does not need to pre-compute any other additional data structures. We denote *SE-Oracle-Adapt2* to be a further adapted oracle of *SE-Oracle* that uses algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without constructing a *TIN*. Our experimental results show that for a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt2* is 2,600s ≈ 45 min, while *RC-Oracle* just needs 200s ≈ 3.2 min.

3.3 Other related work

The work [24, 25] use a multi-resolution terrain model to answer the *kNN* queries on a *TIN* in $O(N^2)$ time and the work [56] uses a Voronoi diagram to answer the *kNN* queries on a *TIN* in $O(N\log^2 N)$ time, which are very costly. The experimental result in work [61, 62] shows the *kNN* query time of work [24, 25, 56] is up to 10 times larger than that of using *SE-Oracle-Adapt*, so they are not our main focus. The work [33] proposes an *arbitrary points-to-arbitrary points* oracle called *EAR-Oracle* on the faces of a *TIN*, which uses the same idea as in *SE-Oracle-Adapt*, i.e., well-separated node pair sets. But, an arbitrary point on the faces of a *TIN* has no physical meaning on a point cloud, so it is not our main focus. We still compare *EAR-Oracle* when we study A2A proximity queries on *TIN*s in the appendix. Due to the same drawback as in *SE-Oracle-Adapt*, *EAR-Oracle* is 10^4 times slower than *RC-Oracle* in terms of oracle construction time.

4 METHODOLOGY

4.1 Overview

4.1.1 ***Components of RC-Oracle.*** There are two components, which are the *path map table* and the *POI map table*.

(1) **The path map table** M_{path} is a *hash table* [21] that stores the selected pairs of POIs u and v in P, i.e., a key $\langle u,v\rangle$, and their corresponding exact shortest path $\Pi^*(u,v|C)$, i.e., a value, on C. 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 $\Pi^*(u,v|C)$ in $O(1)$ time. In Figure 3 (e), M_{path} stores exact shortest paths on C, corresponding to the 7 paths in Figure 3 (d). For the exact shortest paths between b and c, M_{path} stores $\langle b,c\rangle$ as key and $\Pi^*(b,c|C)$ as value.

(2) **The POI map table** M_{POI} is a *hash table* stores the POI u, i.e., a key, that we do not store all the exact shortest paths in M_{path} from u to other non-processed POIs, and the POI v, i.e., a value, that we use the exact shortest path with v as a source to approximate the shortest path with u as a source. The space consumption and query time of M_{POI} is similar to M_{path} (i.e., linear space consumption and constant query time). In Figure 3 (e), we store b as key, and a as value, since we use the exact shortest path with a as a source to approximate the shortest path with b as a source.

4.1.2 ***Phases of RC-Oracle.*** 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 between some selected pairs of POIs on C, 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*

4.2.1 ***Small oracle construction time.*** We discuss the reason why *RC-Oracle* has a small oracle construction time.

(1) **Rapid point cloud on-the-fly shortest path query algorithm**: When constructing *RC-Oracle*, we do not use any existing on-the-fly path query algorithm [50, 58, 67], that is, we do not (1) construct a *TIN*, and (2) calculate the point cloud shortest path on this *TIN* on-the-fly [19, 35, 36, 41, 47, 63]. Instead, we use algorithm *FastFly*, such that the calculated shortest path passes on the *point* of the point cloud.

(2) **Rapid oracle construction**: When constructing the oracle part of *RC-Oracle*, we do not use the best-known oracle [61, 62] due to the large oracle construction time. Intuitively, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, with each POI as a source for n times, then use *different* earlier termination criteria for each POI to terminate *SSAD* algorithm earlier for time-saving, to construct the oracle.

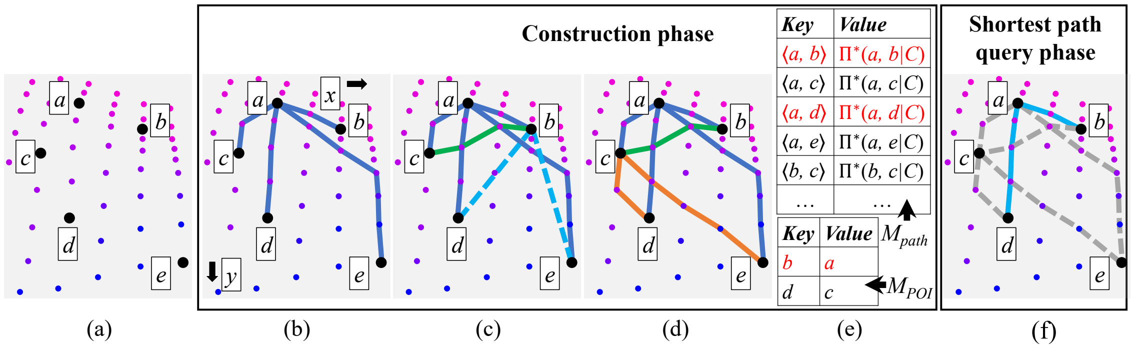


Figure 3: Framework overview

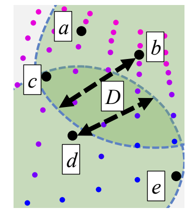


Figure 4: SE-Oracle

There are two versions of a *SSAD* algorithm. (a) Given a source POI and a set of destination POIs, *SSAD* algorithm can terminate earlier if it has visited all destination POIs. (b) Given a source POI and a *termination distance* (denoted by D), *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 path, 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 to illustrate the construction process of *RC-Oracle*. In Figure 3 (b), for a, we use algorithm *FastFly* to calculate the shortest path 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 algorithm *FastFly* (a *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 *SSAD* algorithm earlier with d as a source. We found that there is even no need to use *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 the POIs that we should visit for time-saving.

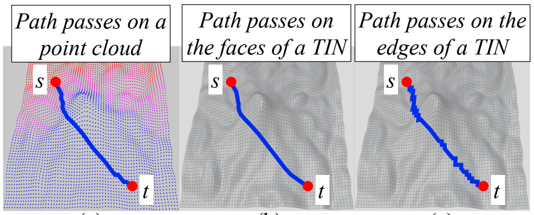
However, in *SE-Oracle-Adapt*, it has the *bad criterion for algorithm earlier termination* drawback. After the construction of the compressed partition tree, it pre-computes the shortest paths using algorithm *CH-Adapt* (a *SSAD* algorithm) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of *SSAD* algorithm and set 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 POI 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 terminates *SSAD* algorithm earlier until e is visited. The two versions of *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*. Furthermore, we directly construct *RC-Oracle* on the point cloud, without any other additional data structures (corresponding to the *additional heavy data structure construction* drawback of *SE-Oracle-Adapt*).

4.2.2 ***Small oracle size.*** We introduce the reason why *RC-Oracle* has a small oracle size. We only store a small number of paths in *RC-Oracle*, and 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. In Figure 3 (f), in the shortest path query phase of *RC-Oracle*, we need to query the shortest path (1) between a and d, (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 $\Pi^*(b,d|C)$.

4.3 Implementation Details of algorithm *FastFly* and *RC-Oracle*

4.3.1 ***Algorithm FastFly.*** In algorithm *FastFly*, given a point cloud C and a pair of points s and t in C, it can calculate the *exact* shortest path between s and t passing on the *points* of C, i.e., $\Pi^*(s,t|C)$, using Dijkstra's algorithm [27] on a conceptual graph (see Figure 2 (c)) of a C. Figure 5 shows the shortest path passing on (1) a point cloud, (2) the faces of a *TIN*, and (3) the edges of a *TIN* of Mount Rainier in an area of 20km × 20km. The shortest path passing on the point cloud and the faces of the *TIN* are similar, but calculating the shortest path passing on the point cloud is much faster than that on the faces of the *TIN*, since the query region of the former is smaller than the latter. But, the shortest path passing



(a) (b) (c)

Figure 5: Shortest path on point cloud and *TIN*

on the edges of the *TIN* has a larger error than the shortest path passing on the point cloud.

4.3.2 ***Construction Phase of RC-Oracle.*** In the construction phase, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths between some selected pairs of POIs on C, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{POI}.

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

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

(1) *POIs sorting* (see line 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, e, d.

(2) *Shortest path calculation* (see line 4-20).

(2.1) *Exact shortest path calculation* (see line 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 on C from a to b, e, c, d using algorithm *FastFly*, these paths are in purple lines, and store each of them with key-value pair in M_{path}.

(2.2) *Shortest path approximation* (see line 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* (see line 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* (see line 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 the approximate shortest path $\Pi(b, d|C)$ by appending $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi(b, e|C)$ by appending $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{POI}.
  - *Close POIs* (see line 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 path result to approximate $\Pi^*(b, c|C)$, then we store c into $P_{dest}(b)$.
  - *Selected exact shortest path calculation* (see line 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 on C between b and c, i.e., $\Pi^*(b, c|C)$ in green line, and store it with key-value pair in M_{path}. Note that we can terminate algorithm *FastFly* earlier since we just need to visit all POIs that are close to b, and we do not need to visit d and e.
- *Leaving approximation looping* (see 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.

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 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 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}

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

4.3.3 ***Shortest Path Query Phase of RC-Oracle.*** In the 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 $\Pi(s, t|C)$, which is a $(1 + \epsilon)$-approximated exact shortest path of $\Pi^*(s, t|C)$ on 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 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 s or t is a key in M_{POI}, or (2) s is processed before t during construction phase if both s and t are keys in M_{POI}. For both of two cases, we retrieve s' from M_{POI} using s 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 Proximity Query Algorithms

In order to show how our proposed algorithm could be used in other proximity queries, we describe some simple applications as follows. Given a point cloud C, a set of points P on C, a query point $q \in P$, and a range value r, we can answer other proximity queries, i.e., the *kNN* query and the range query, using *RC-Oracle*.

4.4.1 ***Definitions***. We give the definition of two proximity queries using the exact shortest distance on C.

- **The *kNN* query**: It returns all the shortest paths on C from q to a set of k POIs, denoted by $X_1 = \{u_1, u_2, \dots u_k\}$, which are k POIs in P nearest to q, in other words, $\max_{\forall u \in X_1} |\Pi^*(q, u|C)| \leq \min_{\forall o \in P \setminus X_1} |\Pi^*(q, o|C)|$.
- **The range query**: It returns all the shortest paths on C from q to a set of POIs, denoted by $X_2 = \{u_1, u_2, \dots\}$, which are a set of POIs in P with shortest distance to q at most r, in other words, $\max_{\forall u \in X_2} |\Pi^*(q, u|C)| \leq r$.

4.4.2 ***Algorithms***. Given a query point $q \in P$, we first perform n shortest path queries between q and all other POIs in P with the assistance of the shortest path query phase in *RC-Oracle*. Recall that $\Pi(s, t|C)$ is the shortest path of returned by *RC-Oracle* between s and t. Then, we process them as follows.

- **The *kNN* query**: We return the shortest paths on C from q to a set of POIs X'_1 containing k POIs in P, such that $\max_{\forall u' \in X'_1} |\Pi(q, u'|C)| \leq \min_{\forall o' \in P \setminus X'_1} |\Pi^*(q, o'|C)|$.
- **The range query**: We return the shortest paths on C from q to a set of POIs X'_2 containing k POIs in P, such that $\max_{\forall u' \in X'_2} |\Pi(q, u'|C)| \leq r$.

It is worth mentioning that the above description is just a simple application of our proposed algorithms. How to design an index particularly for the *kNN* query and the range query is left as the future work.

4.5 Theoretical Analysis

4.5.1 ***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 usage of algorithm FastFly are $O(N \log N)$ and $O(N)$, respectively. Algorithm FastFly returns the exact shortest path 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 usage. Since Dijkstra algorithm is guaranteed to return the exact shortest path, so algorithm *FastFly* returns the exact shortest path on the point cloud. □

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

Proof Sketch. The *oracle construction time* contains (1) the POIs sorting time $O(n \log n)$ due to the n POIs, (2) the shortest path calculation time $O(N \log N + n)$ due to (2a) the usage of algorithm *FastFly* for $O(1)$ POIs according to standard packing property [32] and each algorithm *FastFly* needs $O(N \log N)$ time, and (2b) the usage of Euclidean distance calculation for other $O(n)$ POIs and each calculation needs $O(1)$ time. The *oracle size* contains M_{POI} and M_{path} both with size $O(n)$. The *shortest path query time* is due to the hash table $O(1)$ query time of M_{POI} and M_{path}.

For 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 approximate shortest path $\Pi(s, t|C)$ is calculated by appending $\Pi^*(s, u|C)$ 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 Euclidean distance between two points is no larger than the distance of the shortest path on the point cloud between the same two points. The detailed proof appears in the appendix. □

4.5.2 ***Path passing on the point cloud and the faces or edges of a TIN***. Given a source s and a destination t, recall that $\Pi^*(s, t|C)$ is the shortest path passing on a point cloud, $\Pi^*(s, t|T)$ is the shortest path passing on the faces of a *TIN*, and $\Pi_E(s, t|T)$ is the shortest path passing on the edges of a *TIN*. The relationship between $|\Pi^*(s, t|C)|$ and $|\Pi^*(s, t|T)|$ is shown in Lemma 4.3.

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

Proof Sketch. We let $\Pi'_E(s, t|T)$ be the shortest path between s and t passing on the edges of T where these edges belong to the faces that $\Pi^*(s, t|T)$ passes. According to left hand side equation in Lemma 2 of work [36], we have $|\Pi'_E(s, t|T)| \leq k \cdot |\Pi^*(s, t|T)|$. Since $\Pi_E(s, t|T)$ considers all the edges on T, so $|\Pi_E(s, t|T)| \leq |\Pi'_E(s, t|T)|$. In Figure 2 (a), given a green point q on C, it can connect

Table 1: Comparison of algorithms

	Algorithm	Oracle construction time	Oracle size	Shortest path query time	Error
Oracle	*SE-Oracle-Adapt* [61, 62]	Large	Medium	Small	Small
	SE-Oracle-Adapt2 [61, 62]	Medium	Medium	Small	Small
	RC-Oracle-Naive	Medium	Large	Tiny	Small
	***RC-Oracle* (ours)**	**Small**	**Small**	**Tiny**	**Small**
On-the-fly	*CH-Adapt* [19]	N/A	N/A	Large	No error
	Kaul-Adapt [35]	N/A	N/A	Large	Small
	Dijk-Adapt [36]	N/A	N/A	Medium	Medium
	***FastFly* (ours)**	N/A	N/A	**Medium**	**Small**

Table 2: Point cloud datasets

	Name	$\lvert N \rvert$
Original dataset	*BearHead* (BH_p) [2, 61, 62]	0.5M
	EaglePeak (EP_p) [2, 61, 62]	0.5M
	Gunnison Forest (GF_p) [7, 60]	0.5M
	Laramie Mountain (LM_p) [9, 60]	0.5M
	Robinson Mountain (RM_p) [4, 60]	0.5M
Small-version dataset	BH_p small-version (BH_p*-small*)	10k
	EP_p small-version (EP_p*-small*)	10k
	GF_p small-version (GF_p*-small*)	10k
	LM_p small-version (LM_p*-small*)	10k
	RM_p small-version (RM_p*-small*)	10k
Multi-resolution dataset	Multi-resolution of BH_p	1M, 1.5M, 2M, 2.5M
	Multi-resolution of EP_p	1M, 1.5M, 2M, 2.5M
	Multi-resolution of GF_p	1M, 1.5M, 2M, 2.5M
	Multi-resolution of LM_p	1M, 1.5M, 2M, 2.5M
	Multi-resolution of RM_p	1M, 1.5M, 2M, 2.5M
	Multi-resolution of EP_p*-small*	20k, 30k, 40k, 50k

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 $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$. Thus, we finish the proof by combining these inequalities. The detailed proof appears in the appendix. □

The relationship between $|\Pi^*(s,t|C)|$ and $|\Pi_E(s,t|T)|$ is shown in Lemma 4.4.

Lemma 4.4. *Given a pair of points s and t in P, we have $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$.*

Proof Sketch. The proof is similar to that in Lemma 4.3. □

4.5.3 ***Proximity query algorithms.*** We provide theoretical analysis on the proximity query algorithms using *RC-Oracle*. For the *kNN* query and the range query, since both of them return a set of POIs, for simplicity, given a query point $q \in P$, (1) we let X be a set of POIs containing the *exact* (1a) k nearest POIs of q or (1b) POIs whose distance to q are at most r, calculated using the exact distance on C. Furthermore, given a query point $q \in P$, (2) we let X' be a set of POIs containing (2a) k nearest POIs of q or (2b) POIs whose distance to q are at most r, calculated using the approximated distance on C returned by *RC-Oracle*. We let v_f (resp. v'_f) be the furthest POI to q in X (resp. X') based on the exact distance on C, i.e., $|\Pi^*(q,v_f|C)| \leq \max_{\forall v \in X} |\Pi^*(q,v|C)|$ (resp. $|\Pi^*(q,v'_f|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q,v'|C)|$).

In Figure 1 (a), suppose that the exact k nearest POIs ($k=2$) of a is c, d, i.e., $X=\{c,d\}$. And b is the furthest POI to a among these three POIs, i.e., the value for v is f. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k=2$) of a is b, c, i.e., $X'=\{b,c\}$. Besides, d is the furthest POI to a among these three POIs, i.e., the value for v' is d.

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 4.5.

Theorem 4.5. *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 n time usages of the shortest path query phase of *RC-Oracle*. 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++. For the following experimental setup, we mainly follow the experiment setup in the work [35, 36, 45, 61, 62]. We conducted experiments with a point cloud and a *TIN* as input, separately.

Datasets: (1) We first describe the datasets for experiments on a point cloud. We conducted our experiment based on 34 real point cloud datasets in Table 2, where the subscript p means 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 [3] with 8km × 6km covered region, and then used Blender [1] to generate the point cloud. These three datasets have a resolution of 10m × 10m [24, 45, 56, 61, 62]. We extracted 500 POIs using OpenStreetMap [61, 62] for BH_p, EP_p, GF_p, LM_p and RM_p datasets. For BH_p*-small*, EP_p*-small*, GF_p*-small*, LM_p*-small* and RM_p*-small* datasets, we use the same region of the BH_p, EP_p, GF_p, LM_p and RM_p datasets with a (lower) resolution of 70m × 70m to generate them (i.e., their small-version datasets) using the dataset generation procedure in [45, 61, 62]. This procedure can be found in the appendix. In addition, we have six sets of datasets with different numbers of points (five sets of large-version datasets (where each set contains datasets with 1M, 1.5M, 2M, 2.5M points) and one set of small-version datasets with 20k, 30k, 40k, 50k points) for testing the scalability of our oracle, which are generated using BH_p, EP_p, GF_p, LM_p, RM_p and EP_p*-small* datasets with the same procedure. (2) We then describe the datasets for experiments on a *TIN*. Based on the 34 point clouds datasets, we triangulate [51] 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) We first describe the algorithms for our proximity queries on a point cloud (i.e., the problem we are studying in this paper). In the following, 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* [34] so that the existing algorithm could be used. Their algorithm names are appended by "*-Adapt*". We have four on-the-fly algorithms, i.e., (a) *CH-Adapt*: the adapted best-known exact algorithm that calculates paths passing on the faces of a *TIN* (constructed by the given point cloud) [19], (b) *Kaul-Adapt*: the adapted best-known approximate algorithm that calculates paths passing on the faces of a *TIN* (constructed by the given point cloud) [35], (c) *Dijk-Adapt*: the adapted best-known approximate algorithm that calculates paths passing on the edges of a *TIN* (constructed by the given point cloud) [36], (d) *FastFly*: our algorithm that calculates paths passing on a point cloud. We have four oracles, i.e., (e) *SE-Oracle-Adapt*: the adapted best-known oracle [61, 62] that calculates paths passing on the faces of a *TIN* (constructed by the given point cloud), (f) *SE-Oracle-Adapt2*: the adapted best-known oracle [61, 62] that uses *FastFly* to directly calculate the shortest path passing on a point cloud without constructing a *TIN*, (g) *RC-Oracle-Naive*: the naive version of our oracle *RC-Oracle* without shortest path approximation step, and (h) *RC-Oracle*: our oracle. We compare the oracle construction time, oracle size, and shortest path query time of baselines in Table 1. The theoretical analysis with proofs of these baselines can be found in the appendix.

(2) We then describe the algorithms for our proximity queries on a *TIN* (i.e., the existing problem studied by previous studies). Similarly, we have four on-the-fly algorithms, i.e., (a) *CH*: the best-known exact algorithm that calculates paths passing on the faces of a *TIN* [19], (b) *Kaul*: the best-known approximate algorithm that calculates paths passing on the faces of a *TIN* [35], (c) *Dijk*: the best-known approximate algorithm that calculates paths passing on the edges of a *TIN* [36], (d) *FastFly-Adapt*: our adapted algorithm (for the queries on a *TIN*) that calculates paths passing on a conceptual graph, where the vertices of this 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 have three oracles, i.e., (e) *SE-Oracle*: the best-known oracle [61, 62], (f) *RC-Oracle-Naive-Adapt*: the adapted naive version of our oracle without shortest path approximation step that calculates paths passing on a conceptual graph, and (g) *RC-Oracle-Adapt*: our oracle that calculates paths passing on a conceptual graph.

Query Generation: We conducted all proximity queries, i.e., (1) shortest path query, (2) all POIs *kNN* query, and (3) all POIs range query. (1) For the shortest path query, we issued 100 query instances where for each instance, we randomly chose two POIs in P, 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 POIs *kNN* query (resp. range query), we first calculate the shortest path from each POI to all other POIs, then select the k nearest POIs (resp. the POIs that are no further than a given distance value r) of the current POI as output, and we repeat this for all POIs. Since we first need to find the shortest path from a given POI to all other POIs (because there is no index designed for the *kNN* query and the range query in this study due to our focus on studying the shortest path queries), the value of k and r will not affect their query time, so we set k to be 3 and r to be 1 km.

Factors and Measurements: We studied three factors in the experiments, namely (1) N, (2) n, and (3) ϵ. In addition, we used seven measurements to evaluate the algorithm performance, namely (1) *oracle construction time*, (2) *memory usage* (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 POIs kNN query time), (6) *range query time* (i.e., all POIs 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 4.5.3), and (9) *range query error* (i.e., the error rate of the range query defined in Section 4.5.3).

5.2 Experimental Results for *TIN*s

We first study the proximity queries on *TIN*s (studied by existing studies) to justify why our proximity queries on *point clouds* are useful in practice. The path calculated by *CH* is regarded as the exact shortest path because the path we consider here passes on the faces of the *TIN* (and thus, used for distance error calculation). Our experimental result shows that *SE-Oracle* and *RC-Oracle-Naive-Adapt* are not feasible on large-version datasets due to their expensive oracle construction time (more than 30 days), so we (1a) 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 (1b) compared *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets with default 500 POIs. Figure 6 and 7 show the P2P proximity query result on a *TIN* when varying N and n on EP_t*-small* and GF_t*-small* datasets. The results on other combinations of dataset, the variation of N, n, and ϵ, and the A2A proximity query can be found in the appendix.

There are two types of proximity queries on a *TIN*, including (1) *P2P proximity query*, and (2) *arbitrary points-to-arbitrary points (A2A) proximity query*, i.e., given a *TIN*, conducting proximity queries between *pairs of arbitrary points* on the *TIN*. The P2P proximity query on a *TIN* is similar on a point cloud, but the A2A proximity query on a *TIN* is different on a point cloud. For the A2A proximity query on a *TIN*, a point may lie on the face of a *TIN*. We focus on the P2P proximity query on a *TIN*, and study the A2A proximity query on a *TIN* in the appendix.

5.2.1 *Effect of N (scalability test) for P2P proximity query on a TIN*.

In Figure 6, 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. *RC-Oracle-Adapt* performs better than all the remaining algorithms in terms of oracle construction time, oracle size, and shortest path query time. 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. 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 0.04). This motivates us to conduct experiments on point clouds in Section 5.3.

5.2.2 *Effect of n for P2P proximity query on a TIN*.

In Figure 7, we tested 5 values of n from {50, 100, 150, 200, 250} on GF_t*-small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 7 (a) and (d), 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.

5.2.3 *Effect of ϵ for P2P proximity query on a TIN*.

We tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on different datasets by setting N to be 10k and n to be 50. The result can be found in the

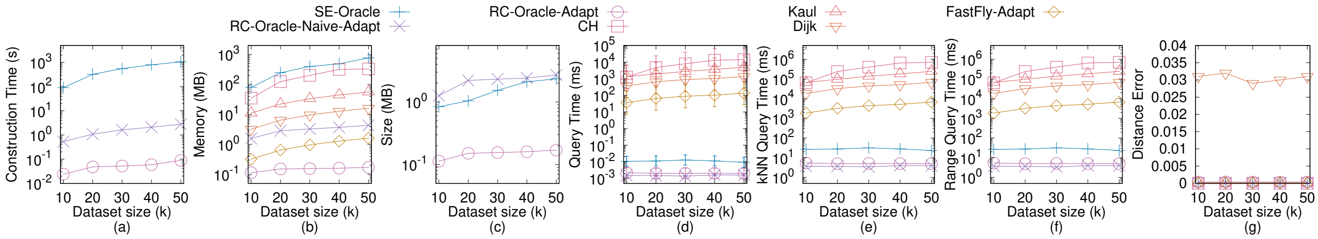


Figure 6: Effect of N on EP_t-*small TIN* dataset

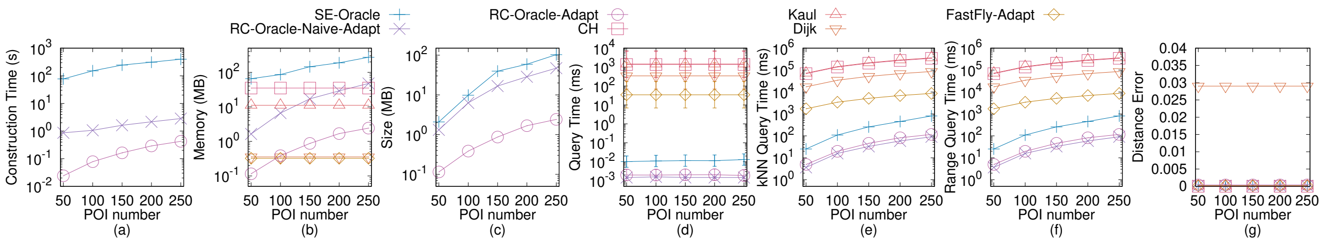


Figure 7: Effect of n on GF_t-*small TIN* dataset

appendix. The oracle construction time, oracle size, and shortest path query time of *RC-Oracle-Adapt* still perform better than *SE-Oracle*, and other algorithms / oracles.

5.2.4 ***Ablation test.*** In Figure 6 and 7, the shortest path query time of *FastFly-Adapt* is the smallest compared with other on-the-fly algorithms, i.e., *CH*, *Kaul*, and *Dijk* (although we adapt *FastFly* to *FastFly-Adapt*, which makes *FastFly-Adapt* even slower). The distance error of *FastFly-Adapt* is still much smaller than that of *Dijk*.

5.3 Experimental Results for Point Clouds

Based on the experimental results in the previous section, we understand the effectiveness of the proximity queries on *point clouds*. In this section, we then study the proximity queries on *point clouds* using the algorithms in Table 1. The path calculated by *FastFly* is regarded as the exact shortest path because the path we consider here passes on points of the point cloud (and thus, used for distance error calculation). Our experimental result shows that *SE-Oracle-Adapt*, *SE-Oracle-Adapt2* and *RC-Oracle-Naive* are not feasible on large-version datasets due to their expensive oracle construction time (more than 30 days), so we (1a) compared *SE-Oracle-Adapt*, *SE-Oracle-Adapt2*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets with default 50 POIs, and (1b) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets with default 500 POIs. Figure 8 to 10 show the P2P proximity query result on a point cloud when varying ϵ, N, and n on RM_p-*small*, RM_p, BH_p, and GF_p datasets. The results on other combinations of dataset, the variation of ϵ, N and n, and the A2A proximity query can be found in the appendix.

5.3.1 ***Effect of N (scalability test) for P2P proximity query on a point cloud.*** In Figure 9, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on RM_p dataset by setting n to be 500 and ϵ to be 0.1 for scalability test. *RC-Oracle* performs better than all the remaining algorithms in terms of oracle construction time, oracle size, and shortest path query time.

5.3.2 ***Effect of n for P2P proximity query on a point cloud.*** In Figure 10, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_p dataset by setting N to be 0.5M and ϵ to be 0.1. In Figure 10 (a), the oracle construction time for *RC-Oracle* is very small. In Figure 10 (c), the shortest path query time for on-the-fly algorithms are large.

5.3.3 ***Effect of ϵ for P2P proximity 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 RM_p-*small* dataset by setting N to be 10k and n to be 50. The oracle construction time, oracle size, and all POIs *kNN* query time of *RC-Oracle* still perform better than the best-known oracle *SE-Oracle-Adapt*, and other adapted algorithms / oracles. The distance error of *RC-Oracle* is also very small (close to 0), but the distance error of *Dijk* is large.

5.3.4 ***Ablation test.*** *SE-Oracle-Adapt2* and *RC-Oracle* only differ by oracle construction. Figure 8 to 10 show that the oracle construction time, oracle size, and shortest path query time of *RC-Oracle* is smaller than that of *SE-Oracle-Adapt2*, which shows the usefulness of the oracle part of *RC-Oracle*.

5.3.5 ***A2A proximity query on a point cloud.*** We tested the A2A proximity query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on EP_p dataset. We selected 50 points as reference points for the *kNN* and range query. The result can be found in the appendix. *RC-Oracle* still performs much better than other adapted algorithms / oracles.

5.3.6 ***Case study.*** We conducted a case study on an evacuation simulation in Mount Rainier [52] due to the frequent heavy snowfall [53], where Mount Rainier has the highest seasonal total snowfall world record [12]. In the case of snowfall, Mount Rainier National Park will be closed and staffs will evacuate tourists in the mountain to the closest hotels immediately for tourists' safety. The time of

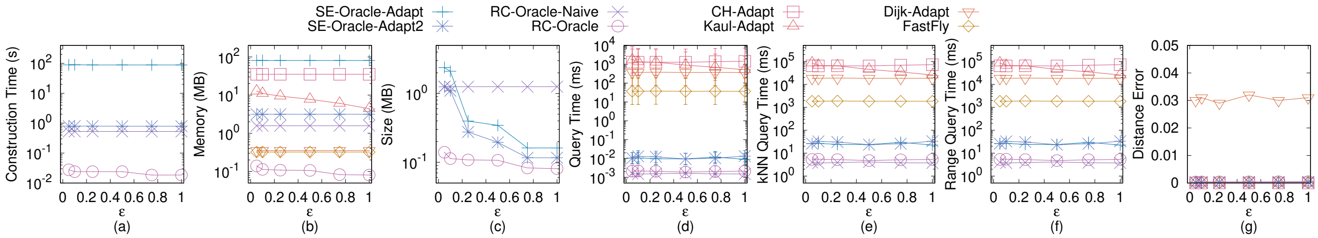


Figure 8: Effect of ϵ on LM_p-*small* point cloud dataset

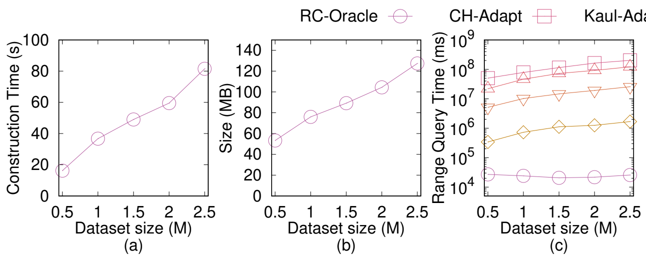


Figure 9: Effect of N on RM_p point cloud dataset

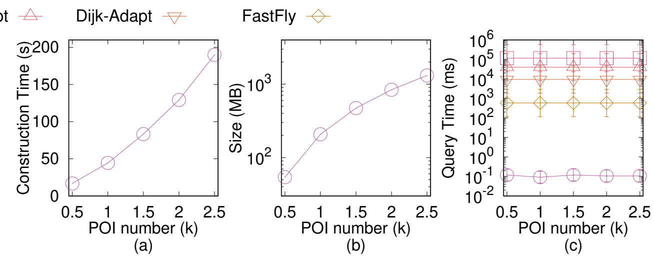


Figure 10: Effect of n on BH_p point cloud dataset

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 [10], 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 shortest paths is expected to be finished within 12 min ($= 2.4 - 2.2$ hours). Figure 1 (a) shows the satellite map of Mount Rainier. We conducted the *kNN* query to find the shortest paths (in blue and purple lines) from different viewing platforms on the mountain to k-nearest hotels (a is one of the viewing platforms, and b to d are the hotels). c and d are the k-nearest hotels to this viewing platform ($k = 2$).

Our experimental result shows that for a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels), the oracle construction time for (1) *RC-Oracle* is 200s ≈ 3.2 min, (2) the best-known oracle *SE-Oracle-Adapt* is 78,000s ≈ 21.7 hours, and (3) the adapted oracle *SE-Oracle-Adapt2* is 2,600s ≈ 45 min. Under the same setting, the query time for calculating 10 nearest hotels of each viewing platform for (1) *RC-Oracle* is 12.5s, (2) the best-known oracle *SE-Oracle-Adapt* is 75s, (3) the oracle *SE-Oracle-Adapt2* is 75s, (4) the best-known on-the-fly exact algorithm *CH-Adapt* is 145,000s ≈ 1.7 days, and (5) the best-known on-the-fly approximate algorithm *Kaul-Adapt* is 80,500s ≈ 22.5 hours. Thus, *RC-Oracle* is the best one in the evacuation since 3.2 min + 12.5s ≤ 12 min, which shows the usefulness of performing proximity queries on the point cloud with POIs by using *RC-Oracle* in real-life application. *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.

[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 [20, 55]) in Mount Rainier is 1 meter per 24 hours [54], 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 [31].

5.3.7 ***Summary.*** *RC-Oracle* consistently outperforms the best-known oracle, i.e., *SE-Oracle-Adapt*, and all other adapted baselines in terms of oracle construction time, oracle size, and shortest path query time. Specifically, *RC-Oracle* is up to 390 times, 2 times, and 6 times better than *SE-Oracle-Adapt*, in terms of the oracle construction time, oracle size and shortest path query time. With the assistance of *RC-Oracle*, our algorithms for the *kNN* query and the range query are both up to 6 times faster than *SE-Oracle-Adapt*. For a point cloud with 2.5M points and 500 POIs, the oracle construction time for (1) *RC-Oracle* is 200s ≈ 3.2 min, and (2) the best-known oracle *SE-Oracle-Adapt* is 78,000s ≈ 21.7 hours, and (3) the oracle *SE-Oracle-Adapt2* is 2,600s ≈ 45 min. Under the same setting, the *kNN* query time and the range query time for (1) *RC-Oracle* are both 25s, (2) the best-known oracle *SE-Oracle-Adapt* are both 150s, (3) the oracle *SE-Oracle-Adapt2* are both 150s, (4) the best-known on-the-fly exact algorithm *CH-Adapt* are both 290,000s ≈ 3.4 days, and (5) the best-known on-the-fly approximate algorithm *Kaul-Adapt* are both 161,000s ≈ 1.9 days.

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* query and the range query. The future work can be proposing a new pruning step to further reduce the oracle construction time and oracle size. Besides, we could explore how to build an index designed for the *kNN* query and the range query for better performance.

REFERENCES

[1] 2022. *Blender.* https://www.blender.org
[2] 2022. *Data Geocomm.* http://data.geocomm.com/
[3] 2022. *Google Earth.* https://earth.google.com/web

[4] 2022. *Robinson Mountain.* https://en.wikipedia.org/wiki/Robinson_Mountain
[5] 2023. *Cyberpunk 2077.* https://www.cyberpunk.net
[6] 2023. *Google Map.* https://www.google.com/maps
[7] 2023. *Gunnison National Forest.* https://en.wikipedia.org/wiki/Gunnison_National_Forest
[8] 2023. *k-nearest neighbors algorithm.* https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[9] 2023. *Laramie Mountain.* https://en.wikipedia.org/wiki/Laramie_Mountains
[10] 2023. *Preferred walking speed.* https://en.wikipedia.org/wiki/Preferred_walking_speed
[11] 2023. *Range query.* https://en.wikipedia.org/wiki/Range_query_(database)
[12] 2023. *Snow.* https://en.wikipedia.org/wiki/Snow
[13] 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
[14] Lyudmil Aleksandrov, Anil Maheshwari, and J-R Sack. 2005. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM (JACM)* 52, 1 (2005), 25–53.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG*. New York, NY, USA, 360–369.
[20] 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
[21] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[28] 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.
[29] David Eriksson and Evan Shellshear. 2016. Fast exact shortest distance queries for massive point clouds. *Graphical Models* 84 (2016), 28–37.
[30] Mingyu Fan, Hong Qiao, and Bo Zhang. 2009. Intrinsic dimension estimation of manifolds by incising balls. *Pattern Recognition* 42, 5 (2009), 780–787.
[31] 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/
[32] 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.
[33] 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.
[34] 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
[35] 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.
[36] 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.
[37] Balázs Kégl. 2002. Intrinsic dimension estimation using packing numbers. *Advances in neural information processing systems* 15 (2002).
[38] 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.
[39] Baki Koyuncu and Erkan Bostancı. 2009. 3D battlefield modeling and simulation of war games. *Communications and Information Technology proceedings* (2009).
[40] Russell LaDuca. 2020. *What would happen to me if I was buried under snow?* https://qr.ae/prt6zQ
[41] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.
[42] 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).
[43] 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).
[44] 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.
[45] 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.
[46] 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.
[47] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (1987), 647–668.
[48] 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
[49] 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.
[50] 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.
[51] 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).
[52] National Park Service. 2022. *Mount Rainier.* https://www.nps.gov/mora/index.htm
[53] National Park Service. 2022. *Mount Rainier Annual Snowfall Totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm
[54] National Park Service. 2022. *Mount Rainier Frequently Asked Questionss.* https://www.nps.gov/mora/faqs.htm
[55] National Weather Service. 2023. *Measuring Snow.* https://www.weather.gov/dvn/snowmeasure
[56] 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.
[57] 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.
[58] 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).
[59] 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
[60] Open Topography. 2022. *USGS 1/3 arc-second Digital Elevation Model.* https://doi.org/10.5069/G98K778D
[61] 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.
[62] 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).
[63] 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.

[64] 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.
[65] 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.
[66] 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.
[67] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering* 2014 (2014).

A SUMMARY OF FREQUENT USED NOTATIONS

Table 3 shows a summary of frequent used notations.

Table 3: 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
$N(p)$	A set of neighbor points of p
$d_E(p, p')$	The Euclidean distance between point p and p'
$\Pi^*(s, t\|C)$	The exact shortest path between s and t passing on the points of C
$\|\Pi^*(s, t\|C)\|$	The length of $\Pi^*(s, t\|C)$
$\Pi(s, t\|C)$	The shortest path between s and t returned by *RC-Oracle*
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
M_{path}	A hash table stores the selected pairs of POIs u and v in P, i.e., a key $\langle u, v\rangle$, and their corresponding exact shortest path $\Pi^*(s, t\|C)$, i.e., a value, on C
M_{POI}	A hash table stores the POI u, i.e., a key, that we do not store all the exact shortest paths in M_{path} from u to other non-processed POIs, and the POI v, i.e., a value, that we use the exact shortest path with v as a source to approximate the shortest path with u as a source
P_{remain}	A set of remaining POIs of P on C that we have not used algorithm *FastFly* to calculate the exact shortest path 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 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
$\Pi^*(s, t\|T)$	The exact shortest path between s and t passing on the faces of T
$\Pi_E(s, t\|T)$	The shortest path between s and t passing on the edges of T
$\Pi'_E(s, t\|T)$	The shortest path between s and t passing on the edges of T where these edges belongs to the faces that $\Pi^*(s, t\|T)$ passes

B COMPARISON OF ALL ALGORITHMS

Table 4 shows a comparison of all algorithms in terms of the oracle construction time, oracle size, and shortest path query time.

C A2A PROXIMITY QUERY ON POINT CLOUDS

Apart from the P2P proximity query on point clouds that we discussed in the main body of this paper, we also present an oracle to answer the *any points-to-any points (A2A) shortest path query* on point clouds based on our oracle *RC-Oracle*. This adapted oracle is similar to the one presented in Section 4, the only difference is that we need to create POIs which has the same coordinate values as all points in the point cloud, then *RC-Oracle* can answer the A2A proximity query on point clouds. In this case, the number of POI becomes N. Thus, for the A2A proximity query on point clouds, the oracle construction time, oracle size, and shortest path query time of *RC-Oracle* are $O(N \log N)$, $O(n)$, and $O(1)$, respectively. For the A2A proximity query on point clouds, *RC-Oracle* always has $|\Pi(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for each pair of points s and t in C. The query time and error rate of both the A2A *kNN* and range query by using *RC-Oracle* are $O(N)$ and $1 + \epsilon$, respectively.

D A2A PROXIMITY QUERY ON *TIN*S

Apart from the P2P proximity query on *TIN*s that we discussed in the main body of this paper, we also present an oracle to answer the *arbitrary points-to-arbitrary points (A2A) shortest path 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 A2A proximity 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 append the line segment (s, u) and (v, t) to 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 for answering the shortest path between s and t.

In general, the number of POI becomes N. Thus, for the A2A proximity query on *TIN*s, the oracle construction time, oracle size, and shortest path query time of *RC-Oracle-Adapt* are $O(N \log N)$, $O(n)$, and $O(1)$, respectively. For the A2A proximity query on *TIN*s, *RC-Oracle-Adapt* always has $|\Pi(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ for

Table 4: Comparison of algorithms with details

	Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error
Oracle	*SE-Oracle-Adapt* [19, 61, 62]	$O(N + nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
	SE-Oracle-Adapt2	$O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Medium	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
	RC-Oracle-Naive	$O(nN \log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Small
	***RC-Oracle* (ours)**	$\boldsymbol{O(N \log N + n \log n)}$	**Small**	$\boldsymbol{O(n)}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Small**
On-the-fly	*CH* [19]	-	N/A	-	N/A	$O(N + N^2)$	Large	No error
	Kaul [35]	-	N/A	-	N/A	$O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large	Small
	Dijk [36]	-	N/A	-	N/A	$O(N + N \log N)$	Medium	Medium
	***FastFly* (ours)**	-	N/A	-	N/A	$\boldsymbol{O(N \log N)}$	**Medium**	**Small**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [30, 37], 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. We include the *TIN* construction time $O(N)$ explicitly for all *TIN* based algorithms / oracles.

each pair of vertices s and t on the faces T, where $\Pi_{adapt}(s, t|T)$ is the shortest path between s and t passing on a conceptual graph returned by *RC-Oracle-Adapt*, where the vertices of this 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 between s and t passing on this conceptual graph. 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 A2A *kNN* and range query by using *RC-Oracle-Adapt* are $O(N)$ and $1 + \epsilon$, respectively.

E EMPIRICAL STUDIES

E.1 Experimental Results for *TIN*s

E.1.1 Experimental Results on the P2P proximity query. We study the P2P proximity queries on *TIN*s. We (1a) 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 (1b) 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 N (scalability test). In Figure 20, Figure 23, Figure 26, Figure 29 and Figure 32, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting n to be 500 and ϵ to be 0.1 for scalability test. *RC-Oracle-Adapt* performs better than all the remaining algorithms in terms of 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 0.04).

Effect of n. In Figure 11, Figure 13, Figure 7, Figure 16 and Figure 18, 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 21, Figure 24, Figure 27, Figure 30 and Figure 33, 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 0.1. 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 ϵ. In Figure 12, Figure 14, Figure 15, Figure 17 and Figure 19, 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 10k and n to be 50. In Figure 22, Figure 25, Figure 28, Figure 31 and Figure 34, 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 usage, 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.

E.1.2 Experimental Results on the A2A proximity query. We study the A2A proximity queries on *TIN*s. We compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt*.

In Figure 35, we tested the A2A proximity query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 5k on a multi-resolution 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* a lot, the oracle construction time, memory usage, oracle size, shortest path query time, all POIs *kNN* query time, and all POIs range query time of *RC-Oracle* still perform much better than the best-known oracle *SE-Oracle-Adapt*, and other adapted algorithms / oracles. The oracle construction time of *EAR-Oracle* is up to 10^4 times larger than that of *RC-Oracle*, even though *EAR-Oracle* is deliberately designed for A2A proximity queries on *TIN*s, its performance is not well compared with *RC-Oracle*.

E.2 Experimental Results for Point Clouds

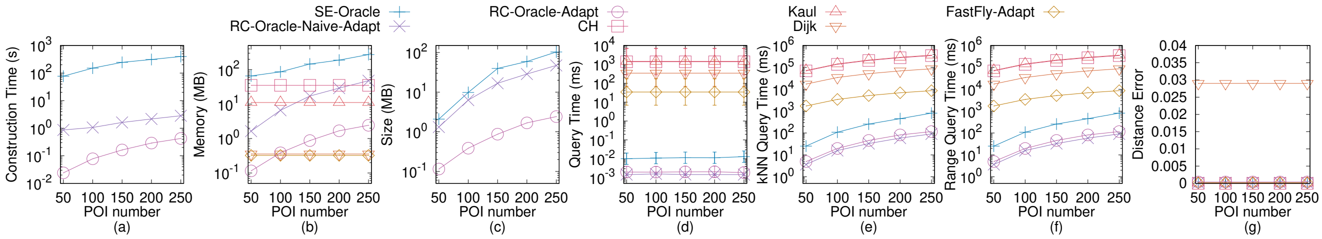


Figure 11: Effect of n on BH_t*-small TIN* dataset (P2P proximity query)

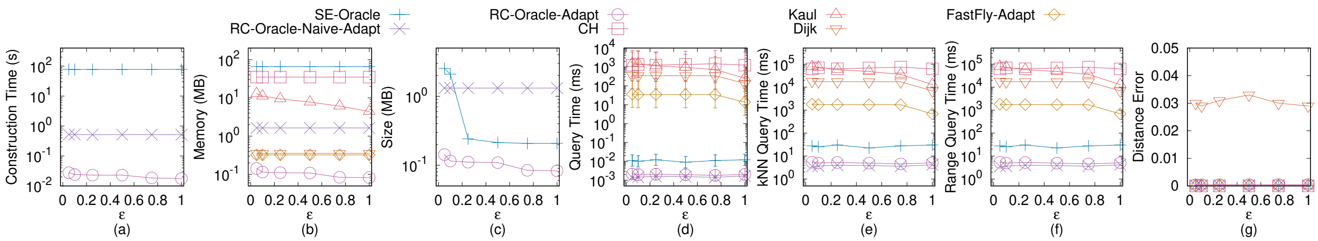


Figure 12: Effect of ϵ on BH_t*-small TIN* dataset (P2P proximity query)

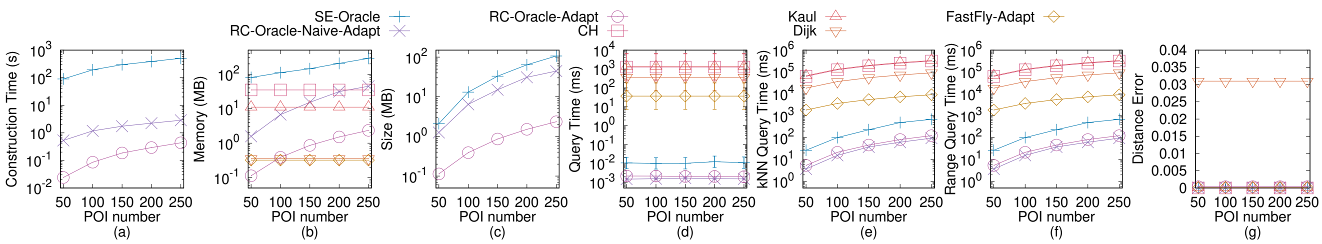


Figure 13: Effect of n on EP_t*-small TIN* dataset (P2P proximity query)

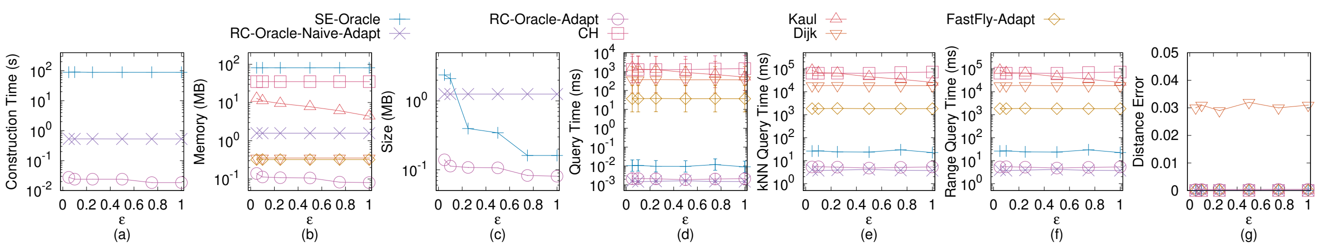


Figure 14: Effect of ϵ on EP_t*-small TIN* dataset (P2P proximity query)

E.2.1 Experimental Results on the P2P proximity query. We study the P2P proximity queries on *point clouds*. We (1a) compared *SE-Oracle-Adapt*, *SE-Oracle-Adapt2*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets with default 50 POIs, and (1b) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets with default 500 POIs.

Effect of N (scalability test). In Figure 46, Figure 49, Figure 52, Figure 55 and Figure 58, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting n to be 500 and ϵ to be 0.1 for scalability test. *RC-Oracle* performs better than all the remaining algorithms in terms of oracle construction time, oracle size, and shortest path query time.

Effect of n. In Figure 36, Figure 39, Figure 41, Figure 43 and Figure 44, 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 47, Figure 50, Figure 53, Figure 56 and Figure 59, 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 0.1. 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 ϵ. In Figure 37, Figure 40, Figure 42, Figure 8 and Figure 45, 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 10k and n to be 50. In Figure 48, Figure 51, Figure 54, Figure 57 and Figure 60,

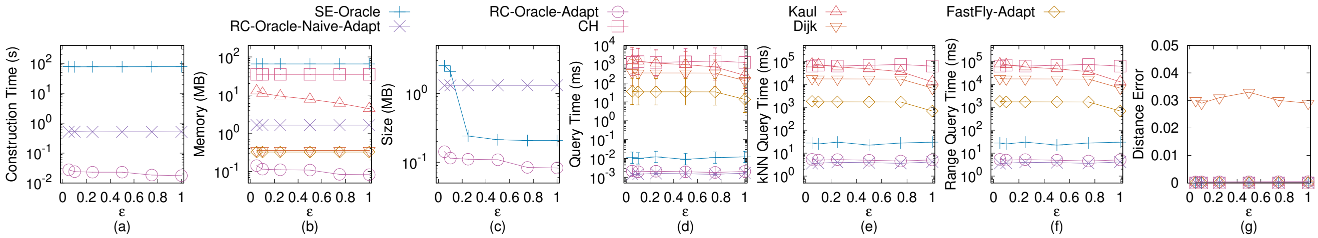


Figure 15: Effect of ϵ on GF_t-small TIN dataset (P2P proximity query)

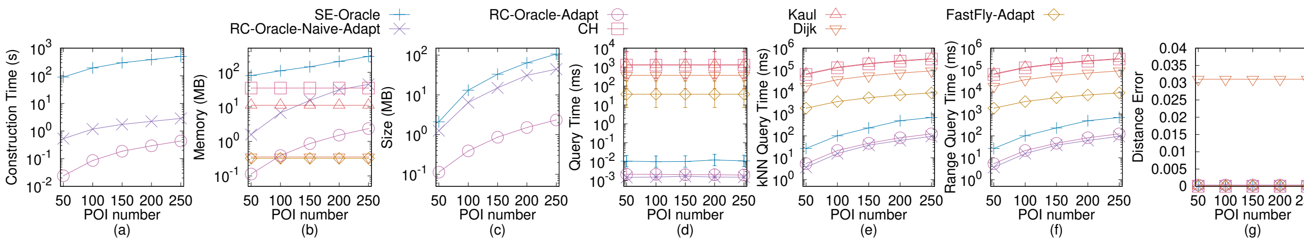


Figure 16: Effect of n on LM_t-small TIN dataset (P2P proximity query)

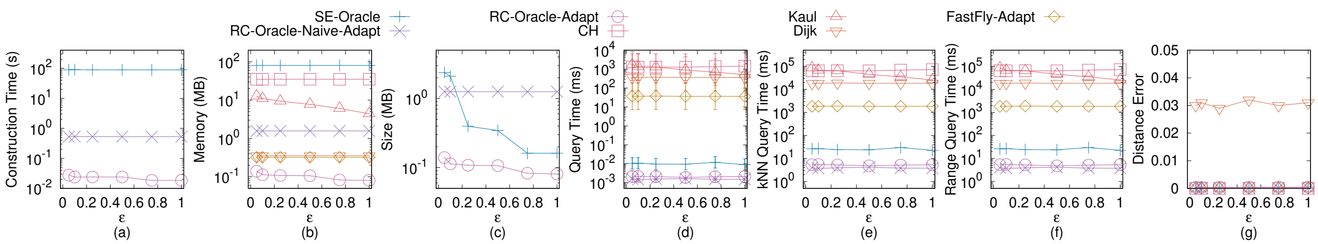


Figure 17: Effect of ϵ on LM_t-small TIN dataset (P2P proximity query)

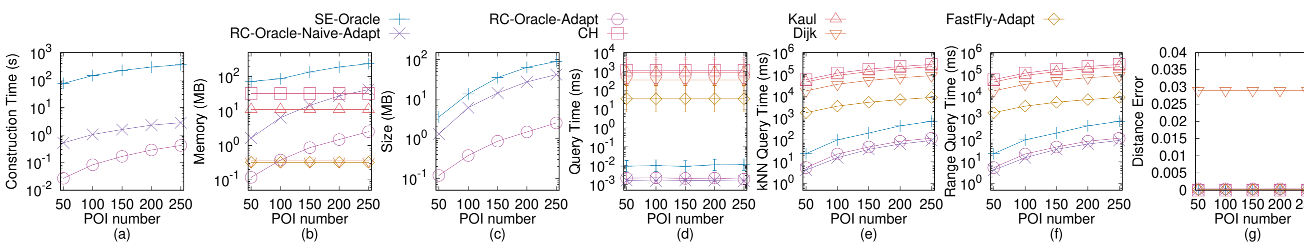


Figure 18: Effect of n on RM_t-small TIN dataset (P2P proximity query)

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 usage, oracle size, shortest path query time, all POIs *kNN* query time, and all POIs range query time of *RC-Oracle* still perform much better than the best-known oracle *SE-Oracle-Adapt*, and other algorithms / oracles.

E.2.2 Experimental Results on the A2A proximity query. We study the A2A proximity queries on point clouds. We compared *SE-Oracle-Adapt*, *SE-Oracle-Adapt2*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly*.

In Figure 61, we tested the A2A proximity query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a multi-resolution 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 other adapted algorithms / oracles.

E.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 [45, 61, 62]. 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

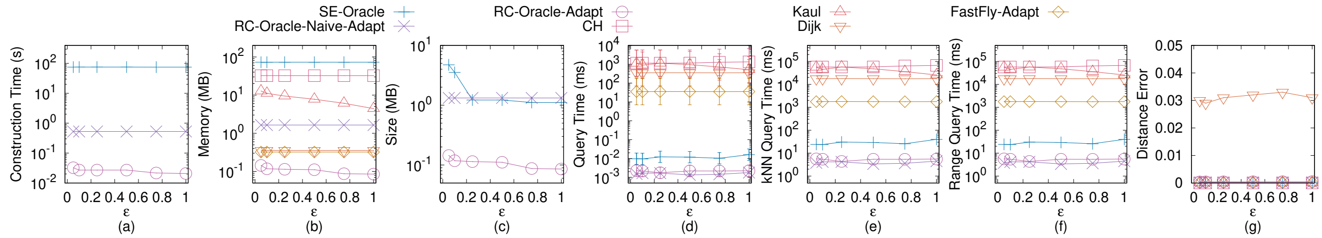


Figure 19: Effect of ϵ on *RM_t-small TIN* dataset (P2P proximity query)

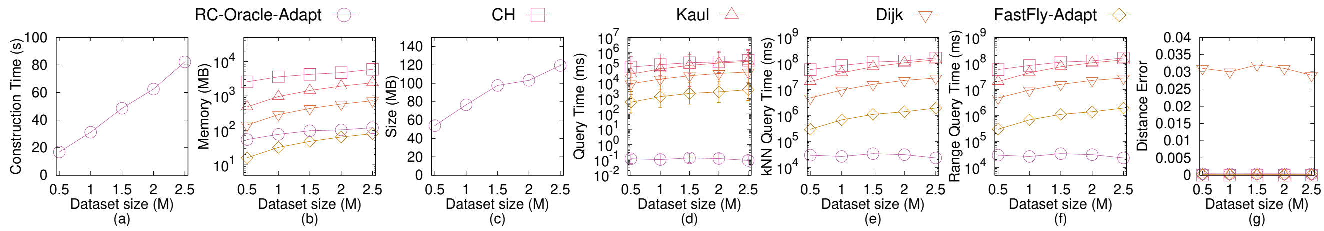


Figure 20: Effect of N on *BH_t TIN* dataset (P2P proximity query)

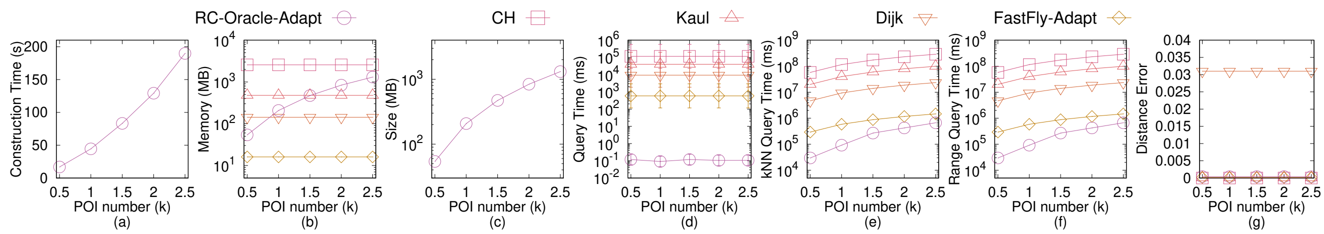


Figure 21: Effect of n on *BH_t TIN* dataset (P2P proximity query)

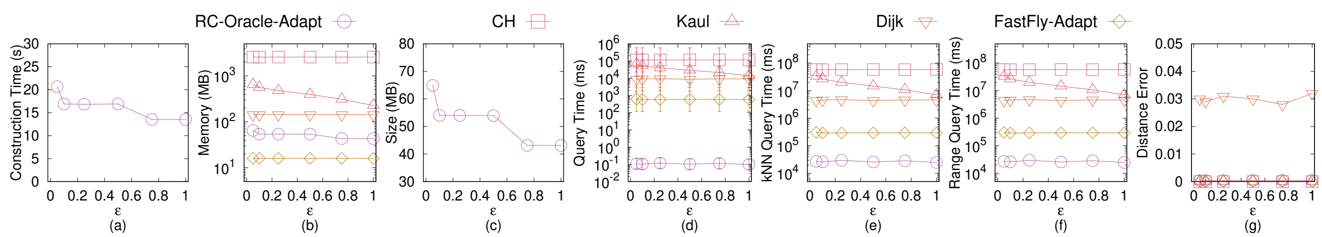


Figure 22: Effect of ϵ on *BH_t TIN* dataset (P2P proximity query)

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 $\sigma_N^2 = (\frac{\sum_{q_o \in C_o} (x_{q_o} - \overline{x})^2}{qx_o \cdot qy_o}, \frac{\sum_{q_o \in C_o} (y_{q_o} - \overline{y})^2}{qx_o \cdot qy_o})$. 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.

F PROOF

Proof of Theorem 4.2. Firstly, we prove the *oracle construction time* of *RC-Oracle*.

- In **POIs sorting** step, it needs $O(n \log)$ time. By using quick sort, we can sort n POIs in $O(n \log)$ time.
- In **shortest path calculation** step, it needs $O(N \log N + n)$ time. Since it needs to run algorithm *FastFly* for $O(1)$ times since according to standard packing property [32], we just need to use $O(1)$ POIs as a source to use algorithm *FastFly* for exact shortest path calculation (which is also shown by our experiment), and it needs $O(N \log N)$ time. For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly* for exact shortest path calculation, we just need to calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for shortest path approximation, and it needs $O(n)$ time. So the total running time is $O(N \log N + n)$.

So the oracle construction time of *RC-Oracle* is $O(N \log N + n \log n)$.

Secondly, we prove the *oracle size* of *RC-Oracle*. There are two components in *RC-Oracle*, i.e., M_{path} and M_{POI}. For M_{POI}, its size is at most $O(n)$. Thus, we mainly focus on the size of M_{path}. we just need to use $O(1)$ POIs as a source to use algorithm *FastFly*

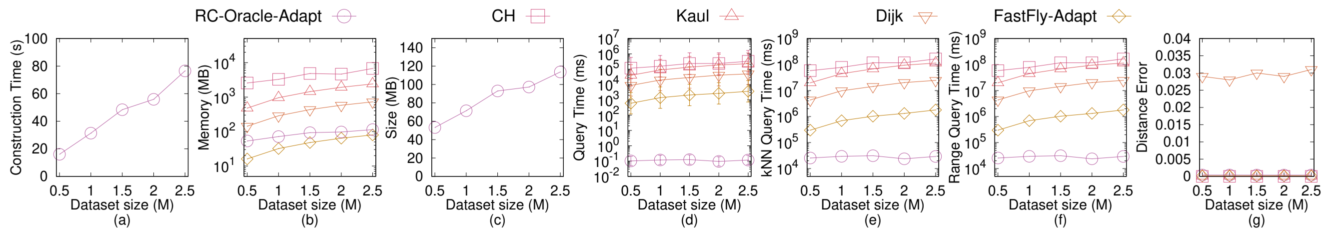


Figure 23: Effect of N on EP_t *TIN* dataset (P2P proximity query)

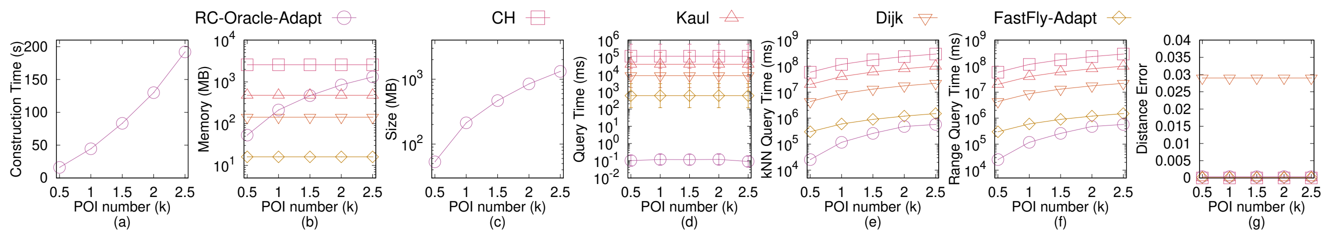


Figure 24: Effect of n on EP_t *TIN* dataset (P2P proximity query)

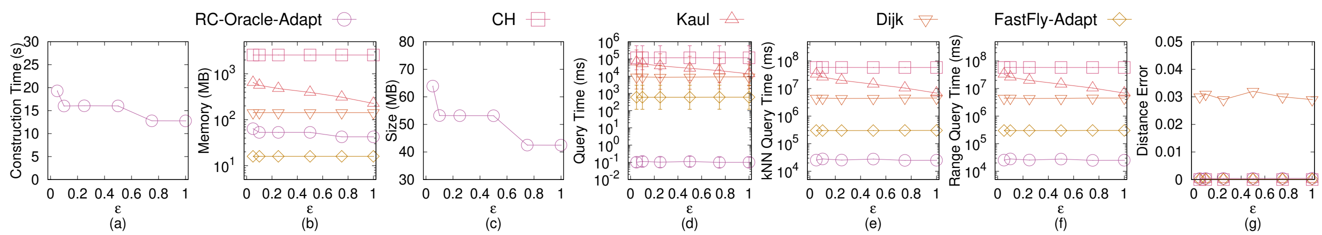


Figure 25: Effect of ϵ on EP_t *TIN* dataset (P2P proximity query)

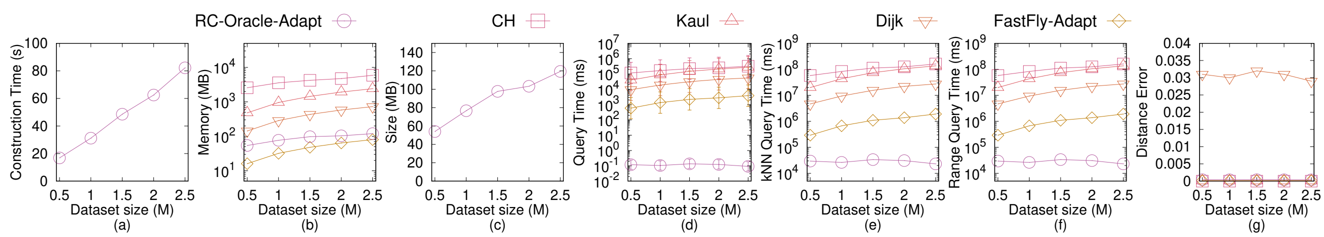


Figure 26: Effect of N on GF_t *TIN* dataset (P2P proximity query)

for the exact shortest path calculation. Given that there are total n POIs, so there are $O(n)$ shortest paths for $O(1)$ POIs. For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly* to all POIs for exact shortest path calculation, we will not store the exact shortest paths between these POIs to other POIs in M_{path}, since they are approximated by the other exact shortest paths stored in M_{path}. For these POIs, we just need to calculate the exact shortest paths between themselves with size $O(1)$, so there are $O(n)$ shortest paths for these POIs. In total, there are $O(n)$ exact shortest paths stored in M_{path}. So we obtain the oracle size is $O(n)$.

Thirdly, we prove the *shortest path query time* of *RC-Oracle*. If $\Pi^*(s,t|C)$ exists in M_{path}, the shortest path query time is $O(1)$. If $\Pi^*(s,t|C)$ does not exist in 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* of *RC-Oracle*. 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 approximate shortest path $\Pi(s,t|C)$ is calculated by appending $\Pi^*(s,u|C)$ and $\Pi^*(u,t|C)$. This means that $d_E(s,t) > \frac{2}{\epsilon}\cdot\Pi^*(u,s|C)$, since we will only use $\Pi^*(s,u|C)$ and $\Pi^*(u,t|C)$ to approximate $\Pi(s,t|C)$ when this condition satisfies. According to [61, 62], the distance of the path on a *TIN* is a metric, and it satisfies the triangle inequality. Since the path on the point cloud will only pass on the points, which is a sub-type of the path on a *TIN*, so the distance of the path on point also satisfies the triangle inequality. So we have $|\Pi^*(s,u|C)| + |\Pi^*(u,t|C)| < |\Pi^*(s,u|C)| + |\Pi^*(u,s|C)| +$

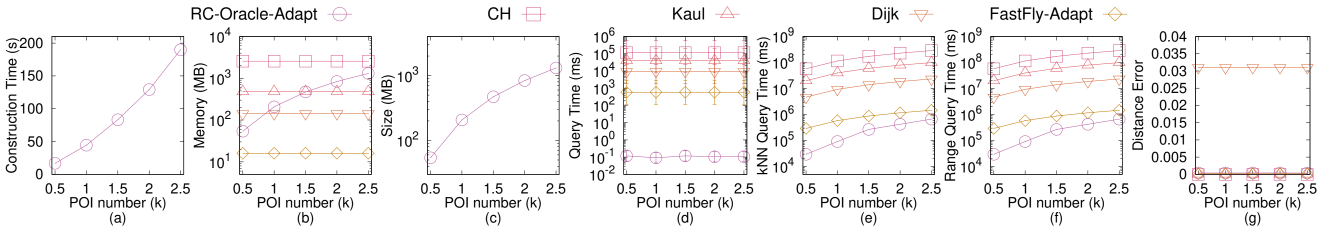


Figure 27: Effect of n on GF_t TIN dataset (P2P proximity query)

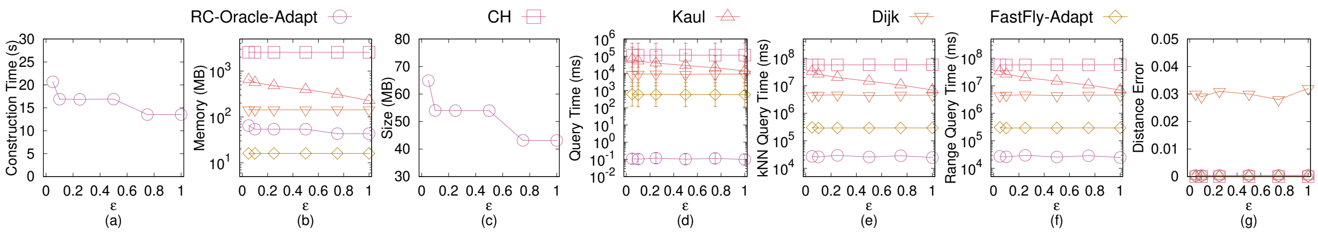


Figure 28: Effect of ϵ on GF_t TIN dataset (P2P proximity query)

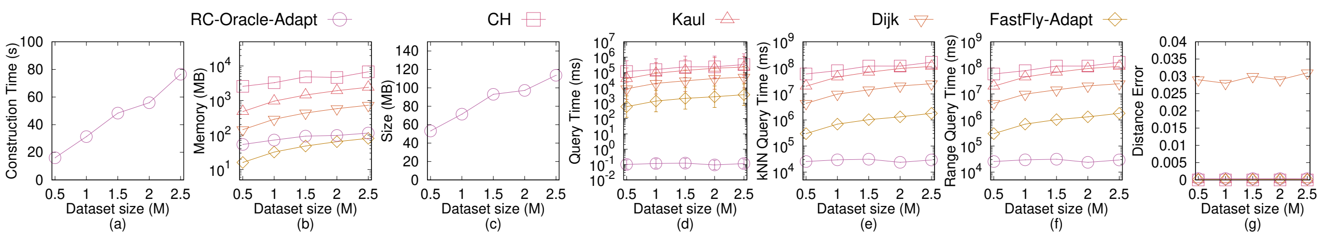


Figure 29: Effect of N on LM_t TIN dataset (P2P proximity query)

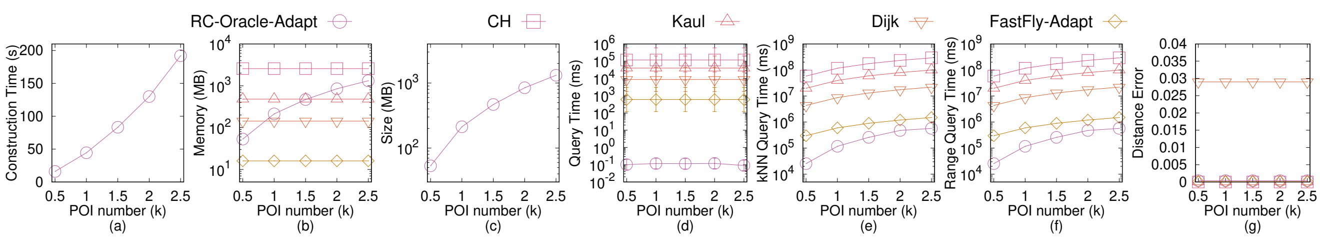


Figure 30: Effect of n on LM_t TIN dataset (P2P proximity query)

$|\Pi^*(s,t|C)| = |\Pi^*(s,t|C)| + 2 \cdot |\Pi^*(u,s|C)| < |\Pi^*(s,t|C)| + \epsilon \cdot d_E(s,t) \leq |\Pi^*(s,t|C)| + \epsilon \cdot |\Pi^*(s,t|C)| = (1+\epsilon)|\Pi^*(s,t|C)|$. 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 Euclidean distance between two points is no larger than the distance of the shortest path on the point cloud between the same two points. The final equation is due to the distributive law of multiplication. □

Proof of Lemma 4.5. Firstly, we prove the query time of both the *kNN* and range query algorithm. Given a query POI, when we need to perform the *kNN* query or the range query, we need to check the distance between this query POI to all other POIs using the shortest path query phase of *RC-Oracle* in $O(1)$ time. Since there are total n POIs, the query time is $O(n)$.

Secondly, we prove the error rate of both the *kNN* and range query algorithm. Recall that we let v_f (resp. v'_f) be the furthest POI 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 POI 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

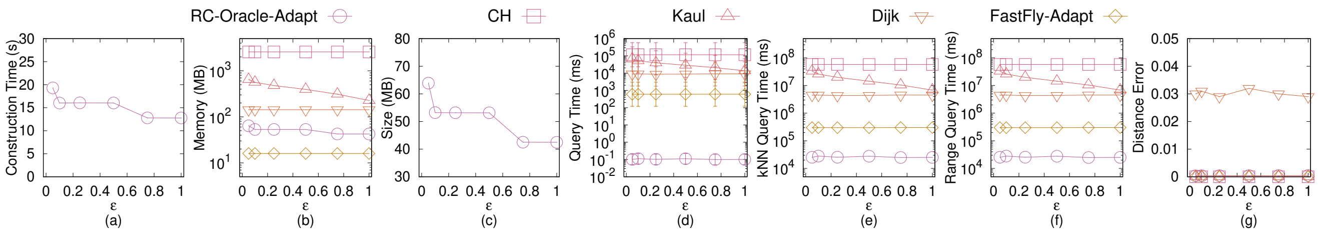


Figure 31: Effect of ϵ on LM_t *TIN* dataset (P2P proximity query)

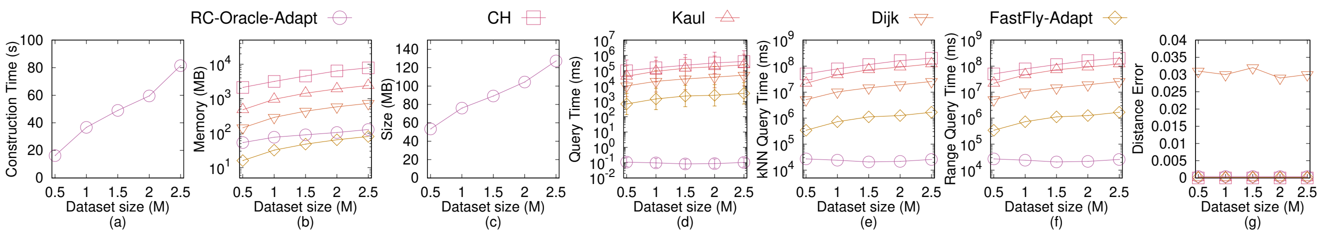


Figure 32: Effect of N on RM_t *TIN* dataset (P2P proximity query)

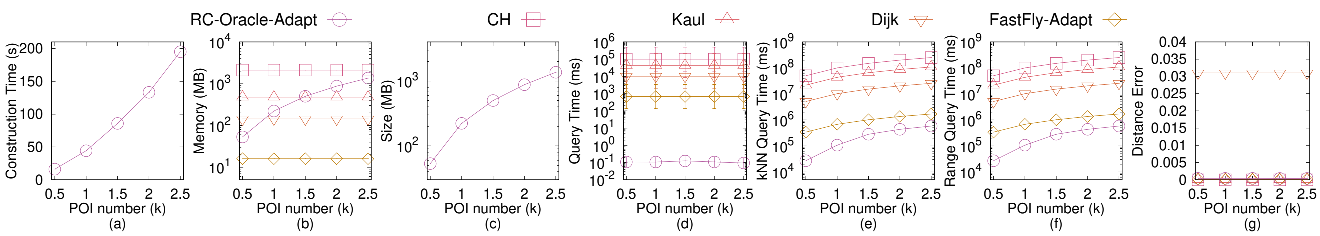


Figure 33: Effect of n on RM_t *TIN* dataset (P2P proximity query)

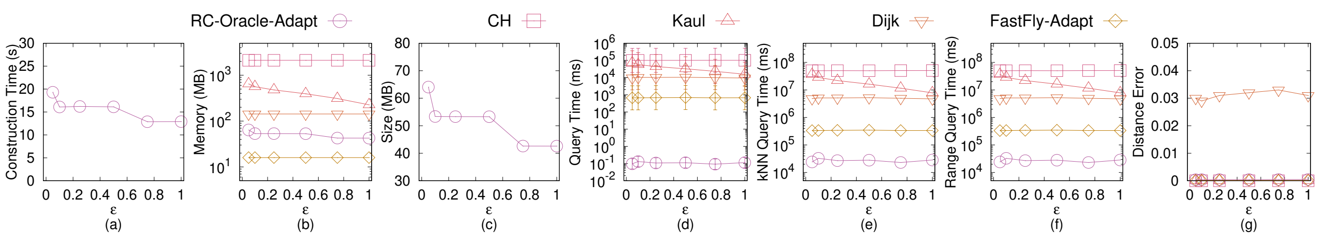


Figure 34: Effect of ϵ on RM_t *TIN* dataset (P2P proximity query)

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

Proof of Lemma 4.3. We let $\Pi'_E(s,t|T)$ be the shortest path between s and t passing on the edges of T where these edges belong to the faces that $\Pi^*(s,t|T)$ passes. According to left hand side equation in Lemma 2 of work [36], we have $\lambda \cdot |\Pi'_E(s,t|T)| \leq |\Pi^*(s,t|T)|$, where $\lambda = \min\{\frac{\sin\theta}{2}, \sin\theta\cos\theta\}$. Even though both $\Pi'_E(s,t|T)$ and $\Pi_E(s,t|T)$ pass on the edges of T, according to work [56] in Section 3.1, $\Pi_E(s,t|T)$ may pass different face of $\Pi'_E(s,t|T)$. Since $\Pi'_E(s,t|T)$ passes on the edges on T where these edges belong to the faces that $\Pi^*(s,t|T)$ passes, it may not be the shortest path passing on the edges considering all the edges on T. But, $\Pi_E(s,t|T)$ is the shortest path passing on the edges of a *TIN* that considering all the edges on T, so $|\Pi_E(s,t|T)| \leq |\Pi'_E(s,t|T)|$. 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. Since $\Pi^*(s,t|C)$ passes on points of C, and $\Pi_E(s,t|T)$ passes on edges of T, for a point / vertex u, if its next searching

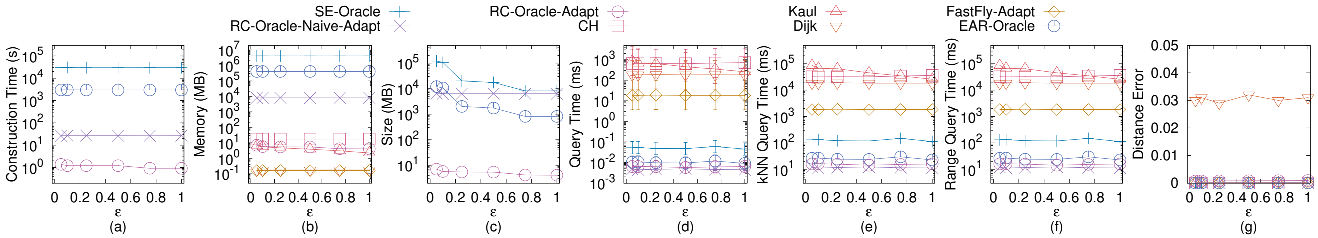


Figure 35: A2A proximity query for *TIN*

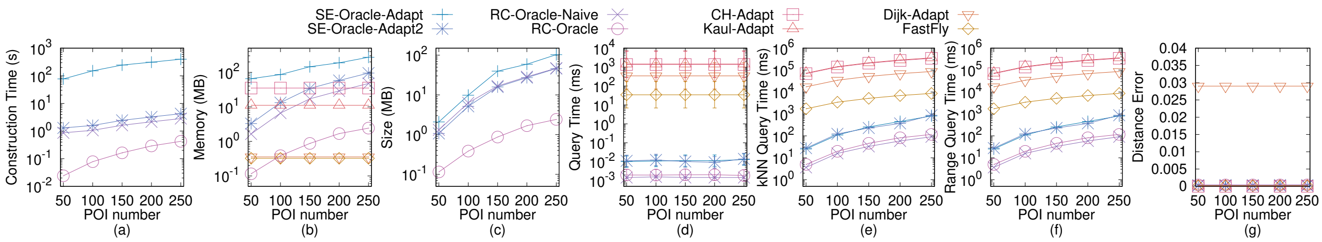


Figure 36: Effect of n on BH_p-*small* point cloud dataset (P2P proximity query)

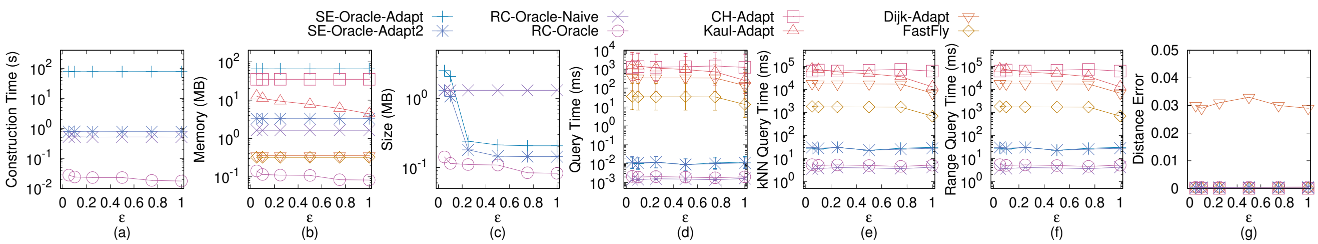


Figure 37: Effect of ϵ on BH_p-*small* point cloud dataset (P2P proximity query)

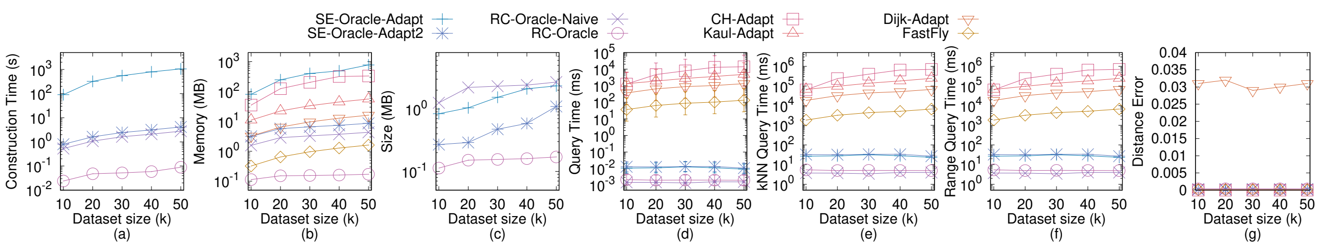


Figure 38: Effect of N on EP_p-*small* point cloud dataset (P2P proximity query)

point / vertex v is its diagonal neighbor point / vertex, $\Pi^*(s,t|C)$ can directly connect u and v ($\Pi^*(s,t|C)$ can directly connect s and q in Figure 2 (a)), but $\Pi_E(s,t|T)$ needs one more vertex to connect u and v ($\Pi_E(s,t|T)$ needs one more vertex to connect s and q in Figure 2 (b)), so $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$. Thus, by combining $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$, $|\Pi_E(s,t|T)| \leq |\Pi'_E(s,t|T)|$, and $|\Pi_V(s,t|C)| \leq \frac{1}{\lambda} \cdot |\Pi^*(s,t|T)|$, we have $|\Pi^*(s,t|C)| \leq k \cdot |\Pi^*(s,t|T)|$, where $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$. □

Proof of Lemma 4.4. 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 black green q on T, it can only connect with one of its 6 blue neighbor vertices. Since $\Pi^*(s,t|C)$ passes on points of C, and $\Pi_E(s,t|T)$ passes on edges of T, for a point / vertex u, if its next searching point / vertex v is its diagonal neighbor point / vertex, $\Pi^*(s,t|C)$ can directly connect u and v ($\Pi^*(s,t|C)$ can directly connect s and q in Figure 2 (a)), but $\Pi_E(s,t|T)$ needs one more vertex to connect u and v ($\Pi_E(s,t|T)$ needs one more vertex to connect s and q in Figure 2 (b)), so $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$. □

Theorem F.1. *The shortest path query time and memory usage of algorithm CH are $O(N + N^2)$ and $O(N)$, respectively. Algorithm CH returns the exact shortest path passing on the faces of a TIN that is constructed by the point cloud.*

Proof. Firstly, we prove the *shortest path query time* of algorithm *CH*. The proof of the shortest path query time of algorithm *CH* is in [19]. But since algorithm *CH* first needs to construct the

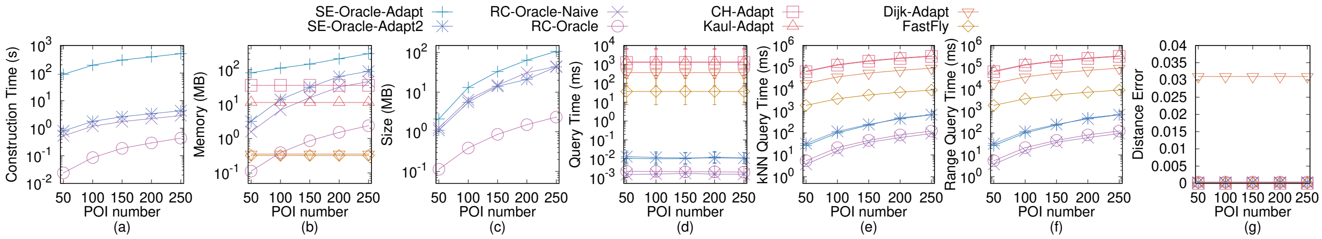


Figure 39: Effect of n on EP_p-*small* point cloud dataset (P2P proximity query)

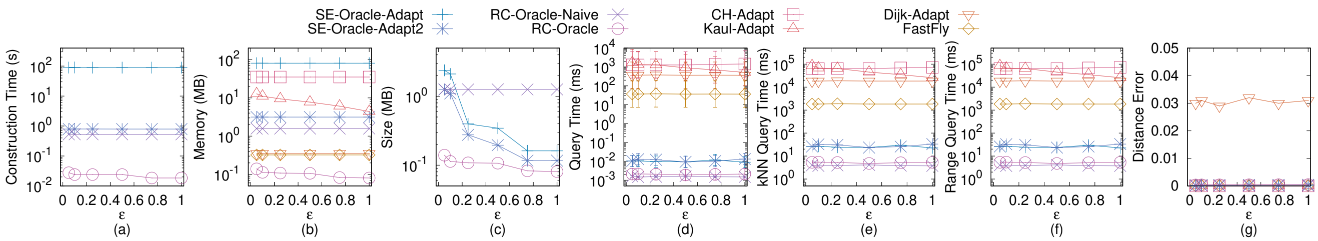


Figure 40: Effect of ϵ on EP_p-*small* point cloud dataset (P2P proximity query)

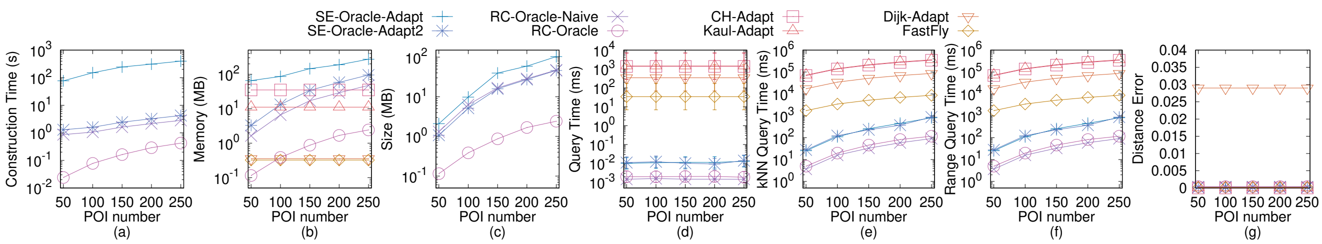


Figure 41: Effect of n on GF_p-*small* point cloud dataset (P2P proximity query)

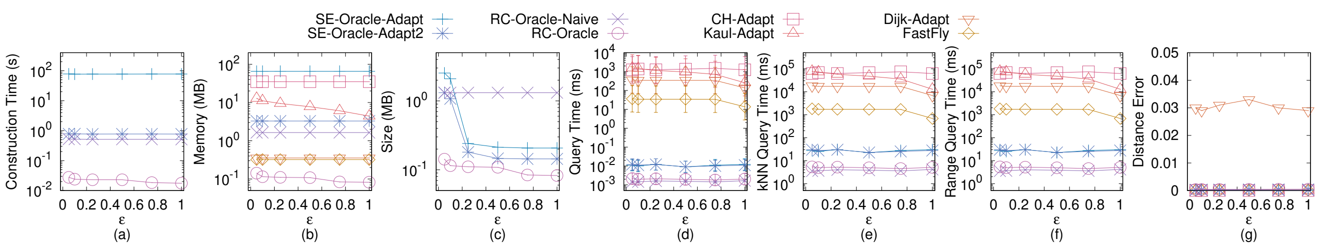


Figure 42: Effect of ϵ on GF_p-*small* point cloud dataset (P2P proximity query)

TIN using the point cloud, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *CH* is $O(N + N^2)$.

Secondly, we prove the *memory usage* of algorithm *CH*. The proof of the memory usage of algorithm *CH* is in [19]. Thus, the memory usage of algorithm *CH* is $O(N)$.

Thirdly, we prove the *error bound* of algorithm *CH*. The proof that algorithm *CH* returns the exact shortest path on the *TIN* is in [19]. Since the *TIN* is constructed by the point cloud, so algorithm *CH* returns the exact shortest path passing on the faces of a *TIN* that is constructed by the point cloud. □

Theorem F.2. *The shortest path query time and memory usage of algorithm Kaul are* $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ *and* $O(N)$*, respectively. Algorithm Kaul always has* $|\Pi_{Kaul}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs* s *and* t *in* P*, where* $\Pi_{Kaul}(s,t|T)$ *is the shortest path of algorithm Kaul between* s *and* t *passing on the faces of a TIN* T *that is constructed by the point cloud.*

Proof. Firstly, we prove the *shortest path query time* of algorithm *Kaul*. The proof of the shortest path query time of algorithm *Kaul* is in [35]. Note that in Section 4.2 of [35], the shortest path query time of algorithm *Kaul* is $O((N + N')(\log(N + N') + (\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where $N' = O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ and K is a parameter which is a positive number at least 1. By Theorem 1 of [35], we obtain that its error bound ϵ is equal to $\frac{1}{K-1}$.

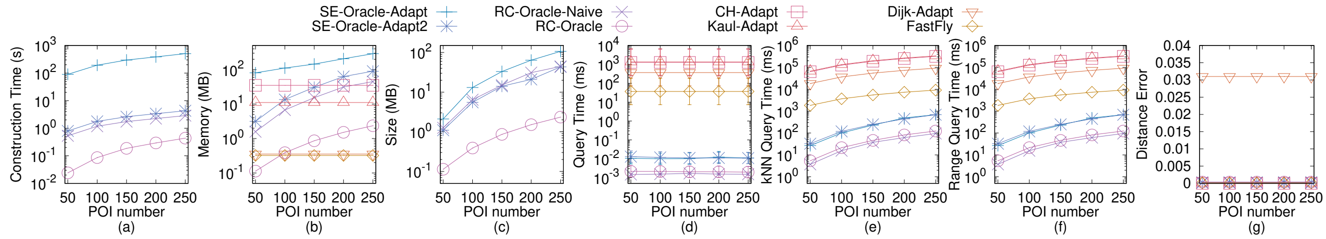


Figure 43: Effect of n on LM_p-*small* point cloud dataset (P2P proximity query)

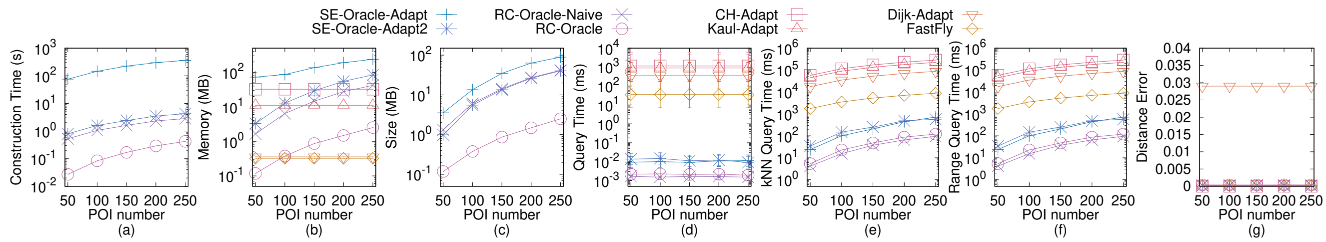


Figure 44: Effect of n on RM_p-*small* point cloud dataset (P2P proximity query)

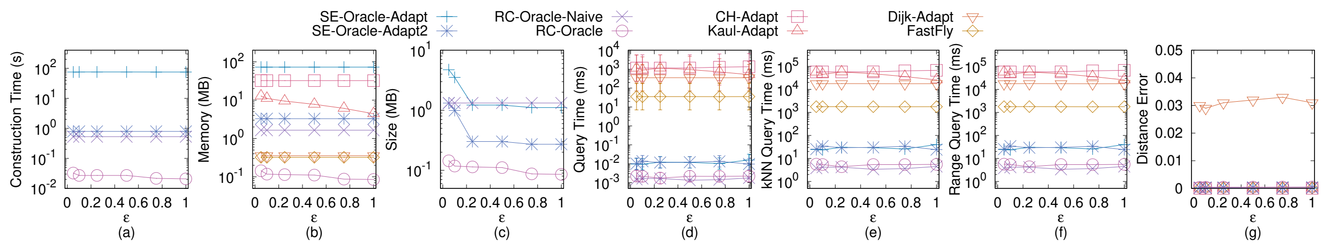


Figure 45: Effect of ϵ on RM_p-*small* point cloud dataset (P2P proximity query)

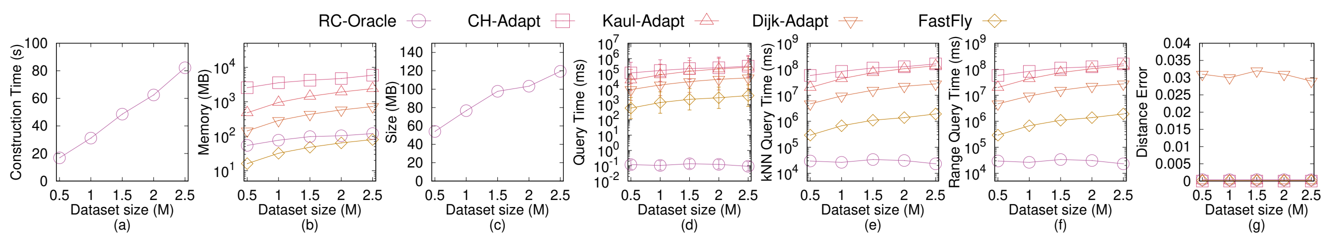


Figure 46: Effect of N on BH_p point cloud dataset (P2P proximity query)

Thus, we can derive that the shortest path query time of algorithm *Kaul* 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* 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 *CH* 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* is $O(N+\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$.

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

Thirdly, we prove the *error bound* of algorithm *Kaul*. The proof of the error bound of algorithm *Kaul* is in [35]. Since the *TIN* is constructed by the point cloud, so algorithm *Kaul* always has $|\Pi_{Kaul}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem F.3. *The shortest path query time and memory usage of algorithm Dijk are $O(N+N\log N)$ and $O(N)$, respectively. Algorithm Dijk always has $|\Pi_{Dijk}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P, where $\Pi_{Dijk}(s,t|T)$ is the shortest path of algorithm Dijk between s and t passing on the faces of a TIN T that is constructed by the point cloud.*

Proof. Firstly, we prove the *shortest path query time* of algorithm *Dijk*. Since algorithm *Dijk* only calculates the paths passing

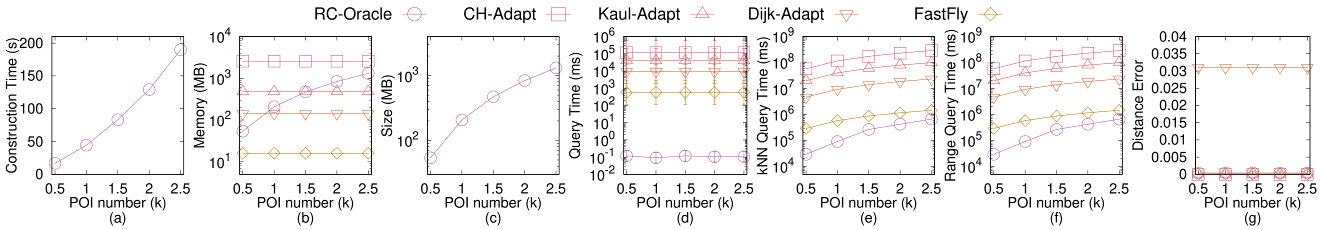


Figure 47: Effect of n on BH_p point cloud dataset (P2P proximity query)

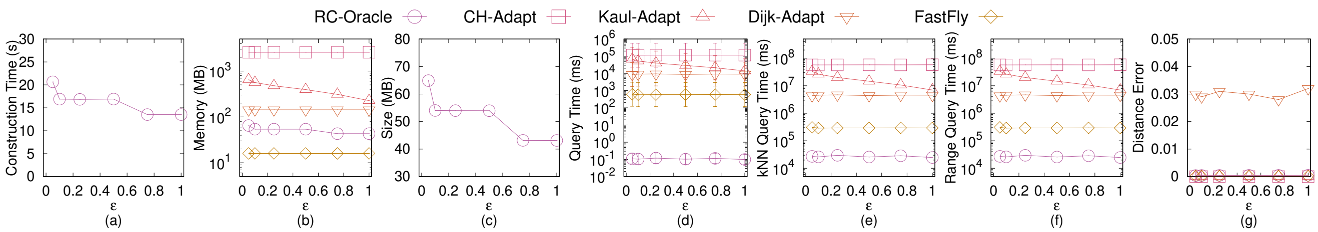


Figure 48: Effect of ϵ on BH_p point cloud dataset (P2P proximity query)

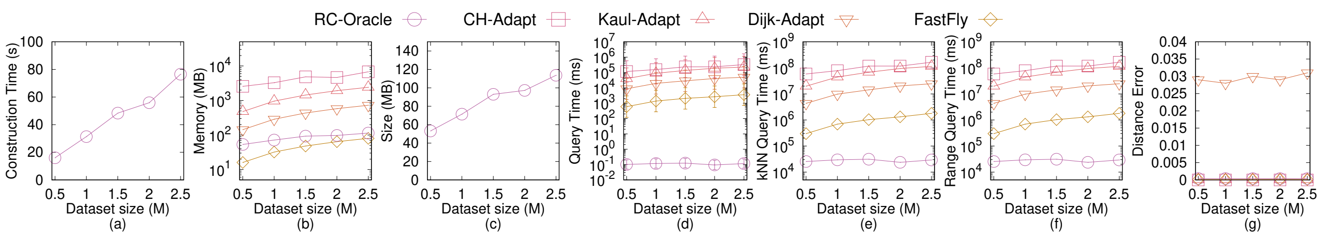


Figure 49: Effect of N on EP_p point cloud dataset (P2P proximity query)

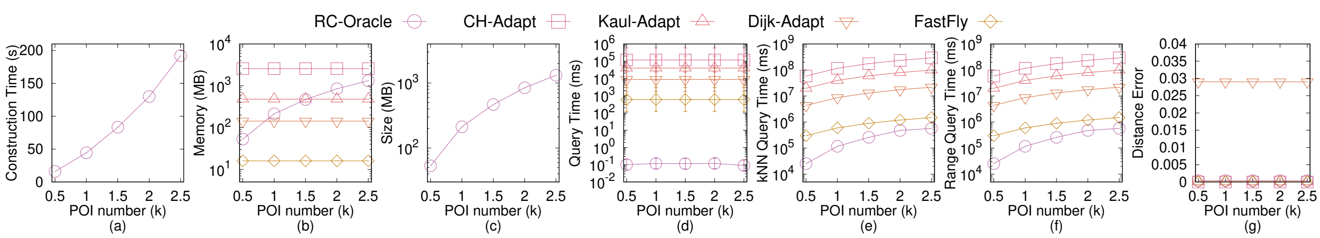


Figure 50: Effect of n on EP_p point cloud dataset (P2P proximity query)

on the edges of the *TIN* 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* 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 *Dijk* is $O(N + N \log N)$.

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

Thirdly, we prove the *error bound* of algorithm *Dijk*. Recall that $\Pi_E(s,t|T)$ is the shortest path between s and t passing on the edges of a *TIN* T that is constructed by the point cloud, so actually $\Pi_E(s,t|T)$ is the same as $\Pi_{Dijk}(s,t|T)$. In Lemma 4.4, we have $|\Pi^*(s,t|C)| \leq |\Pi_E(s,t|T)|$, so we obtain that algorithm *Dijk* always has $|\Pi_{Dijk}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P. □

Theorem F.4. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt are $O(N + nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. SE-Oracle-Adapt always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P, where $\Pi_{SE\text{-}Oracle\text{-}Adapt}(s,t|T)$ is the shortest path of SE-Oracle-Adapt between s and t passing on the faces of a TIN T that is constructed by the point cloud.*

Proof. Firstly, we prove the *oracle construction time* of *SE-Oracle-Adapt*. The oracle construction time of the original oracle in [61, 62] (after pre-computing the shortest path between each pair of POIs) is

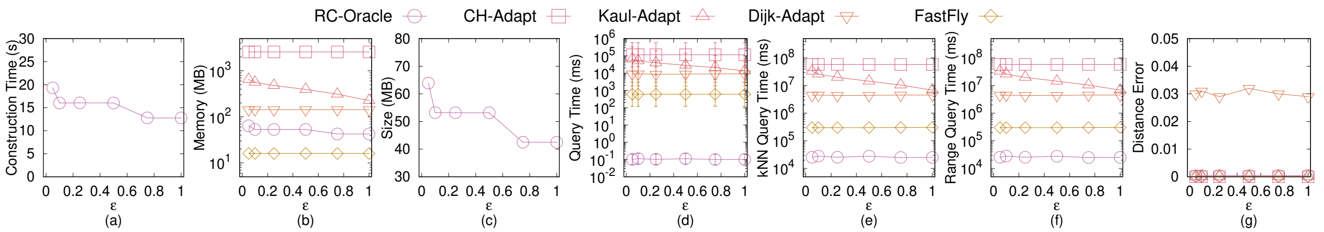


Figure 51: Effect of ϵ on EP_p point cloud dataset (P2P proximity query)

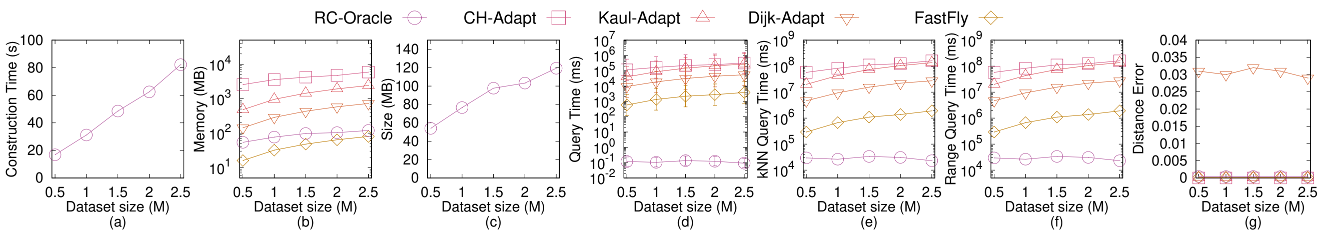


Figure 52: Effect of N on GF_p point cloud dataset (P2P proximity query)

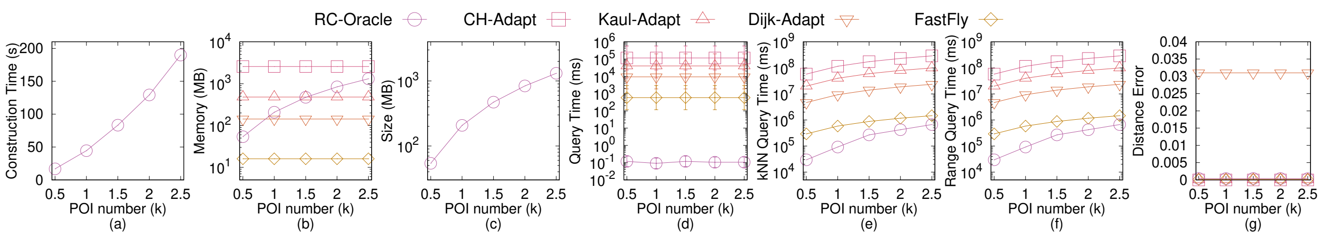


Figure 53: Effect of n on GF_p point cloud dataset (P2P proximity query)

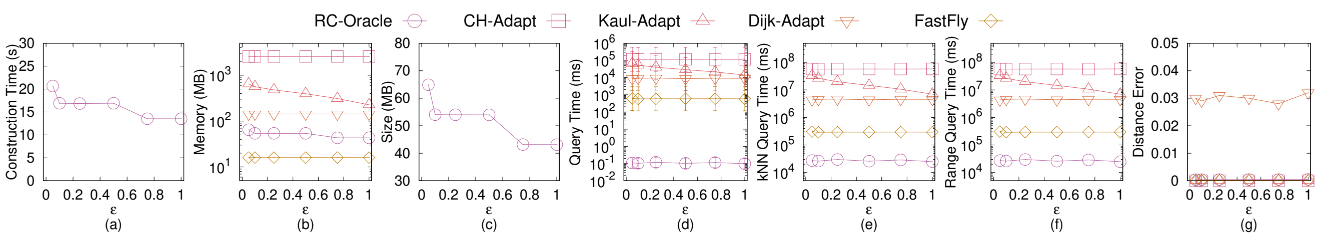


Figure 54: Effect of ϵ on GF_p point cloud dataset (P2P proximity query)

$O(nm+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt*, we use algorithm *CH* for the point cloud shortest path query, which has shortest path query time $O(N+N^2)$ according to Theorem F.1. But, we just need to construct the *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *SE-Oracle-Adapt* only needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt* is $O(N+nN^2+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$.

Secondly, we prove the *oracle size* of *SE-Oracle-Adapt*. The proof of the oracle size of *SE-Oracle-Adapt* is in [61, 62]. 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 [61, 62]. 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 shortest path passing on the faces of a *TIN* that is constructed by the point cloud according to Theorem F.1, so the error of *SE-Oracle-Adapt* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt* is in [61, 62]. Since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem F.5. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt2 are* $O(N + nN\log N +$

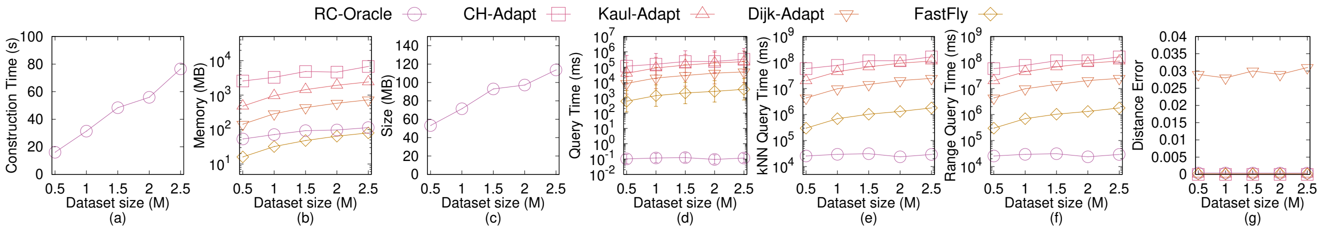


Figure 55: Effect of N on LM_p point cloud dataset (P2P proximity query)

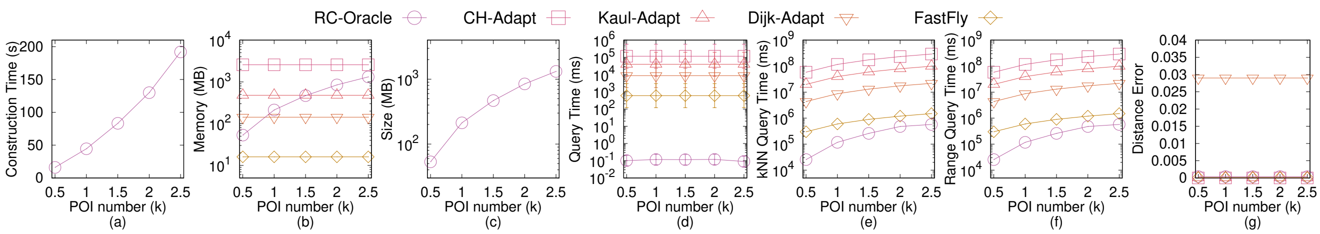


Figure 56: Effect of n on LM_p point cloud dataset (P2P proximity query)

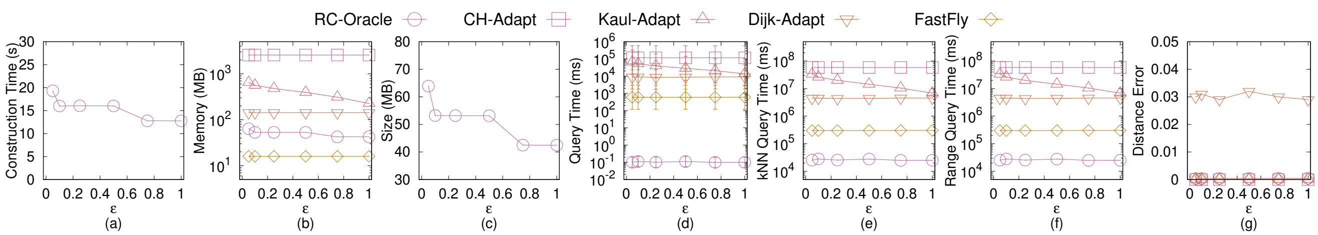


Figure 57: Effect of ϵ on LM_p point cloud dataset (P2P proximity query)

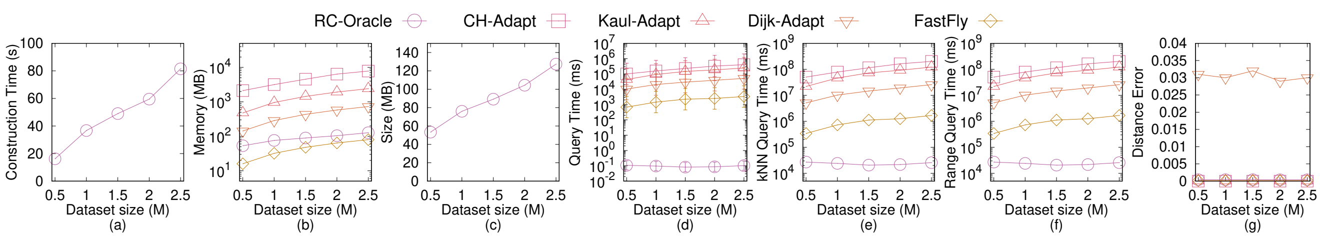


Figure 58: Effect of N on RM_p point cloud dataset (P2P proximity query)

$\frac{nh}{\epsilon^{2\beta}} + nh\log n$), $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. SE-Oracle-Adapt2 always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt2}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P, where $\Pi_{SE\text{-}Oracle(Edge)}(s,t|C)$ is the shortest path of SE-Oracle(Edge) between s and t passing on points of the point cloud C.

Proof. Firstly, we prove the *oracle construction time* of *SE-Oracle-Adapt2*. The oracle construction time of the original oracle in [61, 62] (after pre-computing the shortest path between each pair of POIs) is $O(nm + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt2*, we use algorithm *FastFly* for the point cloud shortest path query, which has 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 *SE-Oracle-Adapt2* is $O(N + nN\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$.

Secondly, we prove the *oracle size* of *SE-Oracle-Adapt2*. The proof of the oracle size of *SE-Oracle-Adapt2* is in [61, 62]. Thus, the oracle size of *SE-Oracle-Adapt2* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the *shortest path query time* of *SE-Oracle-Adapt2*. The proof of the shortest path query time of *SE-Oracle-Adapt2* is in [61, 62]. Thus, the shortest path query time of *SE-Oracle-Adapt2* is $O(h^2)$.

Fourthly, we prove the *error bound* of *SE-Oracle-Adapt2*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt2* is algorithm *FastFly*, which returns the exact shortest path passing on points of the point cloud according to Theorem 4.1, the error of *SE-Oracle-Adapt2* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt2* is in [61, 62]. So we obtain that *SE-Oracle-Adapt2* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq$

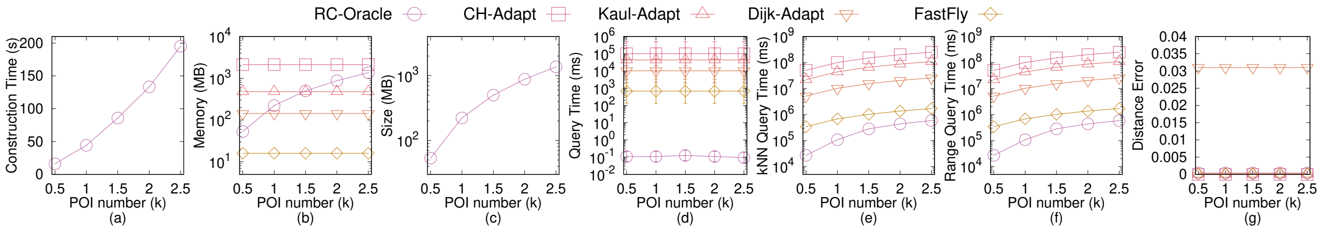


Figure 59: Effect of n on RM_p point cloud dataset (P2P proximity query)

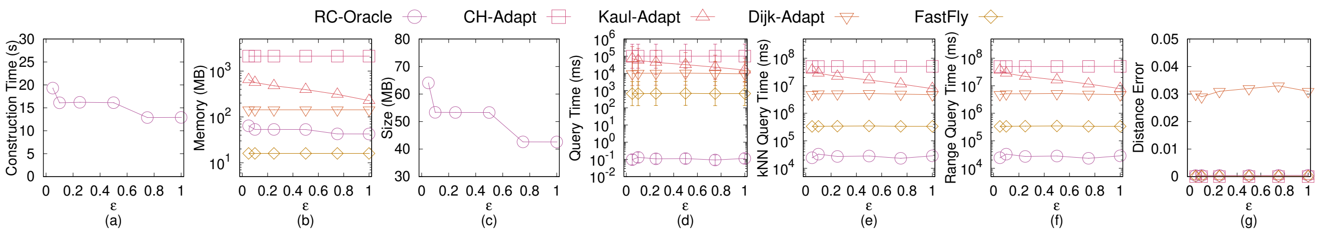


Figure 60: Effect of ϵ on RM_p point cloud dataset (P2P proximity query)

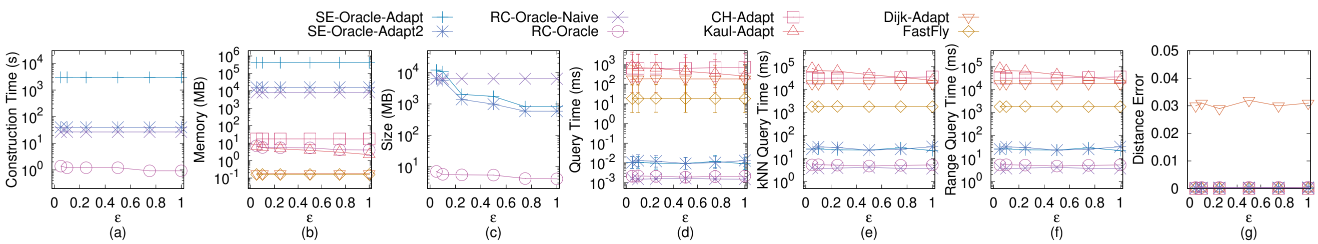


Figure 61: A2A proximity query for *Point-Edge* type

$|\Pi_{SE\text{-}Oracle\text{-}Adapt2}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P □

Theorem F.6. *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 $O(1)$, respectively. RC-Oracle-Naive returns the exact shortest path passing on the points of 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 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 into a hash table. In *RC-Oracle-Naive*, we use algorithm *FastFly* for the point cloud shortest path query, which has 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. 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 path. 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 points of the point cloud according to Theorem 4.1, and the oracle itself regarding *RC-Oracle-Naive* also computes the pairwise P2P exact shortest paths, so *RC-Oracle-Naive* returns the exact shortest path passing on the points of the point cloud. □