Efficient Path Oracles for Proximity Queries on Point Clouds

ANONYMOUS and ANONYMOUS

The prevalence of computer graphics technology boosts the developments of point clouds in recent years, which offer advantages over terrain surfaces (represented by *Triangular Irregular Networks*, i.e., *TIN*s) in proximity queries, including the *shortest path query*, the *k-Nearest Neighbor* (*kNN*) *query*, and the *range query*. Since (1) all existing on-the-fly and oracle-based shortest path query algorithms on a *TIN* are very expensive, (2) all existing on-the-fly shortest path query algorithms on a point cloud are still not efficient, and (3) there are no oracle-based shortest path query algorithms on a point cloud, we propose (1) an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the shortest path query among a set of *Points-Of-Interests* (*POIs*) on the point cloud, and (2) a different efficient $(1+\epsilon)$-approximate shortest path oracle that directly answers the shortest path query between any point and a POI both have a good performance (in terms of the oracle construction time, oracle size and shortest path query time). We also propose (1) two adaptions of the first oracle that answer the shortest path query between any point on the point cloud and a POI, (2) two different adaptions of both two oracles that answer the shortest path query for any points if no POIs are given, and (3) two efficient algorithms that answer the $(1+\epsilon)$-approximate *kNN* and range queries using these oracle. Our experimental results show that our two oracles are up to (1) 975 times, 30 times, 6 times, and (2) 42,000 times, 10,800 times and 27 times better than the best-known oracle on a *TIN* in terms of the oracle construction time, oracle size and shortest path query time. Our two algorithms for both *kNN* and range queries are up to 100 times and 100 times faster than the best-known algorithms.

CCS Concepts: • **Information systems → Proximity search**.

Additional Key Words and Phrases: proximity queries; spatial database; point clouds

ACM Reference Format:
Anonymous and Anonymous. 2024. Efficient Path Oracles for Proximity Queries on Point Clouds. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (April 2024), 112 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Conducting proximity queries, including (1) the *shortest path query*, i.e., given a source s and a destination t, which answers the shortest path between s and t, (2) the *k-Nearest Neighbor* (*kNN*) *query* [58], i.e., given a query object q and a user parameter k, which answers all the shortest paths from q to the k nearest objects of q, and (3) the *range query* [50], i.e., given a query object q and a range value r, which answers all the shortest paths from q to the objects whose distance to q are at most r, on a 3D surface is a topic of widespread interest in both industry and academia [29, 66]. The shortest path query is the most fundamental type of the proximity query. In industry, numerous companies and applications, such as Google Earth [2] and Cyberpunk 2077 [4], utilize the shortest path passing on a 3D surface (such as Earth) for route planning. In academia, the shortest path query on a 3D model is a prevalent research topic in the field of databases [23, 35, 36, 45, 62, 63, 67, 68]. There are different representations of a 3D surface, including a terrain surface represented by a *Triangular Irregular Network* (*TIN*) and a point cloud. While performing the shortest path query

Authors' address: Anonymous; Anonymous.

ACM 0362-5915/2024/4-ART1
https://doi.org/XXXXXXX.XXXXXXX

on a *TIN* has been extensively studied, answering the shortest path query on a point cloud is an emerging topic. For example, Tesla uses the shortest path passing on point clouds of the driving environment for autonomous driving [15, 21, 44, 49], and Metaverse uses the shortest path passing on point clouds of objects such as mountains for efficient navigation in Virtual Reality [42, 43]. Applications of the other two proximity queries include rover path planning [17] and military tactical analysis [39].

Point cloud and *TIN*: (1) A point cloud is represented by a set of 3D *points* in space. Figure 1 (a) shows a satellite map of Mount Rainier [54] (a national park in the USA) in an area of 20km × 20km, and Figure 1 (b) shows the point cloud with 63 points of Mount Rainier. Given a point cloud, we create a *conceptual graph* of the point cloud, such that its *vertices* consist of the points in the point cloud, and its *edges* consist of a set of edges between each vertex and its 8 neighbor vertices in the 2D plane (this graph is stored in the memory and used for the shortest path query). Figure 1 (c) shows a conceptual graph of a point cloud. (2) A *TIN* contains a set of *faces* each of which is denoted by a triangle. Each face consists of three line segments called *edges* connected with each other at three *vertices*. The gray surface in Figure 1 (d) is a *TIN* of Mount Rainier, which consists of vertices, edges, and faces. We focus on three paths: (1) the path passing on (a conceptual graph of) a point cloud in Figures 1 (b) and (c), (2) the *surface path* [36] passing on (the faces of) a *TIN* in Figure 1 (d), and (3) the *network path* [36] passing on (the edges of a) a *TIN* in Figure 1 (e).

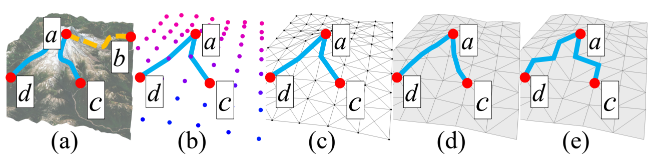

Fig. 1. (a) A satellite map, (b) paths passing on a point cloud, (c) a conceptual graph of a point cloud, (d) surface and (e) network paths passing on a *TIN*

1.1 Motivation

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

(i) *More direct access to point cloud data.* We can use an iPhone 12/13 Pro LiDAR scanner to scan an object and generate a point cloud in 10s [61], 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 [51]. But, in order to obtain a *TIN* of an object, typically, researchers need to transform a point cloud to a *TIN* [34]. Our experimental results show that it needs 210s ≈ 3.5 min to transform a point cloud with 25M points to a *TIN*.

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

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

(iv) *Small distance error of the shortest path passing on a point cloud.* In Figures 1 (b) and (d), the shortest path passing on a point cloud is similar to the shortest surface path passing on a *TIN* (since for the former path, each point can connect with 8 neighbor points). But, in Figures 1 (d) and (e), the shortest surface path and the shortest network path passing on a *TIN* are very different (since for the latter path, each vertex can only connected with only 6 neighbor vertices). Our experimental results show that the distance of the shortest path passing on a point cloud (resp. the shortest network path passing on a *TIN*) is 1.002 (resp. 1.1) times larger than that of the shortest surface path passing on a *TIN*.

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

1.1.2 ***P2P, A2P and A2A queries****.* (1) Given a set of *Points-Of-Interests* (*POIs*) on a point cloud or a *TIN*, conducting (i) the shortest path query between pairs of POIs, or (ii) the *kNN* and range queries such that the query object and other objects are all POIs, on the point cloud, i.e., *POIs-to-POIs* (*P2P*) *query*, is important. For example, we can select POIs as reference points when measuring similarities between two different 3D objects [38, 59], and we can select POIs as residential locations when studying migration patterns of the wildness animals [26, 46]. (2) If POIs are not given as the source or the query object, but POIs are given as the destination or the other objects, we need to conduct (i) the shortest path query between any point and a POI, or (ii) the *kNN* and range queries such that the query object can be any point but other objects are POIs, on the point cloud, i.e., *Any points-to-POIs* (*A2P*) *query*. (3) If POIs are not given as input, we need to conduct (i) the shortest path query between pairs of any points, or (ii) the *kNN* and range queries such that the query object and other objects are any points, on the point cloud, i.e., *Any points-to-Any points* (*A2A*) *query*. Note that the A2A query is more general than the A2P and P2P query, and the A2P query is more general than the P2P query, since POIs need to be pre-selected. By substituting the point cloud to a *TIN*, and any points on the point cloud to arbitrary points on the *TIN*, we obtain similar queries, i.e., *P2P*, *ARbitrary points-to-POIs* (*AR2P*), and *ARbitrary points-to-ARbitrary points* (*AR2AR*) *query* on a *TIN*. The AR2P (resp. AR2AR) query on a *TIN* is more general than the A2P (resp. A2A) query on a point cloud since a point may lie on the face of a *TIN*.

1.1.3 ***Usage of oracles****.* Although answering the proximity query on a point cloud *on-the-fly* is fast, if we can pre-compute the 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, where 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*, the time taken to return the shortest path is called the *shortest path query time*, and the time taken to return the *kNN* or range queries result is called the *kNN or range query time*.

1.1.4 ***Snowfall evacuation example****.* We conducted a case study on an evacuation simulation in Mount Rainier due to snowfall [55] to show the usefulness of performing proximity queries (in terms of the P2P, A2P and A2A queries) on point clouds using oracles. For the P2P query, in Figure 1 (a), we need to find the shortest paths (in blue and yellow lines) from one of the viewing platforms (e.g., POI a) on the mountain to its k-nearest hotels (e.g., POIs b to d) due to the limited capacity of each hotel. In Figures 1 (b) - (e), c and d are the k-nearest hotels to a where $k = 2$. Our case study shows that to successfully evacuate all the visitors, the calculation of the shortest paths is expected to be finished within 12 min. Our experimental results show that we can construct an oracle for the P2P query on a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels) in 80s ≈ 1.3 min, but it needs 77,200s ≈ 21.4 hours on a *TIN* (constructed based on the same point cloud) to construct the same oracle. In addition, we can return the shortest paths from each viewing platform to its k-nearest hotels in 6s with the oracle, but it needs 4,400s ≈ 1.2 hours on a point cloud without the oracle. Thus, constructing an oracle for the P2P query on point clouds is necessary since 1.3 min + 6s ≤ 12 min, but 21.4 hours ≥ 12 min and 1.2 hours ≥ 12 min. For the A2P query, if a visitor who can be at any position is climbing the mountain (i.e., we do not know the position of the visitor before constructing the oracle), we need to find the shortest paths from this visitor to his/her k-nearest hotels. For the A2A query, if a visitor who can be at any position is climbing the mountain, and the positions of the hotels are also not given (i.e., no hotels are available, and there are only temporary resettlement sites available, such that we do not know the position of these temporary resettlement sites before constructing the oracle), we need to find the shortest paths from this visitor to his/her k-temporary resettlement sites. Although the A2A query generalizes the A2P query, if the hotels are given as input, there is no need to use build an oracle for the A2A query to answer the A2P query. We can construct an oracle for the A2P query on a point cloud with 5M points and 500 POIs (250 viewing platforms and 250 hotels) in 250s ≈ 4.1 min and return the shortest paths from each viewing platform to its k-nearest hotels in 11s, but it needs 21,000 ≈ 5.8 hours to construct an oracle for the A2A query on the same point cloud and needs 6s to return the shortest paths. Thus, it is necessary to design an oracle for the A2P query since 4.1 min + 11s ≤ 12 min, but 5.8 hours + 6s ≥ 12 min.

1.1.5 ***Solar storm example.*** We conducted another case study on the evacuation of Mars rovers (used in NASA's Mars exploration project with cost USD 2.5 billion [47]) due to the frequent solar storms [11] to show the usefulness of the oracle for the A2P query compared with the P2P and A2A queries, where the Mars surface is represented in a point cloud. In the case of solar storms, Mars rovers need to find the shortest escape paths quickly from their current positions (which can be any position) on Mars to shelters or working stations (which are POIs) to avoid damage. The memory size of NASA's Mars 2020 rover is 256MB [10]. Our experimental results show that constructing an oracle for the A2P query on a point cloud with 250k points and 500 POIs (shelters or working stations) needs 25s and 28MB ≤ 256MB, but it needs 2,100 ≈ 35 min and 10GB ≥ 256MB to construct an oracle for the A2A query on the same point cloud. Thus, it is necessary to design an oracle for the A2P query. The oracle for the P2P query is not applicable in this example.

1.2 Challenges

1.2.1 ***Inefficiency of on-the-fly algorithms.*** All existing algorithms [52, 60, 70] for conducting proximity queries on a point cloud *on-the-fly* are very slow, since they (1) first construct a *TIN* using the given point cloud in $O(N)$ time, where N is the number of points in the point cloud, and (2) then calculate the shortest path passing on this *TIN*. For calculating the shortest surface path passing on a *TIN*, the best-known on-the-fly *exact* [64] and *approximate* [35] algorithm run in $O(N^2 \log N)$ and $O((N + N') \log(N + N'))$ time, respectively, where N' is the number of additional points introduced for bound guarantee. For calculating the shortest network path passing on a *TIN*, the best-known on-the-fly *approximate* algorithm [36] runs in $O(N \log N)$ time. Our experimental results show (1) algorithm [64] needs 290,000s ≈ 3.4 days, (2) algorithm [35] needs 161,000s ≈ 1.9 days, and (3) algorithm [36] needs 15,000s ≈ 4.2 hours to perform the *kNN* query for all 500 objects on a *TIN* (constructed by the given point cloud) with 0.5M vertices.

1.2.2 ***Non-existence of oracles.*** No existing oracle can answer proximity queries on a point cloud. The best-known oracle [62, 63] for the P2P query and the best-known oracle [33] for both of the AR2P and AR2AR queries only pre-compute shortest surface paths passing on a *TIN*. Although we can first construct a *TIN* using the point cloud, then use [33, 62, 63] for point cloud oracle construction, their oracle construction time is very large due to the *bad criterion for algorithm earlier termination*. This is because although they use the *Single-Source All-Destination* (*SSAD*) algorithm [18, 35, 36, 64], i.e., a Dijkstra-based algorithm [27], to pre-compute the shortest surface path passing on the *TIN* from each POI (or point) to other POIs (or points), and provide a criterion to *terminate it earlier*, its criterion is very loose, and different POIs (or points) have the *same* earlier termination criterion. In our experiment, even after the *SSAD* algorithm has visited most of the POIs (or points), their earlier termination criterion are still not reached. After constructing a *TIN* using the given point cloud, the oracle construction time is $O(nN^2 + c_1 n)$ for the oracle [62, 63], and is $O(c_2 N^2)$ for the oracle [33], respectively, where n is the number of POIs on the point cloud and c_1, c_2 are constants depending on the point cloud ($c_1 \in [35, 80]$ on a point cloud with 2.5M points, $c_2 \in [75, 154]$ on a point cloud with 100k points). In our experiment, the oracle construction time for the oracle [62, 63] is 78,000s $\approx$ 21.7 hours on a point cloud with 2.5M points and 500 POIs, and for the oracle [33] is 50,000s $\approx$ 13.9 hours on a point cloud with 100k points.

1.3 Our First Oracle

We first propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the P2P shortest path query on a point cloud called *Rapid Construction path Oracle*, i.e., *RC-Oracle*, which has a good performance in terms of the oracle construction time, oracle size and shortest path query time compared with the best-known oracle [62, 63] for the P2P query on a point cloud due to the concise information about the pairwise shortest paths between any pair of POIs stored in the oracle, where $\epsilon > 0$ is the *error parameter*. We adapt *RC-Oracle* to be *RC-Oracle-A2P-Small Construction time* (*RC-Oracle-A2P-SmCon*), *RC-Oracle-A2P-Small Query time* (*RC-Oracle-A2P-SmQue*) and *RC-Oracle-A2A*, that are three efficient $(1+\epsilon)$-approximate shortest path oracles that answer the A2P and A2A shortest path queries on a point cloud. All of them also achieve good performances compared with the best-known oracle [33] for the A2P and A2A queries on a point cloud. Based on the four oracles, we develop an efficient $(1+\epsilon)$-approximate proximity query algorithm that answers the *kNN* and range queries (in terms of the P2P, A2P and A2A queries) on a point cloud. We introduce the key idea of the small oracle construction time of *RC-Oracle*, and the key idea of the efficient adaption to *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*.

(1) ***RC-Oracle***: It has a small oracle construction time due to two reasons. (i) *Rapid point cloud on-the-fly shortest path query algorithm*: When constructing *RC-Oracle*, we propose algorithm *Fast on-the-Fly shortest path query*, i.e., *FastFly*, which is a Dijkstra-based algorithm [27] returning its calculated shortest path passing on a point cloud. It can significantly reduce the algorithm's running time, since computing the shortest path passing on a *TIN* is expensive. (ii) *Rapid oracle construction*: When constructing *RC-Oracle*, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, to calculate the shortest path passing on the point cloud from for each POI to other POIs *simultaneously*, and set *different* earlier termination criterion for different POIs, i.e., this criterion is tight.

(2) ***RC-Oracle-A2P-SmCon***: After the construction of *RC-Oracle*, when a source or a query object s that is not a POI, we use algorithm *FastFly* to answer the proximity queries starting from s to other POIs. But, by using the shortest paths passing on a point cloud between any pair of POIs stored in *RC-Oracle*, we can terminate algorithm *FastFly earlier* and avoid visiting all the POIs for time-saving. In this way, we can easily adapt *RC-Oracle* to be *RC-Oracle-A2P-SmCon*. The oracle construction time of *RC-Oracle-A2P-SmCon* is the same as that of *RC-Oracle* and is small.

(3) ***RC-Oracle-A2P-SmQue***: Since *RC-Oracle* uses algorithm *FastFly* to calculate the shortest path passing on the point cloud from for each POI to other POIs, by regarding the destination POIs

as all points on the point cloud, and keep the source POIs the same, we can easily adapt *RC-Oracle* to be *RC-Oracle-A2P-SmQue*. The shortest path query time of *RC-Oracle-A2P-SmQue* is the same as that of *RC-Oracle* and is small.

(4) ***RC-Oracle-A2A***: Since *RC-Oracle* stores the pairwise P2P approximate shortest path passing on a point cloud, by creating POIs that have the same coordinate values as all points on the point cloud, we can easily adapt *RC-Oracle* to be *RC-Oracle-A2A*.

1.4 Our Second Oracle

We then propose a total different but also efficient $(1+\epsilon)$-approximate shortest path oracle that directly answers the A2P shortest path query on a point cloud called *TIght result path Oracle*, i.e., *TI-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 [33] for the A2P query on a point cloud due to the tight result about the shortest paths between only some points stored in the oracle. We adapt *TI-Oracle* to be *TI-Oracle-A2A*, which is an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the A2A shortest path query on a point cloud. It also achieves good performances compared with the best-known oracle [33] for the A2A query on a point cloud. Based on them, we develop another efficient $(1+\epsilon)$-approximate proximity query algorithm that answers the *kNN* and range queries (in terms of the A2P and A2A queries) on a point cloud. We introduce the key idea of the small oracle construction time and small oracle size of *TI-Oracle*, and the key idea of efficient adaption to *TI-Oracle-A2A*.

(1) ***TI-Oracle***: It has a small oracle construction time and small oracle size due to the *tight shortest paths result*. When constructing *TI-Oracle*, although the source or the query object s can be any point on the point cloud, we only calculate and store the shortest paths passing on the point cloud *among some points close to the given POIs*, and the shortest paths *between each POI and some points on the point cloud close to it*, instead of *among all the points on the point cloud* (e.g., the case in *RC-Oracle-A2A*), for time-saving and space-saving. When answering the shortest path results using *TI-Oracle*, we first calculate the shortest paths passing on the point cloud between s and some points close to s on-the-fly, and then use these paths together with the stored paths to get the final result.

(2) ***TI-Oracle-A2A***: Since no POI is given, we first randomly select some points as POIs, such that we can first construct *TI-Oracle*. After the construction of *TI-Oracle*, when the destination or other objects t are also any points on the point cloud, we also calculate the shortest paths passing on the point cloud between t and some points close to t on-the-fly. In this way, we can easily adapt *TI-Oracle* to be *TI-Oracle-A2A*.

1.5 Contributions and Organization

1.5.1 ***Contributions of this journal paper***. We summarize the major contributions of this journal paper as follows.

(1) We propose (i) algorithm *FastFly* for answering the shortest path query on-the-fly on a point, (ii) six oracles *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A*, *TI-Oracle* and *TI-Oracle-A2A* that efficiently answer the P2P, A2P and A2A shortest path queries on a point cloud, and (iii) two different efficient proximity query algorithms that answer the *kNN* and range queries using the first four and last two oracles.

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

(3) Our six oracles all perform much better than the best-known oracle [62, 63] for the P2P query, the best-known oracle [33] for the A2P and A2A queries on a point cloud in terms of the oracle construction time, oracle size and shortest path query time. The *kNN* and range queries time with the assistance of these oracles also perform much better than the best-known oracles [33, 62, 63]. Our experimental results show that (i) for the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time and oracle size are 80s $\approx$ 1.3 min and 50MB for *RC-Oracle*, and are 78,000s $\approx$ 21.7 hours and 1.5GB for the best-known oracle [62, 63], the *kNN* and range queries time of all 500 POIs are both 12.5s for *RC-Oracle*, are both 150s for the best-known oracle [62, 63], and are both 161,000s $\approx$ 1.9 days for the best-known on-the-fly approximate shortest surface path query algorithm [35] on the *TIN* (constructed by the given cloud), (ii) for the A2P query on a point cloud with 250k points and 500 POIs, the oracle construction time, oracle size and *kNN* query time are 25s, 28MB and 11s for *TI-Oracle*, and are 1,050,000s $\approx$ 12 days, 300GB and 300s for the best-known oracle [33]. *RC-Oracle* and *TI-Oracle* also support real-time responses, i.e., they can construct the oracle in 0.4s and 1.25s, and then answer the *kNN* query in 7ms and 14ms on a point cloud with 10k points and 250 POIs.

1.5.2 ***Contributions comparison with previous conference paper*.** This paper is an extension of the previous conference paper [69]. The conference version [69] only has (1) algorithm *FastFly* for answering the shortest path query on-the-fly on a point, (2) two oracles *RC-Oracle* and *RC-Oracle-A2A* for answering the P2P and A2A shortest path queries on a point cloud, and (3) an efficient proximity algorithm that answers the *kNN* and range queries using these two oracles. Compared with study [69], this paper extends *P2P and A2A queries* to *P2P, A2P and A2A queries*. Although the A2A query is more general than the A2P query, such that *RC-Oracle-A2A* can be also used for the A2P query, it is unnecessary to construct a more complicated oracle *RC-Oracle-A2A* for answering a simpler A2P query. Two examples in Sections 1.1.4 and 1.1.5 motivate us to build oracles for the A2P query. We summarize the new contribution of this journal paper compared with the previous conference paper [69].

(1) We propose (i) four new oracles *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A* that efficiently answer the A2P and A2A shortest path queries on a point cloud, and (ii) a proximity query algorithm that answers the *kNN* and range queries (in terms of the A2P and A2A queries) on a point cloud using *TI-Oracle* and *TI-Oracle-A2A*. We design three oracles (i) *RC-Oracle-A2P-SmCon*, (ii) *RC-Oracle-A2P-SmQue* and (iii) *TI-Oracle* for the A2P query, each of them performs better in the case of (i) fewer proximity queries, (ii) more proximity queries and the POIs are close to each other (e.g., the density of POIs is high), and (iii) more proximity queries and the POIs are far away from each other (e.g., the density of POIs is low).

(2) We provide additional theoretical analysis on (i) the oracle construction time, oracle size, shortest path query time and error bound of four new oracles, and (ii) the *kNN* query time, range query time and error bound for proximity query algorithm of using *TI-Oracle* and *TI-Oracle-A2A*.

(3) Although the new techniques in this paper are an extension of the previous conference paper [69], these new techniques are all up-to-date, and they are state-of-the-art in the A2P and A2A queries compared with the techniques in paper [69]. Specifically, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* all perform better than *RC-Oracle-A2A*, i.e., the oracle with the best performance proposed in the previous conference paper [69], for the A2P query in terms of the oracle construction time and oracle size. Our experimental results show that for the A2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time, oracle size and *kNN* query time are 250s $\approx$ 4.1 min, 280MB and 11s for *TI-Oracle*, and are 21,000 $\approx$ 5.8 hours, 100GB and 6s for *RC-Oracle-A2A*. The oracle construction time of *RC-Oracle-A2P-SmCon* and *RC-Oracle-A2P-SmQue* are also 160 times and 79 times better than that of *RC-Oracle-A2A*.

1.5.3 ***Organization***. The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 covers the related work. Section 4 present *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*. Section 5 present *TI-Oracle* and *TI-Oracle-A2A*. Section 6 covers the empirical studies and Section 7 concludes the paper.

2 PROBLEM DEFINITION

2.1 Notations and Definitions

2.1.1 ***Point cloud and TIN***. Given a set of points, we let C be a point cloud of these points, and N be the number of points in 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 on C. We let $L_x = x_{max} - x_{min}$ (resp. $L_y = y_{max} - y_{min}$) be the side length of C along x-axis (resp. y-axis), and $L = \max(L_x, L_y)$. Figure 2 (a) shows a point cloud C with $L_x = L_y = 4$. In this paper, the point cloud C that we considered is a grid-based point cloud [14, 28], because a grid-based 3D object, e.g., a grid-based point cloud [14, 28] and a grid-based *TIN* [24, 45, 58, 62, 63], 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. In Figure 2 (a), given a green point q, $N(q)$ is denoted as 7 blue points and 1 red point s. We can easily extend our problem to the non-grid-based point cloud. Given a point p in a non-grid-based point cloud, we just change $N(p)$ to be a set of neighbor points of p such that the Euclidean distance between p and all points on this non-grid-based point cloud is smaller than a user-defined parameter. Let P be a set of POIs each of which is a point on the point cloud and n be the size of P. Since a POI can only be a point on C, $n \leq N$, i.e., POIs are a subset of points in a point cloud. In the P2P and A2P queries, there is no need to consider the case when a new POI is added or removed. In the case when a POI is added, we can create an oracle to answer the A2A query, which implies we have considered all possible POIs to be added. In the case when a POI is removed, we can still use the original oracle. Let T be a *TIN* triangulated [53] by the points in C. Figure 2 (b) shows an example of a *TIN* T. In this figure, given a green vertex q, the neighbor vertices of q are 6 blue vertices.

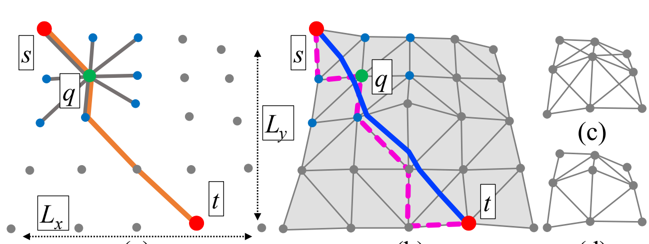

Fig. 2. (a) A point cloud with orange $\Pi^*(s,t|C)$, (b) a *TIN* with blue $\Pi^*(s,t|T)$ and pink $\Pi_N(s,t|T)$, (c) a conceptual graph of a point cloud, and (d) a conceptual graph of a *TIN*

2.1.2 ***Conceptual graph***. We define G to be a conceptual graph of C. Let $G.V$ and $G.E$ be the set of vertices and edges of G. Each point in C is denoted by a vertex in $G.V$. For each point $q \in C$, $G.E$ consists of a set of edges between q and $q' \in N(q)$. Figure 2 (c) shows a conceptual graph of a point cloud. Given a pair of points p and p' in 3D space, we define $d_E(p, p')$ to be the Euclidean distance between p and p'. Given a pair of points s and t on C, let $\Pi^*(s,t|C) = (s = q_1, q_2, \ldots, q_l = t)$, with $l \geq 2$, be the exact shortest path passing on (G of) C between s and t, such that (i) each q_i is a vertex in $G.V$, (ii) each (q_i, q_{i+1}) is an edge in $G.E$, and (iii) $\sum_{i=1}^{l-1} d_E(q_i, q_{i+1})$ is the minimum. Given a pair of points s and t on C, let $\Pi_A(s, t|C)$ be the shortest path returned by oracle A, where $A \in \{$*RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A*, *TI-Oracle*, *TI-Oracle-A2A*$\}$. In the P2P query, the shortest path passing on C from a source (POI) to a destination (POI) can contain different sub-paths where a sub-path starts from a point on C to another point on C, i.e., the differences between the points and POIs are that (1) we use points (from C) to construct G, and then calculate the shortest path passing on G, but (2) we use POIs as sources and destinations to calculate the shortest path. In the A2P query, the source is not necessary to be a POI. In the A2A query, both the source and destination are not necessary to be POIs. G is stored as a data structure in the memory for internal processing and C can be cleared from the memory, so we do not need to construct G every time when we need to calculate the shortest path passing on C. Our experimental results show that it just needs 0.01s to construct G of C with 2.5M points. Figure 2 (a) shows an example of $\Pi^*(s, t|C)$ in orange line. We define $|\cdot|$ to be the distance of a path (e.g., $|\Pi^*(s, t|C)|$ is the distance of $\Pi^*(s, t|C)$). *RC-Oracle* guarantees that $|\Pi_{RC\text{-}Oracle}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any s and t in P, oracle A_1 guarantee that $|\Pi_{A_1}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any s on C and any t in P where $A_1 \in \{$*RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle*$\}$, oracle A_2 guarantee that $|\Pi_{A_2}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any s and t on C where $A_2 \in \{$*RC-Oracle-A2A*, *TI-Oracle-A2A*$\}$.

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

2.2 Problem

The problem is to (1) design efficient $(1 + \epsilon)$-approximate shortest path oracles 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 thess oracles for efficiently answering the $(1 + \epsilon)$-approximate *kNN* and range queries.

3 RELATED WORK

3.1 On-the-fly Algorithms

All existing *on-the-fly* proximity query algorithms [52, 60, 70] on a point cloud are very slow. Given a point cloud, they first triangulate it into a *TIN* [53] in $O(N)$ time, then they calculate the shortest path passing on this *TIN*. To the best of our knowledge, no algorithm can answer proximity queries on a point cloud directly without converting it to a *TIN*. There are two types of *TIN* shortest path query algorithms, i.e., (1) the *shortest surface path* [18, 35, 41, 48, 64, 65] and (2) the *shortest network path* [36] query algorithms.

3.1.1 ***Shortest surface path query algorithms.*** There are two more sub-types. (1) *Exact algorithms*: Algorithm [48], algorithm [65] and algorithm [18] use continuous Dijkstra, checking window and terrain face unfolding algorithm to calculate the result in $O(N^2 \log N)$, $O(N^2 \log N)$ and $O(N^2)$ time, and the best-known exact shortest surface path query algorithm *DirectIon-Oriented*, i.e., algorithm *DIO* [64] unfolds the 3D *TIN* into a 2D *TIN*, and then connects the source and destination using a line segment on this 2D *TIN* with a visibility tree structure to calculate the result in $O(N^2 \log N)$ time. Although its time complexity is slightly larger than that of algorithm [18], the experimental results of study [64] show algorithm *DIO* runs more than 100 times faster than algorithm [18]. But, algorithm *DIO* (without constructing a *TIN* first) cannot be directly adapted on the point cloud, because there is no face to be unfolded in a point cloud. (2) *Approximate algorithms*: All algorithms [35, 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 result. The best-known $(1+\epsilon)$-approximate shortest surface path query algorithm, i.e., algorithm *Kaul* [35] (as recognized by [62, 63]) runs in $O(\gamma N \log(\gamma N))$, where $\gamma = \frac{l_{max}}{\epsilon l_{min}\sqrt{1-\cos\theta}}$, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of the *TIN*, and θ is the minimum inner angle of any face in the *TIN*. If we let the path pass on the conceptual graph of the point cloud, algorithm *Kaul* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

3.1.2 ***Shortest network path query algorithm***. Since the shortest network path does not cross the faces of a *TIN*, it is an approximate path. The best-known approximate shortest network path query algorithm *Dijkstra*, i.e., algorithm *Dijk* [36] runs in $O(N \log N)$ time. If we let the path pass on the conceptual graph of the point cloud, algorithm *Dijk* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

Drawbacks of the on-the-fly algorithms: Although we can pre-process the point cloud and store the generated *TIN* as a data structure in the memory, all these algorithms are still time-consuming. Since the time for calculating the shortest path passing on a *TIN* is much larger than (i.e., 10^2 to 10^5 times larger than) the time for converting a point cloud to a *TIN*. Thus, the latter time can be neglected. We denote algorithm (1) *DIO-Adapt*, (2) *Kaul-Adapt*, and (3) *Dijk-Adapt*, to be the adapted algorithm [52, 60, 70], which first constructs a *TIN* using the given point cloud (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory), and then use algorithm (1) *DIO* [64], (2) *Kaul* [35], and (3) *Dijk* [36] to compute the corresponding shortest path passing on the *TIN*. Since we regard the shortest path passing on a point cloud as the exact shortest path, algorithm *DIO-Adapt*, *Kaul-Adapt*, and *Dijk-Adapt* return the approximate shortest path passing on a point cloud. Our experimental results show algorithm *DIO-Adapt*, *Kaul-Adapt*, and *Dijk-Adapt* first needs to convert a point cloud with 0.5M points to a *TIN* in 4.2s, then perform the *kNN* query for all 2500 objects on this *TIN* in 290,000s ≈ 3.2 days, 90,000s ≈ 1 day, and 15,000s ≈ 4.2 hours, respectively.

3.2 Oracles for the shortest path query

Only the previous conference paper [69] (i.e., the preliminary version of this journal paper) answers the P2P and A2A shortest path queries on a point cloud. The comparison of this journal paper and the previous conference paper [69] has been presented in Section 1.5.2. Apart from the previous conference paper [69], no existing oracle can answer the shortest path query on a point cloud. But, study [62, 63] (resp. study [33]) can answer the P2P (resp. AR2AR) by using an oracle to index shortest surface paths passing on a *TIN*. They use algorithm [18] for calculating the shortest surface paths passing on a *TIN* during oracle construction. We denote them by *Space Efficient Oracle* (*SE-Oracle*) [62, 63] and *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [33] such that they use algorithm *DIO* for calculating the shortest surface paths passing on a *TIN*. In addition, we denote *SE-Oracle-Adapt* to be the adapted oracle of *SE-Oracle* [62, 63] that first constructs a *TIN* from a point cloud (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory), then uses *SE-Oracle* on this *TIN*. Similarly, we denote *EAR-Oracle-Adapt* as the adapted oracle of *EAR-Oracle* [33]. By performing a linear scan using the shortest path query results, they can answer other proximity queries.

3.2.1 ***SE-Oracle-Adapt***. It uses a *compressed partition tree* [62, 63] and *well-separated node pair sets* [16] to index the $(1+\epsilon)$-approximate pairwise P2P shortest surface paths passing on a *TIN* (constructed by the given point cloud). Its oracle construction time, oracle size and shortest path query time are $O(nN^2 + n\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [30, 37] ($\beta \in [1.5, 2]$ according to [62, 63]). It is regarded as the best-known oracle for the P2P query on a point cloud.

Drawbacks of *SE-Oracle-Adapt*: Its oracle construction time is large due to the *bad criterion for algorithm earlier termination*. For POIs in the same level of the compressed partition tree, they have the *same* earlier termination criteria. But, in *RC-Oracle*, we have *different* earlier termination criteria for each different POI, to minimize the running time of the *SSAD* algorithm. In the P2P query on a point cloud, our experimental results show that for a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt* is 78,000s ≈ 21.7 hours, while *RC-Oracle* just needs 80s ≈ 1.3 min.

3.2.2 ***EAR-Oracle-Adapt***. It also uses well-separated node pair sets, which is similar to *SE-Oracle-Adapt*. But, *EAR-Oracle-Adapt* adapts *SE-Oracle-Adapt* from the P2P query on a point cloud to the A2P and A2A queries on a point cloud by using Steiner points on the faces of the *TIN* (constructed by the given point cloud) and *highway nodes* as POIs in well-separated node pair sets construction. Its oracle construction time, oracle size and shortest path query time are $O(\lambda\xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m\log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face. It is regarded as the best-known oracle for the A2P and A2A queries on a point cloud.

Drawbacks of *EAR-Oracle-Adapt*: It also has the *bad criterion for algorithm earlier termination* drawback. But, in *TI-Oracle*, we also have *different* earlier termination criteria for each different point, which is similar to *RC-Oracle*. In the A2P query on a point cloud, our experimental results show that for a point cloud with 250k points and 500 POIs, the oracle construction time of *EAR-Oracle-Adapt* is 1,050,000s ≈ 12 days, while *TI-Oracle* just needs *TI-Oracle* is 25s.

3.3 Oracles for other proximity queries

Except for the preliminary conference version [69] of this journal paper, no existing oracle can answer proximity queries on a point cloud. But, studies [24, 25, 58] build an oracle to answer proximity queries on a *TIN*. Specifically, studies [24, 25] use a multi-resolution terrain model (resp. *SUrface Oracle* (*SU-Oracle*) [58] uses a surface index) to answer the AR2P *kNN* query on a *TIN* in $O(N^2)$ (resp. $O(N\log^2 N)$) time. We adapt *SU-Oracle* to be *SU-Oracle-Adapt* for the A2P query on a point cloud in a similar way of *SE-Oracle-Adapt*. Although *SU-Oracle-Adapt* is regarded as the best-known oracle to directly answer the *kNN* query, studies [62, 63] show the *kNN* query time of *SU-Oracle-Adapt* is up to 5 times larger than that of using *SE-Oracle-Adapt* with a linear scan of the shortest path query result. This is because *SU-Oracle-Adapt* only indexes the first nearest POI of the given query point. It still needs to use on-the-fly algorithm to find other k-nearest POIs ($k > 1$), such the results are not stored in the oracle. In addition, study [66] builds an oracle to answer the dynamic version of the *kNN* query, and study [67] builds an oracle to answer the reverse nearest neighbor query, but they are not our main focus.

3.4 Comparisons

We compare *RC-Oracle*, algorithm *FastFly* and other algorithms that support the shortest path query on a point cloud in Table 1. Recall that when constructing *RC-Oracle*, we have different earlier termination criteria for different POIs when using algorithm *FastFly*. We denote the naive version of our oracle as *RC-Oracle-Naive* if no earlier termination criterion is used. From the table, *RC-Oracle* performs better than the best-known oracle *SE-Oracle-Adapt* [62, 63] for the P2P query, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle-A2P* (resp. *RC-Oracle-A2A* and *TI-Oracle-A2A*) perform better than the best-known oracle *EAR-Oracle-Adapt* [33] for the A2P (resp. A2A) query, and algorithm *FastFly* is the best on-the-fly algorithm.

Table 1. Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error	Query type
Oracle-based algorithm								
SE-Oracle-Adapt [62, 63]	$O(nN^2 + n\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P
EAR-Oracle-Adapt [33]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m\log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium	Small	P2P, A2P, A2A
RC-Oracle-Naive	$O(nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
RC-Oracle	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(1)$	**Small**	**Small**	**P2P**
RC-Oracle-A2P-SmCon	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(N\log N)$	**Medium**	**Small**	**P2P, A2P**
RC-Oracle-A2P-SmQue	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P**
RC-Oracle-A2A	$O(\frac{N\log N}{\epsilon})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P, A2A**
TI-Oracle	$O(\frac{N\log N}{\epsilon} + Nn + n\log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P**
TI-Oracle-A2A	$O(\frac{N\log N}{\epsilon} + N\sqrt{N} + \sqrt{N}\log\sqrt{N})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P, A2A**
On-the-fly algorithm								
DIO-Adapt [64]	-	N/A	-	N/A	$O(N^2\log N)$	Large	Small	P2P, A2P, A2A
Kaul-Adapt [35]	-	N/A	-	N/A	$O(\gamma N\log(\gamma N))$	Large	Small	P2P, A2P, A2A
Dijk-Adapt [36]	-	N/A	-	N/A	$O(N\log N)$	Medium	Medium	P2P, A2P, A2A
FastFly	-	**N/A**	-	**N/A**	$O(N\log N)$	**Medium**	**No error**	**P2P, A2P, A2A**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [62, 63], λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, $\gamma = \frac{l_{max}}{\epsilon l_{min}\sqrt{1-\cos\theta}}$, 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.

4 RC-ORACLE AND ITS ADAPTIONS

4.1 Overview of *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*

We first use an example to illustrate *RC-Oracle*. In Figure 3 (a), we have a point cloud and a set of POIs a, b, c, d, e. In Figures 3 (b) - (e), we construct *RC-Oracle* by calculating shortest paths among these POIs. In Figure 3 (f), we answer the shortest path query between two POIs using *RC-Oracle*. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A* have similar process. Next, we introduce the two components and two phases of these otacles.

4.1.1 ***Components of RC-Oracle, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue and RC-Oracle-A2A.*** There are two components, i.e., the *path map table* and the *endpoint map table*.

(1) **The path map table** M_{path} is a *hash table* [20] that stores a set of key-value pairs. For each key-value pair, it stores a pair of endpoints u and v, as a key $\langle u, v\rangle$, and the corresponding exact

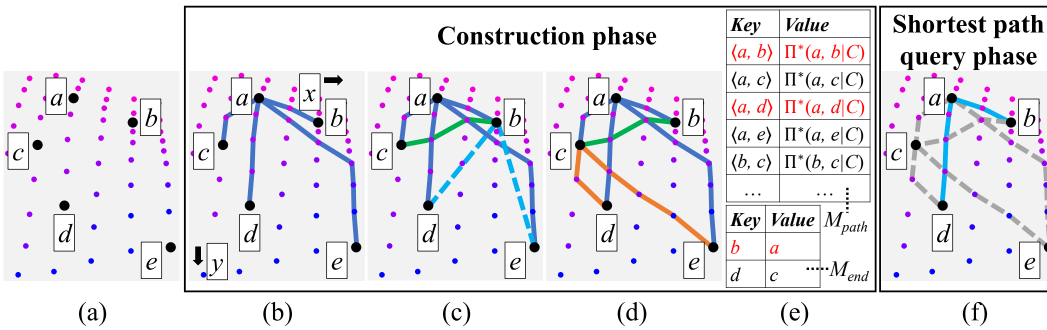

Fig. 3. *RC-Oracle* framework overview

shortest path $\Pi^*(u, v|C)$ passing on C, as a value, where the endpoint can be (i) a POI in P used in *RC-Oracle*, or as the destination used in *RC-Oracle-A2P-SmCon* and *RC-Oracle-A2P-SmQue*, or (ii) any point on C as the source used in *RC-Oracle-A2P-SmCon* and *RC-Oracle-A2P-SmQue*, or used in *RC-Oracle-A2A*. M_{path} needs linear space in terms of the number of paths to be stored. Given a pair of endpoints u and v, M_{path} can return the associated exact shortest path $\Pi^*(u, v|C)$ passing on C in $O(1)$ time. We use *RC-Oracle* as an example. In Figure 3 (d), there are 7 exact shortest paths passing on C, and they are stored in M_{path} in Figure 3 (e). For the exact shortest paths passing on C between b and c, M_{path} stores $\langle b, c\rangle$ as a key and $\Pi^*(b, c|C)$ as a value.

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

4.1.2 ***Phases of RC-Oracle, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue and RC-Oracle-A2A***. There are two phases, i.e., *construction phase* and *shortest path query phase*. (1) For *RC-Oracle* (see Figure 3): (i) In the construction phase, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{end}. (ii) In the shortest path query phase, given a pair of POIs in P, M_{path} and M_{end}, we answer the path results between this pair of POIs efficiently. (2) For *RC-Oracle-A2P-SmCon*: (i) In the construction phase, given a point cloud C and a set of POIs P, the procedure is the same as of *RC-Oracle*. (ii) In the shortest path query phase (see Figure 4), given any point (e.g., f) on C and a POI in P, M_{path} and M_{end}, we efficiently compute the exact shortest paths passing on C between this point and some selected POIs, store the calculated paths in M_{path}, store this point and its corresponding selected POIs in M_{end}, and return the path results between this point and this POI. (3) For *RC-Oracle-A2P-SmQue*: (i) In the construction phase, given a point cloud C and a set of POIs P, the procedure is similar to *RC-Oracle*, the only difference is that the destinations are not POIs in P, but all points on C. (ii) In the shortest path query phase, given any point on C and a POI in P, we answer the path results between this point and POI efficiently. (4) For *RC-Oracle-A2A*: (i) In the construction phase, given a point cloud C, the procedure is similar to *RC-Oracle*, the only difference is that no POI is given as input, we need to create POIs that have the same coordinate values as all points on C. (ii) In the shortest path query phase, given any pair of points on C, we answer the path results between this pair of points efficiently.

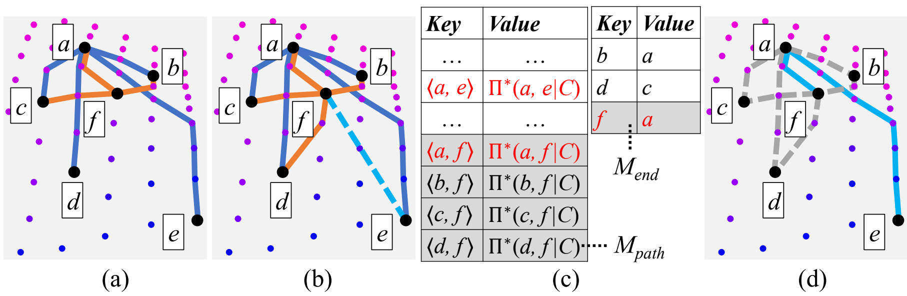

Fig. 4. *RC-Oracle-A2P-SmCon* shortest path query phase

4.2 Key Idea of *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A* and proximity query algorithms

4.2.1 ***Key Idea of RC-Oracle.*** We introduce the key idea of the small oracle construction time, small oracle size and small shortest path query time of *RC-Oracle* as follows.

(1) **Small oracle construction time**: We give the reason why *RC-Oracle* has a small oracle construction time.

(i) *Rapid point cloud on-the-fly shortest path querying by algorithm FastFly*: When constructing *RC-Oracle*, given a point cloud C and a pair of POIs s and t on C, we use algorithm *FastFly* (a Dijkstra's algorithm [27]) to directly calculate the *exact* shortest path passing on the conceptual graph of C between s and t. Figure 5 (a) shows the shortest path passing on a point cloud calculated by algorithm *FastFly*, and Figure 5 (b) (resp. Figure 5 (c)) shows the shortest surface (resp. network) path passing on a *TIN* calculated by algorithm *DIO-Adapt* (resp. *Dijk-Adapt*) of Mount Rainier in an area of 20km × 20km. The path in Figures 5 (a) and (b) are similar, but calculating the former path is much faster than the latter path, since the query region of the former path is smaller than the latter path. The path in Figure 5 (c) has a larger error than the path in Figure 5 (a). Thus, we use algorithm *FastFly* as the on-the-fly algorithm for constructing *RC-Oracle*.

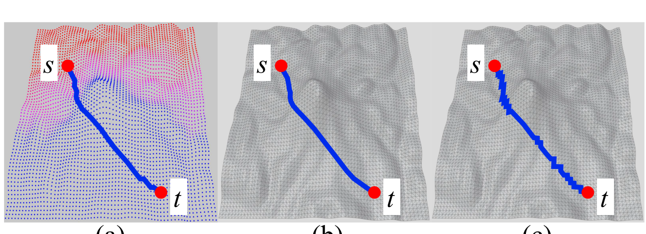

Fig. 5. (a) The shortest path passing on a point cloud, the shortest (b) surface and (c) network path passing on a *TIN*

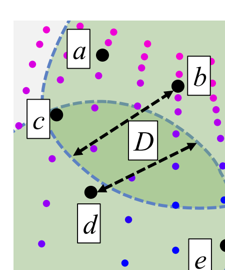

Fig. 6. *SE-Oracle-Adapt*

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

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

However, in *SE-Oracle-Adapt*, it has the *bad criterion for algorithm earlier termination* drawback. After the construction of the compressed partition tree, it pre-computes the shortest surface paths passing on T using the *SSAD* algorithm (i.e., *DIO-Adapt*) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of the *SSAD* algorithm and sets termination distance $D = \frac{8r}{\epsilon} + 10r$, where r is the radius of the source POI in the compressed partition tree. Given two POIs a and b in the same level of the tree, their termination distances are the same (suppose that the value is d_1). However, for a, it is enough to terminate the *SSAD* algorithm when the searching distance from a is larger than d_2, where $d_2 < d_1$. This results in a large oracle construction time. In Figure 6, when processing d, suppose that b and d are in the same level of the tree, and they use the *same* termination criteria to get the *same* termination distance D. Since $|\Pi^*(d, e|C)| < D$, for d, it cannot terminate the *SSAD* algorithm earlier until e is visited. The two versions of the *SSAD* algorithm are similar, we achieve a small oracle construction time mainly by using *different* termination criteria for different POIs, unlike using the *same* termination criteria for different POIs in *SE-Oracle-Adapt*.

(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*, i.e., we do not store the paths between any pairs 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.

(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 passing on C (1) between a and d, and (2) between b and d. (1) For a and d, since $\langle a, d\rangle \in M_{path}.key$, we can directly return $\Pi^*(a, d|C)$. (2) For b and d, since $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{end}, we use the key b with a smaller y-coordinate

value to retrieve the value a in M_{end}, 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 $\Pi^*(b, d|C)$.

4.2.2 ***Key Idea of RC-Oracle-A2P-SmCon***.

We introduce the key idea of the efficient adaption from *RC-Oracle* to *RC-Oracle-A2P-SmCon*, such that in the A2P query, the oracle construction time of *RC-Oracle-A2P-SmCon* remains the same with *RC-Oracle*, and the shortest path query time of *RC-Oracle-A2P-SmCon* is smaller than algorithm *FastFly*, i.e., the *SSAD* algorithm. The adaption is achieved by using the *SSAD* algorithm with the assistance of *RC-Oracle*, such that the *SSAD* algorithm can *terminate earlier*. The reason why we can terminate the *SSAD* algorithm earlier is similar to the *rapid oracle construction* reason for *RC-Oracle*, i.e., given a source that is not a POI, by considering more geometry information of the point cloud, including the Euclidean distance and the distance of the previously calculated shortest paths stored in *RC-Oracle*, we can minimize the number of destination POIs used in the *SSAD* algorithm.

Since *RC-Oracle-A2P-SmCon* has the same construction phase as *RC-Oracle* in Figures 3 (b) - (e), we only illustrate the shortest path query phase of *RC-Oracle-A2P-SmCon* with an example. There is no need to consider the case that the source and destination are both POI, in this case, we just use *RC-Oracle* for the shortest path query. So we only consider the case that one of two query points is not given as a POI. In Figure 4 (a), for point f that is not given as a POI when constructing *RC-Oracle-A2P-SmCon*, we first use the *SSAD* algorithm with f as a source, and visit all POIs with the y-coordinate smaller than or equal to f (i.e., a, b, c). Note that in this figure, it seems that the y-coordinate of c is larger than f in the 3D point cloud. But indeed, their y-coordinates are the same. At the same time, before the termination of the *SSAD* algorithm, if we can also visit the POIs with the y-coordinate larger than f, we also calculate the shortest paths passing on C between f and these POIs. In Figure 4 (b), we need to find a POI such that we have used this POI as a source in the *SSAD* algorithm and cover all other non-processed POIs as destinations (i.e., this POI is not key in M_{end}), and exact distance on C between f and this POI is the smallest. This POI is a. If f is close to the POI a, i.e., judged using the previously calculated $|\Pi^*(a, f|C)|$, and f is far away from e, i.e., judged using the Euclidean distance $d_E(e, f)$, we can use $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$ to approximate $\Pi^*(f, e|C)$. Thus, we just need to continue the previous *SSAD* algorithm with f as a source, and terminate earlier when it has visited d. We store the paths from the *SSAD* algorithm in M_{path}, and store f as key and a as value in M_{end}. Note that the *SSAD* algorithm (i.e., *FastFly*) is a Dijkstra's algorithm, so given a source, after we terminate it, we can save the result of the Dijkstra's algorithm from the memory to the hard disk. If we continue the *SSAD* algorithm with the same source, we can retrieve the previously saved result from the hard disk to the memory, and there is no need to start from scratch for time-saving. In Figure 4 (c), we have the updated M_{path} and M_{end}. In Figure 4 (d), we need to query the shortest path passing on C between e and f. Similar to the shortest path query phase of *RC-Oracle*, since $\langle e, f\rangle \notin M_{path}.key$, f is key in M_{end}, we retrieve the value a using the key f, in M_{end}, then in M_{path}, we use $\langle e, a\rangle$ and $\langle a, f\rangle$ to retrieve $\Pi^*(e, a|C)$ and $\Pi^*(a, f|C)$, for approximating $\Pi^*(e, f|C)$.

However, the shortest path query time of simply using algorithm *FastFly* with f as a source without pruning any other destination POIs is large. Our experimental result shows that for a point cloud with 2.5M points and 500 POIs, the *kNN* query time is 320s $\approx$ 5.3 min for *RC-Oracle-A2P-SmCon*, but is 1600s $\approx$ 27 min for algorithm *FastFly*.

4.2.3 ***Key Idea of RC-Oracle-A2P-SmQue***.

We introduce the key idea of the efficient adaption from *RC-Oracle* to *RC-Oracle-A2P-SmQue*, such that in the A2P query, the oracle construction time of *RC-Oracle-A2P-SmQue* will not increase a lot, and the shortest path query time of *RC-Oracle-A2P-SmQue* remains the same with *RC-Oracle*. We still regard each POI as a source and use algorithm *FastFly* for n times, the only difference between *RC-Oracle* is that the destinations are not POIs in P, but all points on C, then we can adapt *RC-Oracle* to *RC-Oracle-A2P-SmQue*. We just need to pre-compute the exact shortest paths passing on the point cloud between *some* selected pairs of POIs and points on the point cloud (not *all* pairs of POIs and points on the point cloud), so *RC-Oracle-A2P-SmQue* also has a small oracle construction time, small oracle size and small shortest path query time.

4.2.4 ***Key Idea of RC-Oracle-A2A***. We introduce the key idea of the efficient adaption from *RC-Oracle* to *RC-Oracle-A2A*. We just need to create POIs that have the same coordinate values as all points on the point cloud, then we can adapt *RC-Oracle* to *RC-Oracle-A2A*. We just need to pre-compute the exact shortest paths passing on the point cloud between *some* selected pairs of points on the point cloud (not *all* pairs of points on the point cloud), so *RC-Oracle-A2A* also has a small oracle construction time, small oracle size and small shortest path query time.

4.2.5 ***Key Idea of Proximity Query Algorithms using RC-Oracle and its adaptions***. We introduce the key idea of proximity query algorithms using these oracles. Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k, and a range value r, we can answer other proximity queries, i.e., the *kNN* and range queries using the three oracles. In the P2P query, these objects are POIs in P. In the A2P query, these objects are a point on C (regarded as the query object) and also POIs in P (regarded as the other objects). In the A2A query, these objects are POIs any points on C. A naive algorithm is to perform a linear scan using the shortest path query results. We propose an efficient algorithm for it. Intuitively, when constructing the three oracles, we have used the *SSAD* algorithm to calculate the shortest paths passing on C with q as a source and sorted these paths in ascending order based on their distance in M_{path} (we can use an additional table to store these sorted paths). For these paths, we do not need to perform linear scans over all of them in proximity queries for time-saving.

4.3 Implementation Details of *RC-Oracle*

4.3.1 ***Construction Phase***. We give the construction phase of *RC-Oracle*.

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

Detail and example: Algorithm 1 shows the construction phase of *RC-Oracle* in detail, and the following illustrates it with an example.

(1) *POIs sort* (lines 2-3): In Figure 3 (b), since $L_x < L_y$, the sorted POIs are a, b, c, d, e.

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

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

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

- *Approximation loop start* (lines 11-20): In Figure 3 (c), we first select a's closest POI in P'_{remain}, i.e., b, since $d_E(a, b) \leq \epsilon L$, it means a and b are not far away, we start the approximation loop, delete b from P_{remain} and P'_{remain}, so $P_{remain} = P'_{remain} = \{c, d, e\}$. There are three steps:

Algorithm 1 *RC-Oracle-Construction* (C, P)

Input: a point cloud C and a set of POIs P
Output: a path map table M_{path} and an endpoint map table M_{end}
1: $P_{remain} \leftarrow P, M_{path} \leftarrow \emptyset, M_{end} \leftarrow \emptyset$
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: $P_{remain} \leftarrow P_{remain} - \{u\}, P'_{remain} \leftarrow P_{remain}$
7: calculate the exact shortest paths passing on C from u to each POI in P'_{remain} simultaneously using algorithm *FastFly*
8: **for** each POI $v \in P'_{remain}$ **do**
9: $key \leftarrow \langle u, v\rangle$, $value \leftarrow \Pi^*(u, v|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
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: $P_{remain} \leftarrow P_{remain} - \{v\}, P'_{remain} \leftarrow P'_{remain} - \{v\}, P_{dest}(v) \leftarrow \emptyset$
13: **for** each POI $w \in P'_{remain}$ **do**
14: **if** $d_E(v, w) > \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ and $v \notin M_{end}.key$ **then**
15: $key \leftarrow v$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
16: **else if** $d_E(v, w) \leq \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ **then**
17: $P_{dest}(v) \leftarrow P_{dest}(v) \cup \{w\}$
18: calculate the exact shortest paths passing on C from v to each POI in $P_{dest}(v)$ simultaneously using algorithm *FastFly*
19: **for** each POI $w \in P_{dest}(v)$ **do**
20: $key \leftarrow \langle v, w\rangle$, $value \leftarrow \Pi^*(v, w|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
21: **return** M_{path} and M_{end}

  - *Far away POIs selection* (lines 13-15): In Figure 3 (c), $d_E(b, d) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$ $d_E(b, e) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, $d \notin M_{end}.key$ and $e \notin M_{end}.key$, it means d and e are far away from b, we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ that we have already calculated before to approximate $\Pi^*(b, d|C)$, and use $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$ that we have already calculated before to approximate $\Pi^*(b, e|C)$, so we get $\Pi_{RC\text{-}Oracle}(b, d|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi_{RC\text{-}Oracle}(b, e|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{end}.
  - *Close POIs selection* (line 13 and lines 16-17): In Figure 3 (c), $d_E(b, c) \leq \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, it means c is close to b, so we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(b, c|C)$, then we store c into $P_{dest}(b)$.
  - *Selected exact shortest paths calculation* (lines 18-20): In Figure 3 (c), when we have processed all POIs in P'_{remain} with b as a source, we have $P_{dest}(b) = \{c\}$, we use algorithm *FastFly* to calculate the exact shortest path passing on C between b and c, i.e., $\Pi^*(b, c|C)$ (in green line), and store $\langle b, c\rangle$ as a key and $\Pi^*(b, c|C)$ as a value in M_{path}. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to b, and we do not need to visit d and e.
- *Approximation loop end* (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 end the approximation loop and terminate the iteration.

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

4.3.2 ***Shortest Path Query Phase.*** We give the shortest path query phase of *RC-Oracle*. Given a pair of POIs s and t in P, there are two cases (s and t are interchangeable, i.e., $\langle s,t\rangle = \langle t,s\rangle$):

(1) *Exact shortest path retrieval*: If $\langle s,t\rangle \in M_{path}.key$, we retrieve $\Pi^*(s,t|C)$ as $\Pi_{RC\text{-}Oracle}(s,t|C)$ using $\langle s,t\rangle$ in $O(1)$ time (in Figures 3 (d) and (e), given a and d, since $\langle a,d\rangle \in M_{path}.key$, we retrieve $\Pi^*(a,d|C)$).

(2) *Approximate shortest path retrieval*: If $\langle s,t\rangle \notin M_{path}.key$, it means $\Pi^*(s,t|C)$ is approximated by two exact shortest paths passing on C in M_{path}, and (i) either s or t is a key in M_{end}, or (ii) both s and t are keys in M_{end}. Without loss of generality, suppose that (i) s exists in M_{end} if either s or t is a key in M_{end}, or (ii) the x- (resp. y-) coordinate of s is smaller than t when $L_x \geq L_y$ (resp. $L_x < L_y$) if both s and t are keys in M_{end}. For both of two cases, we retrieve the value s' using the key s from M_{end} 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 $\Pi^*(s',t|C)$ as $\Pi_{RC\text{-}Oracle}(s,t|C)$ to approximate $\Pi^*(s,t|C)$ ((i) in Figures 3 (d) and (e), given b and e, since $\langle b,e\rangle \notin M_{path}.key$, b is a key in M_{end}, so we retrieve the value a using the key b in M_{end}, 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 (ii) in Figure 3 (d), (e) and (f), given b and d, since $\langle b,d\rangle \notin M_{path}.key$, b and d are both keys in M_{end}, and $L_x < L_y$, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}, 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 Implementation Details of *RC-Oracle-A2P-SmCon*

We only give the shortest path query phase of *RC-Oracle-A2P-SmCon*, since it has the same construction phase as *RC-Oracle*. Suppose that we need to answer the shortest path query between a source s that can be any point on C and a destination t that is a POI in P.

Notation: Given a source q, we re-use the notation $P_{dest}(q)$ as of in the construction phase of *RC-Oracle*. In Figure 4 (a), $P_{dest}(f) = \{a,b,c\}$ since we need to use *FastFly* to calculate the exact shortest path from f to a,b,c.

Detail and example: Algorithm 2 shows the shortest path query phase of *RC-Oracle-A2P-SmCon* in detail, and the following illustrates it with an example.

(1) *New shortest paths calculation* (lines 1-18): In Figure 4 (a), given f as a source that is not a POI, and e as a destination that is a POI, there are five steps. If both source and destination are POIs, we can skip these five steps.

(i) *Smaller x- or y-coordinate POIs exact shortest paths calculation* (lines 3-6): In Figure 4 (a), since $L_x < L_y$, we have $P_{dest}(f) = \{a,b,c\}$. We calculate the exact shortest paths passing on C from f to a, b, c (in orange lines) using algorithm *FastFly*.

(ii) *Approximate POI selection and destination POIs update* (lines 7-10): In Figure 4 (a), we have a as the POI that the exact distance on C between f and a is the smallest and $a \notin M_{end}.key = \{b,d\}$, and $P' = \{d,e\}$. During the execution of algorithm *FastFly*, if we can also visit the POIs with the y-coordinate larger than f, we also calculate the shortest paths passing on C between f and these POIs, and update $P_{dest}(f)$ to cover those POIs. In this figure, there are no such POIs and we do not need to $P_{dest}(f)$. So $P_{dest}(f) = \{a,b,c\}$ and $P' = \{d,e\}$.

(iii) *Far away POIs selection* (lines 11-13): In Figure 4 (b), $d_E(e,f) > \frac{2}{\epsilon} \cdot |\Pi^*(a,f|C)|$ and $f \notin M_{end}.key$, it means e is far away from f, we can use $\Pi^*(f,a|C)$ and $\Pi^*(a,e|C)$ that we have already calculated before to approximate $\Pi^*(f,e|C)$, so we get $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(f,e|C)$ by concatenating $\Pi^*(f,a|C)$ and $\Pi^*(a,e|C)$, we store f as key and a as value in M_{end}.

Algorithm 2 *RC-Oracle-A2P-SmCon-Query* $(C, P, s, t, M_{path}, M_{end})$

Input: a point cloud C, a set of POIs P, a source s that can be any point on C, a destination t that is a POI in P, the path map table M_{path} and the endpoint map table M_{end}

Output: the updated path map table M_{path}, the updated endpoint map table M_{end} and the shortest path $\Pi_{\textit{RC-Oracle-A2P-SmCon}}(s, t|C)$ between s and t passing on C

1: **if** $\langle s, t\rangle \notin M_{path}.key$ and $s \notin M_{end}.key$ and $t \notin M_{end}.key$ **then**
2: $\quad P_{dest}(s) \leftarrow \emptyset$
3: $\quad$ **if** $L_x \geq L_y$ (resp. $L_x < L_y$) **then**
4: $\quad\quad$ **for** each POI $u \in P$ such that the x-coordinate (resp. y-coordinate) of u is smaller than s **do**
5: $\quad\quad\quad P_{dest}(s) \leftarrow P_{dest}(s) \cup \{u\}$
6: $\quad$ calculate the exact shortest paths passing on C from s to each POI in $P_{dest}(s)$ simultaneously using algorithm *FastFly* and store the result of the algorithm from the memory to the hard disk
7: $\quad u \leftarrow$ the POI in $P_{dest}(s)$ such that the exact distance on C between s and u, i.e., $|\Pi^*(s, u|C)|$, is the smallest and $u \notin M_{end}.key$, $P' \leftarrow P \setminus P_{dest}(s)$
8: $\quad$ **for** each POI $v \in P'$ such that v is also visited during the execution of algorithm *FastFly* **do**
9: $\quad\quad P_{dest}(s) \leftarrow P_{dest}(s) \cup \{v\}$
10: $\quad P' \leftarrow P \setminus P_{dest}(s)$
11: $\quad$ **for** each POI $v \in P'$ **do**
12: $\quad\quad$ **if** $d_E(s, v) > \frac{2}{\epsilon} \cdot |\Pi^*(u, s|C)|$ and $s \notin M_{end}.key$ **then**
13: $\quad\quad\quad key \leftarrow s$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
14: $\quad\quad$ **else if** $d_E(s, v) \leq \frac{2}{\epsilon} \cdot |\Pi^*(u, s|C)|$ **then**
15: $\quad\quad\quad P_{dest}(s) \leftarrow P_{dest}(s) \cup \{v\}$
16: $\quad$ continue the previous algorithm *FastFly* with s as a source by retrieving the previously saved result from the hard disk to the memory, to calculate the exact shortest paths passing on C from s to each POI in $P_{dest}(s)$ simultaneously
17: $\quad$ **for** each POI $v \in P_{dest}(s)$ **do**
18: $\quad\quad key \leftarrow \langle s, v\rangle$, $value \leftarrow \Pi^*(s, v|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
19: use the same shortest path query phase of *RC-Oracle* to retrieve $\Pi_{\textit{RC-Oracle-A2P-SmCon}}(s, t|C)$, the only difference is to substitute the *RC-Oracle* with *RC-Oracle-A2P-SmCon* and $\Pi_{\textit{RC-Oracle}}(s, t|C)$ with $\Pi_{\textit{RC-Oracle-A2P-SmCon}}(s, t|C)$ in the shortest path query phase of *RC-Oracle*
20: **return** M_{path}, M_{end} and $\Pi_{\textit{RC-Oracle-A2P-SmCon}}(s, t|C)$

(iv) *Close POIs selection* (line 11 and lines 14-15): In Figure 4 (b), $d_E(d, f) \leq \frac{2}{\epsilon} \cdot |\Pi^*(a, f|C)|$, it means d is close to f, so we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(f, d|C)$, then we store d into $P_{dest}(f)$.

(v) *Selected exact shortest paths calculation* (lines 16-18): In Figure 4 (b), when we have processed all POIs in P' with f as a source, we have $P_{dest}(f) = \{a, b, c, d\}$, we continue the previous algorithm *FastFly* with f as a source to calculate the exact shortest path passing on C from f to each POI in $P_{dest}(f)$ (in orange lines), and store each endpoints pair as a key and the corresponding path as a value in M_{path}. Since we terminate the previous algorithm *FastFly* when it visited a, b, c, there is no need to start from scratch for time-saving. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to f, and we do not need to visit e.

(2) *Shortest path query* (line 19): In Figure 4 (d), given f as a source that is not a POI, and e as a destination that is a POI, we use the same shortest path query phase of *RC-Oracle* to query $\Pi_{\textit{RC-Oracle-A2P-SmCon}}(f, e|C)$, i.e., since $\langle f, e\rangle \notin M_{path}.key$, f is a key in M_{end}, so we retrieve the value a using the key f in M_{end}, then in M_{path}, we use $\langle f, a\rangle$ and $\langle a, e\rangle$ to retrieve $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$, for approximating $\Pi^*(f, e|C)$.

4.5 Implementation Details of Proximity Query Algorithms

Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k, and a range value r, we can answer other proximity queries, i.e., the *kNN* and range queries using the three oracles. Since the proximity query algorithms for *RC-Oracle* and its adaptions are similar, we use *RC-Oracle* as an example. For *RC-Oracle-A2P-SmCon*, we first use Algorithm 2 lines 1-18 to calculate new shortest paths for q if needed, then perform similarly for *RC-Oracle* using the following details.

Detail and example: There are two cases. For both two cases, we can then return the corresponding *kNN* and range queries results.

(1) *Approximation needed in direct result return*: If $q \in M_{end}.key$, it means we need to use two paths in M_{path} to approximate some other paths in a later stage, we retrieve the value q' using the key q from M_{end} (in Figures 3 (d) and (e), $b \in M_{end}.key$, we retrieve the value a using the key b from M_{end}), there are two more cases:

(i) *Linear scan*: For objects with a smaller x- (resp. y-) coordinate compared with q' when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and these objects (in Figures 3 (d) and (e), since $L_x < L_y$, there is no POI with a smaller y-coordinate compared with a).

(ii) *Direct result return*: For objects (not including q) with a larger x- (resp. y-) coordinate compared with q' when $L_x \geq L_y$ (resp. $L_x < L_y$) (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with a are $\{c, d, e\}$), there are further more two cases:

- *Direct result return without approximation*: If the endpoint pairs of q and these objects are keys in M_{path}, it means that we have used the *SSAD* algorithm with q as a source for such objects and we have already sorted such paths in order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $\langle b, c\rangle \in M_{path}.key$, we know that $|\Pi^*(b, c|C)|$ is sorted in order, but since there is only one distance, it does not matter whether itself is sorted in order or not).
- *Direct result return with approximation*: If the endpoint pairs of q and these objects are not keys in M_{path}, it means that we have used the *SSAD* algorithm with q' as a source for such objects and we have already sort such paths in order, we just need to use the exact distance between q' and these objects plus $|\Pi^*(q', q|C)|$, to get the approximate distance between q and o in sorted order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $\langle b, d\rangle \notin M_{path}.key$ and $\langle b, e\rangle \notin M_{path}.key$, we know that $|\Pi^*(a, d|C)|$ and $|\Pi^*(a, e|C)|$ are sorted in order, so $|\Pi(b, d|C)|$ and $|\Pi(b, e|C)|$ are also sorted in order).

(2) *Approximation not needed in direct result return*: If $q \notin M_{end}.key$, it means we do need to use two paths in M_{path} to approximate all other paths in a later stage (in Figures 3 (d) and (e), $c \notin M_{end}.key$), there are two more cases:

(i) *Linear scan*: For objects with a smaller x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and these objects (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a smaller y-coordinate compared with c are $\{a, b\}$, we perform a linear scan on the shortest path query result between c and $\{a, b\}$).

(ii) *Direct result return*: For objects with a larger x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we have used the *SSAD* algorithm with q as a source for such objects and we have already sorted such paths in order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with c are $\{d, e\}$, we know that $|\Pi^*(c, d|C)|$ and $|\Pi^*(c, e|C)|$ are sorted in order).

4.6 Theoretical Analysis

4.6.1 ***Algorithm FastFly, RC-Oracle and its adaptions.*** The analysis of algorithm *FastFly* is in Theorem 4.1, and the analysis of *RC-Oracle* and its adaptions are in Theorem 4.2.

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

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

Theorem 4.2. *The oracle construction time, oracle size and shortest path query time of (1) RC-Oracle are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$, $O(1)$, (2) RC-Oracle-A2P-SmCon are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$, $O(N \log N)$, (3) RC-Oracle-A2P-SmQue are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{N}{\epsilon})$, $O(1)$, and (4) RC-Oracle-A2A are $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$, $O(1)$, respectively. RC-Oracle always has $|\Pi_{RC\text{-}Oracle}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any pairs of POIs s and t in P, RC-Oracle-A2P-SmCon and RC-Oracle-A2P-SmQue always have $|\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ and $|\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmQue}(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ for any s on C and any t in P, and RC-Oracle-A2A always has $|\Pi_{RC\text{-}Oracle\text{-}A2A}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any pairs of points s and t on C.*

Proof. We give the proof for *RC-Oracle* as follows.

Firstly, we show the *oracle construction time*. (1) In *POIs sort* step, it needs $O(n \log n)$ time. Since there are n POIs, and we use the quick sort for sorting. (2) In *shortest paths calculation* step, it needs $O(\frac{N \log N}{\epsilon} + n)$ time. (i) It needs to use $O(\frac{1}{\epsilon})$ POIs as a source to run algorithm *FastFly* for the exact shortest paths calculation according to standard packing property [32], and each algorithm *FastFly* needs $O(N \log N)$ time. (ii) For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly*, we just calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for the shortest paths approximation. (3) So the oracle construction time is $O(\frac{N \log N}{\epsilon} + n \log n)$.

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

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

We give the proof for *RC-Oracle-A2P-SmCon* as follows. We need to change (1) $O(1)$ to $O(N \log N)$ in the shortest path query time since it involves algorithm *FastFly*, and (2) any pairs of POIs in P to any s on C and any t in P in the error bound. The other analysis is the same as *RC-Oracle*.

We give the proof for *RC-Oracle-A2P-SmQue* as follows. We need to change (1) n to N in the oracle size since we regard all points on the point cloud as destinations during oracle construction, and (2) any pairs of POIs in P to any s on C and any t in P in the error bound. The other analysis is the same as *RC-Oracle*.

We give the proof for *RC-Oracle-A2A* as follows. We need to change (1) n to N in the oracle construction time and oracle size since we create POIs that have the same coordinate values as all points on the point cloud, and (2) any pairs of POIs in P to any pairs of points on C in the error bound. The other analysis is the same as *RC-Oracle*. □

4.6.2 ***The shortest path passing on a point cloud and the shortest surface or network path passing on a TIN***. We show the relationship of $|\Pi^*(s,t|C)|$ with $|\Pi_N(s,t|T)|$ and $|\Pi^*(s,t|T)|$ in Lemma 4.3.

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

Proof. (1) In Figure 2 (a), given a green point q on C, it can connect with one of its 8 neighbor points (7 blue points and 1 red point s). In Figure 2 (b), given a green vertex q on T, it can only connect with one of its 6 blue neighbor vertices. So $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$. (2) We let $\Pi_E(s,t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s,t|T)$ passes) between s and t. According to left hand side equation in Lemma 2 of [36], we have $|\Pi_E(s,t|T)| \leq k \cdot |\Pi^*(s,t|T)|$. Since $\Pi_N(s,t|T)$ considers all the edges on T, $|\Pi_N(s,t|T)| \leq |\Pi_E(s,t|T)|$. Thus, we finish the proof by combining these inequalities. □

4.6.3 ***Proximity query algorithms***. We provide analysis on the proximity query algorithms using *RC-Oracle* and its adaptions. For the *kNN* and range queries, both of them return a set of objects. Given a query object q, we let v_f (resp. v'_f) be the furthest object to q among the returned objects calculated using the exact distance on C (resp. the approximated distance on C returned by *RC-Oracle*). In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d. And d is the furthest POI to a in these two POIs, i.e., $v_f = d$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c. And b is the furthest POI to a in these two POIs, i.e., $v'_f = b$. We define the error rate of the *kNN* and range queries 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 $\frac{|\Pi^*(a,b|C)|}{|\Pi^*(a,d|C)|}$. Then, we show the query time and error rate of *kNN* and range queries using *RC-Oracle* and its adaptions in Theorem 4.4.

Theorem 4.4. *The query time and error rate of both the kNN and range queries by using RC-Oracle, RC-Oracle-A2P-SmQue and RC-Oracle-A2A are both $O(n')$ and $1+\epsilon$, respectively. The query time and error rate of both the kNN and range queries by using RC-Oracle-A2P-SmCon are $O(N\log N + n')$ and $1+\epsilon$, respectively.*

Proof Sketch. The *query time* of *RC-Oracle*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A* is due to the usages of the shortest path query phase of them for n' times in the worst case. The *query time* of *RC-Oracle-A2P-SmCon* is due to the usage of the new shortest paths calculation step in $O(N \log N)$ time of it for only once, and the usage of the shortest path query step of it for n' times in the worst case. The *error rate* is due to its definition and the error of *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*. The detailed proof appears in our technical report [13]. □

5 TI-ORACLE AND ITS ADAPTIONS

5.1 Overview of *TI-Oracle* and *TI-Oracle-A2A*

We first use an example to illustrate *TI-Oracle*. In Figure 7 (a), we have a point cloud and a set of POIs a, b, c, d, e. In Figures 7 (b) - (h), we divide the points into several regions, and calculate the shortest paths (i) between each POI and each point (including the boundary point) of the region that this POI lies in, and (ii) between each pair of intersection points by regarding all the intersection points as POIs and using *RC-Oracle* for indexing. In Figures 7 (i) - (k), for a point k that is not given when constructing *TI-Oracle*, we calculate the shortest paths between k and each point (including the boundary point) of the region corresponding to e, then answer the shortest path query between k and another POI using *TI-Oracle*. *TI-Oracle-A2A* has similar process. Next, we introduce the two concepts, six components and two phases of *TI-Oracle* and *TI-Oracle-A2A*.

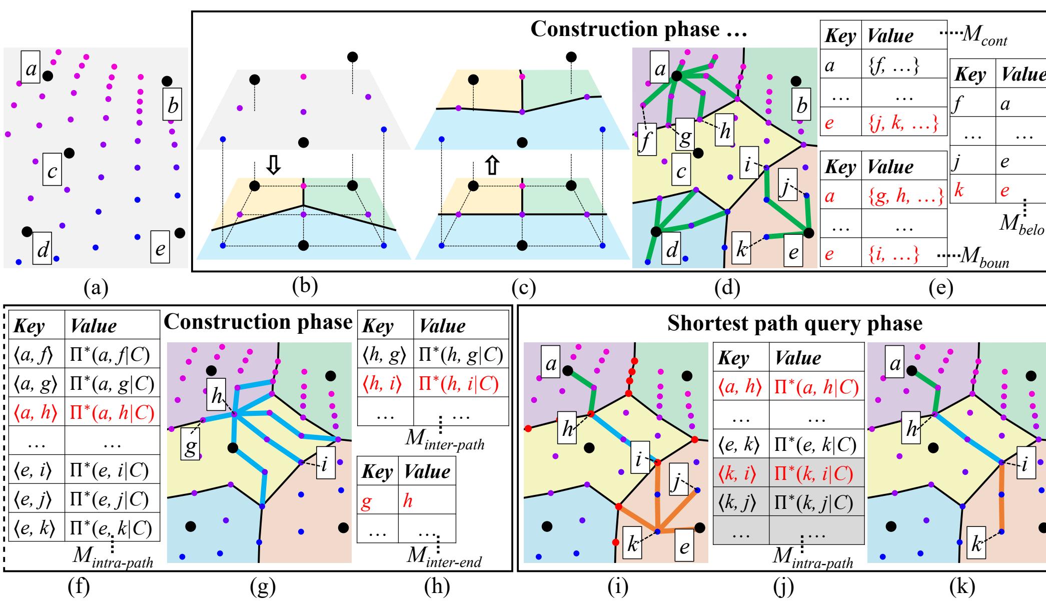

Fig. 7. *TI-Oracle* framework overview

5.1.1 ***Concepts of TI-Oracle and TI-Oracle-A2A***. There are two concepts, i.e., the *partition cell* and the *boundary point*.

(1) **The partition cell** is a region that contains a set of points on the point cloud. Before we give the details of the partition cell, we introduce the Voronoi diagram and the Voronoi cell [22] first. Given a space and a set of POIs, a Voronoi diagram partitions the space into a set of disjoint Voronoi cells using the POIs, such that for any point q lies in a Voronoi cell, the nearest POI of q is the POI corresponding to this Voronoi cell. Then, we give the details of the partition cell. Similar to the Voronoi cell, given a point cloud C and a set of POIs in P, we partition C into a set of partition cells using P (such that the boundary of each partition cell passes on edges of the conceptual graph of C), but we do not require the nearest POI condition to be satisfied in the partition cell. If a point lies inside a partition cell but does not lie on the boundary of this partition cell, we say that this point belongs to this partition cell. In Figure 7 (b), given C and P, we project C in the xy coordinate 2D plane, build a Voronoi diagram in Euclidean space using the grid-based 2D point cloud and P

with the sweep line algorithm [22] in $O(n \log n)$ time, and obtain a set of Voronoi cells. In Figure 7 (c), we obtain a set of partition cells in the xy coordinate 2D plane by correcting the boundary of each Voronoi cell (such that the boundary of each partition cell passes on edges of the conceptual graph of C), and project the partition cells in the xy coordinate 2D plane back to the 3D space to obtain the partition cells of C. In Figure 7 (d), there are five partition cells in different colors, f belongs to the partition cell corresponding to a, but g and h do not belong to the partition cell corresponding to a (since they are on the boundary of this partition cell).

(2) **The boundary point** is an intersection point between C and the boundary of partition cells. Given C and P, we can obtain a set of unique boundary points B. In Figure 7 (d), g, h, i are three boundary points.

5.1.2 *Components of TI-Oracle and TI-Oracle-A2A*.

There are six components, i.e., the *containing point map table*, the *boundary point map table*, the *belonging point map table*, the *intra-path map table*, the *inter-path map table* and the *inter-endpoint map table*.

(1) **The containing point map table** M_{cont} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores an endpoint u as a key (such that u is used for creating the partition cell), and a set of points $\{v_1, v_2, \dots\}$ as a value (such that $\{v_1, v_2, \dots\}$ are the points on C except u that belong to the partition cell corresponding to u), where the endpoint can be (i) a POI in P used in *TI-Oracle*, or (ii) any point on C used in *TI-Oracle-A2A*. The space consumption and query time of M_{cont} is similar to M_{path} in *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*. In Figure 7 (d), f is a point (resp. j, k are two points) on C that belong to the partition cell corresponding to a (resp. e), so we store a (resp. e) as a key and $\{f, \dots\}$ (resp. $\{j, k, \dots\}$) as a value in M_{cont} in Figure 7 (e).

(2) **The boundary point map table** M_{boun} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores an endpoint u as a key (such that u is used for creating the partition cell), and a set of boundary points $\{v_1, v_2, \dots\}$ as a value (such that $\{v_1, v_2, \dots\}$ are the boundary points of the partition cell corresponding to u), where the endpoint has the same meaning as in M_{cont}. The space consumption and query time of M_{boun} is similar to M_{cont}. In Figure 7 (d), g, h are two boundary points (resp. i is a boundary point) of the partition cell corresponding to a (resp. e), so we store a (resp. e) as a key and $\{g, h, \dots\}$ (resp. $\{i, \dots\}$) as a value in Figure 7 (e).

(3) **The belonging point map table** M_{belo} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores a point u on C as a key and another endpoint v as a value (such that v is used for creating the partition cell, and u belongs to the partition cell corresponding to v), where the endpoint has the same meaning as in M_{cont}. The space consumption and query time of M_{belo} is similar to M_{cont}. In Figure 7 (d), f belongs to (resp. j, k belong to) the partition cell corresponding to a (resp. e), so we store f as a key (resp. j and k as keys) and a as a value (resp. e and e as values) in M_{belo} in Figure 7 (e).

(4) **The intra-path map table** $M_{intra\text{-}path}$ is a *hash table* that stores a set of key-value pairs. Given an endpoint u and a point v on C (such that v belongs to the partition cell corresponding to u or v is the boundary point of the partition cell corresponding to u), the exact shortest path passing on C between u and v is called the *intra-path*, where the endpoint has the same meaning as in M_{cont}. For each key-value pair, $M_{intra\text{-}path}$ stores an endpoint u and a point v (such that v belongs to the partition cell corresponding to u or v is the boundary point of the partition cell corresponding to u), as a key $\langle u, v\rangle$, and the corresponding intra-path $\Pi^*(u, v|C)$, as a value. The space consumption and query time of $M_{intra\text{-}path}$ is similar to M_{cont}. In Figure 7 (d), there are 6, 4 and 3 intra-paths in green lines with a, d and e as a source, and they are stored in $M_{intra\text{-}path}$ in Figure 7 (f). For the intra-paths between a and f (resp. between a and g), $M_{intra\text{-}path}$ stores $\langle a, f\rangle$ (resp. $\langle a, g\rangle$) as a key

and $\Pi^*(a, f|C)$ (resp. $\Pi^*(a, g|C)$) as a value. Similarly, in Figure 7 (i), there are 3 intra-paths in orange lines with k as a source, where k is not a POI.

(5) **The inter-path map table** $M_{\textit{inter-path}}$ is a *hash table* that stores a set of key-value pairs. Given a pair of boundary points u and v, the shortest path passing on C between u and v is called the *inter-path*. For each key-value pair, $M_{\textit{inter-path}}$ stores a pair of boundary points u and v, as a key $\langle u, v\rangle$, and the corresponding exact inter-path $\Pi^*(u, v|C)$, as a value. By regarding all the boundary points as POIs in *RC-Oracle*, $M_{\textit{inter-path}}$ in *TI-Oracle* and *TI-Oracle-A2A* is the same as M_{path} in *RC-Oracle*. The space consumption and query time of $M_{\textit{inter-path}}$ is similar to M_{cont}. In Figure 7 (g), there are 6 exact inter-paths in light blue lines, and they are stored in $M_{\textit{inter-path}}$ in Figure 7 (h). For the exact inter-paths between h and g (resp. h and i), $M_{\textit{inter-path}}$ stores $\langle h, g\rangle$ (resp. $\langle h, i\rangle$) as a key and $\Pi^*(h, g|C)$ (resp. $\Pi^*(h, i|C)$) as a value.

(6) **The inter-endpoint map table** $M_{\textit{inter-end}}$ is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores a boundary point u as a key and another boundary point v as a value. By regarding all the boundary points as POIs in *RC-Oracle*, $M_{\textit{inter-path}}$ in *TI-Oracle* and *TI-Oracle-A2A* is the same as M_{path} in *RC-Oracle*. The space consumption and query time of $M_{\textit{inter-end}}$ is similar to M_{cont}. In Figure 7 (g), g is close to h, we concatenate $\Pi^*(g, h|C)$ and the exact shortest paths passing on C with h as a source, to approximate the shortest paths passing on C with g as a source, so we store g as a key, and h as a value in $M_{\textit{inter-end}}$ in Figure 7 (h).

5.1.3 ***Phases of TI-Oracle and TI-Oracle-A2A.*** There are two phases, i.e., *construction phase* and *shortest path query phase*. (1) For *TI-Oracle* (see Figure 7): (i) In the construction phase, given a point cloud C and a set of POIs P, we divide C into several partition cells, store the points belonging to each partition cell in M_{cont} and M_{belo}, store the points belonging to each partition cell in M_{belo}, pre-compute the exact shortest paths passing on C between each POI and each point that belongs to the partition cell generated by this POI (and also between each POI and each boundary point of the same partition cell) and store them in $M_{\textit{intra-path}}$, pre-compute the exact shortest paths passing on C between some selected pairs of boundary points (by regarding all the boundary points as POIs in *RC-Oracle*) and store them in $M_{\textit{inter-path}}$, and store the non-selected boundary points and their corresponding selected boundary points in $M_{\textit{inter-end}}$. (ii) In the shortest path query phase, given any point on C and a POI in P, M_{cont}, M_{boun}, M_{belo}, $M_{\textit{intra-path}}$, $M_{\textit{inter-path}}$ and $M_{\textit{inter-end}}$, we calculate the exact shortest paths passing on C between this point and each point that belongs to the partition cell generated by this point (and also between this point and each boundary point of the same partition cell), update $M_{\textit{intra-path}}$, and then return the path results between this point and this POI. (2) For *TI-Oracle-A2A*: (i) In the construction phase, given a point cloud C, the procedure is similar to *TI-Oracle*, the only difference is that no POI is given as input, we need to randomly select some points on the point cloud as POIs to construct the partition cells. (ii) In the shortest path query phase, given any pairs of points on C, the procedure is similar to *TI-Oracle*, the only difference is that for both of two points, we need to calculate the exact shortest paths passing on C between each point and each point that belongs to the partition cell generated by this point (and also between this point and each boundary point of the same partition cell).

5.2 Key Idea of *TI-Oracle*, *TI-Oracle-A2A* and proximity query algorithms

5.2.1 ***Key Idea of TI-Oracle.*** We introduce the key idea of the construction of partition cells of *TI-Oracle*, the key idea of the small oracle construction time and small oracle size of *TI-Oracle*, and the key idea of shortest path query of *TI-Oracle* as follows.

(1) **Construction of partition cells**: We give the process for constructing the partition cells in Section 5.1.1, but we omit the process for transferring the Voronoi cells in the xy coordinate 2D plane to the partition cells in the xy coordinate 2D plane, such that the boundary of each partition

cell passes on edges of the conceptual graph of C. We give the key idea of this step here. In Figure 8 (a), we have a part of the Voronoi diagram in the xy coordinate 2D plane with three Voronoi cells and a conceptual graph (without the diagonal edges) of C in the xy coordinate 2D plane. For each intersection point between the boundary of the 2D Voronoi cells and the 2D conceptual graph (without the diagonal edges), e.g., the blue points, we move the point to one of the two closest points on C of the edge that this intersection point lies on, i.e., following the red arrows. For each intersection point among the boundary of the 2D Voronoi cells, e.g., the green point, we move the point to one of the four closest points on C of the square that this intersection point lies on, i.e., following the orange arrow. In Figure 8 (b), we connect these intersection points, to form the boundary (in pink lines) of partition cells in the xy coordinate 2D plane.

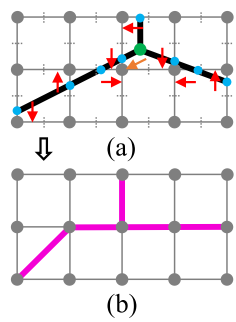

Fig. 8. Boundary correction of the 2D partition cells

(2) **Small oracle construction time**: We give the reason why *TI-Oracle* has a small oracle construction time. It is due to the *tight shortest paths result* of *TI-Oracle*. When constructing *TI-Oracle*, we only calculate the shortest paths passing on C (i) between each POI and each point that belongs to the partition cell generated by this POI (and also between each POI and each boundary point of the same partition cell), i.e., the intra-paths, and (ii) between each pair of boundary points, i.e., the inter-paths, by regarding all the boundary points as POIs and using *RC-Oracle* for indexing. When POIs are far away from each other, the oracle construction time of *TI-Oracle* is much smaller. This is because for the boundary points belong to one partition cell, they are close to each other. By regarding these boundary points as POIs and using *RC-Oracle* for indexing, we can terminate the *SSAD* algorithm earlier for most of the POIs.

We use an example for illustration. In Figure 7 (a), we have a point cloud and a set of POIs a, b, c, d, e. In Figures 7 (b) and (c), we construct partition cells. In Figures 7 (d) - (f), by using the partition cells, we store the corresponding information in M_{cont}, M_{boun} and M_{belo}. We use the *SSAD* algorithm with each POI as a source, to calculate the intra-paths in green lines, and terminate earlier when it has visited all the points that belong to the partition cell generated by this POI and the boundary points of the same partition cell. We store the paths in $M_{intra\text{-}path}$. Note that all the intra-paths do not cross through the boundary of the partition cells that they belong to, and the number of points of each partition cell is much smaller than N, so the *SSAD* algorithm can terminate very early. Our experimental result shows that we can use the *SSAD* algorithm to calculate the intra-paths with a POI as a source in $O(1)$ time. In Figures 7 (g) and (h), we regard each boundary point as POIs in *RC-Oracle*, calculate the exact inter-paths in blue lines between boundary points, and store them in $M_{inter\text{-}path}$, and the corresponding boundary points in $M_{inter\text{-}end}$. Since the number of boundary points is much smaller than N, we do not need to run the *SSAD* algorithm many times. In addition, for two boundary points that belong to one partition cell (e.g., h

and g), they are close to each other, after we use the *SSAD* algorithm with h as a source to visit other boundary points, when we need to use the *SSAD* algorithm with g as a source, we can terminate it very earlier.

However, in *RC-Oracle-A2A*, it needs to use the *SSAD* algorithm with each point on C as a source to cover all other points on C, which results in a large oracle construction time, although it can terminate the *SSAD* algorithm earlier for some points as sources. In addition, in *RC-Oracle-A2P-smallQue*, when POIs are far away from each other, it is difficult to utilize the *rapid oracle construction* advantage of *RC-Oracle* during construction, since it is difficult to use the previous calculated shortest paths to approximate other shortest paths, and the *SSAD* algorithm with each POI as a source is difficult to terminate earlier.

(3) **Small oracle size**: We give the reason why *TI-Oracle* has a small oracle size. We only store a small number of paths in *TI-Oracle*, i.e., we do not store the paths between any pairs of points on C. In Figure 7 (d), we only store the intra-paths in green lines. In Figure 7 (g), for a pair of boundary points g and i, we use exact inter-paths $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$ in light blue lines to approximate $\Pi^*(g, i|C)$, i.e., we will not store $\Pi^*(g, i|C)$ in $M_{\textit{inter-path}}$ for memory saving.

(4) **Shortest path query**: We give the process for the shortest path query phase of *TI-Oracle*. Given a source s that is not a POI, we use the *SSAD* algorithm to calculate the shortest paths passing on C between s and each point that belongs to the partition cell that s belongs to (and also between s and each boundary point of the same partition cell), i.e., the intra-paths, and store in $M_{\textit{intra-path}}$ (if the path does not exist in $M_{\textit{intra-path}}$). Similar to the construction phase, this can be finished in $O(1)$ time. Given a destination POI t, there are two cases. (i) If s belongs to the partition cell generated by t or s is a boundary point of the same partition cell, i.e., the intra-path between s and t exists in $M_{\textit{intra-path}}$, we directly return the result. (ii) If not, we find a point s' (resp. t') that is a boundary point of the partition cell that s (resp. t) belongs to, such that the distance of the intra-path between s and s' (calculated using $M_{\textit{intra-path}}$), plus the distance of the inter-path between s' and t' (calculated using the shortest path query phase of *RC-Oracle* with $M_{\textit{inter-path}}$ and $M_{\textit{inter-end}}$), plus the distance of the intra-path between t' and t (calculated using $M_{\textit{intra-path}}$), is the smallest among all other possible boundary point of the partition cell that s and t belong to, and concatenate these three paths.

We use an example for illustration. In Figures 7 (i) - (k), for a point k that is not given when constructing *TI-Oracle*, we use the *SSAD* algorithm with k as a source, to calculate the intra-paths in orange lines, and terminate earlier when it has visited all the points that belong to the partition cell generated by k and the boundary points of the same partition cell. We store the paths in $M_{\textit{intra-path}}$. Given a POI a, if we need to find the shortest path between a and k, among all the boundary points of the partition cell that a (resp. e) belongs to, we find a boundary point h (resp. i), such that $|\Pi^*(a, h|C)| + |\Pi^*(h, i|C)| + |\Pi^*(i, k|C)|$ is the smallest, where we can use $\langle a, h\rangle$ (resp. $\langle i, k\rangle$) to retrieve $\Pi^*(a, h|C)$ (resp. $\Pi^*(i, k|C)$) in $M_{\textit{intra-path}}$, and use the shortest path query phase in *RC-Oracle*, h, i, $M_{\textit{inter-path}}$ and $M_{\textit{inter-end}}$ to calculate $\Pi^*(h, i|C)$, and concatenate $\Pi^*(a, h|C)$, $\Pi^*(h, i|C)$ and $\Pi^*(i, k|C)$ to be the result path.

5.2.2 ***Key Idea of TI-Oracle-A2A***. We introduce the key idea of the efficient adaption from *TI-Oracle* to *TI-Oracle-A2A*. In the oracle construction phase, since no POI is given, we first randomly select some points (e.g., $\sqrt{N}$ points) as POIs, and then we follow the same oracle construction phase as of *TI-Oracle* to construct *TI-Oracle-A2A*. In the shortest path query phase, given any pair of points on C, we follow the same shortest path query phase as of *TI-Oracle*, the only difference is that we use the *SSAD* algorithm for both points as sources to calculate the intra-paths.

5.2.3 ***Key Idea of Proximity Query Algorithms using TI-Oracle and TI-Oracle-A2A***. We introduce the key idea of proximity query algorithms using *TI-Oracle* and *TI-Oracle-A2A*. Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k, and a range value r, we can answer other proximity queries, i.e., the *kNN* and range queries using the two oracles. The objects have the same meaning in the proximity query algorithms using *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *RC-Oracle-A2A*. Similarly, a naive algorithm is still to perform a linear scan using the shortest path query results. We also propose an efficient algorithm for it. Intuitively, when we need to answer the shortest path query phase of *TI-Oracle* and *TI-Oracle-A2A* more than once, we need to use the shortest path query phase of *RC-Oracle* to find the inter-path between the same pair of boundary points more than once. Although the shortest path query time of *RC-Oracle* (i.e., the query time for finding the inter-path between a pair of boundary points) is $O(1)$, if we do not store the exact shortest path passing on C between this pair of boundary points in $M_{inter\text{-}path}$, we also need to search in $M_{inter\text{-}end}$, and again in $M_{inter\text{-}path}$ for path appending, which increases the running time. To handle this, if the exact shortest path passing on C between a pair of boundary points is not stored in $M_{inter\text{-}path}$, after we find it using the shortest path query phase of *RC-Oracle*, we use an additional table to store it, so when we need to calculate this inter-path again later, we can directly retrieve the result in this additional table for time-saving.

5.3 Implementation Details of *TI-Oracle*

5.3.1 ***Construction Phase****.* We give the construction phase of *TI-Oracle*.

Notation: Given a source q, we re-use the notation $P_{dest}(q)$ as of in the construction phase of *RC-Oracle*.

Detail and example: Algorithm 3 shows the construction phase of *TI-Oracle* in detail, and the following illustrates it with an example.

(1) *Partition cells calculation* (line 1): In Figures 7 (b) and (c), we obtain a set of partition cells. In Figure 7 (d), there are five partition cells in different colors.

(2) M_{cont}, M_{boun} *and* M_{belo} *calculation* (lines 2-12): In Figure 7 (d), f is a point (resp. j, k are two points) on C that belong to the partition cell corresponding to a (resp. e), g, h are two boundary points (resp. i is a boundary point) of the partition cell corresponding to a (resp. e), f belongs to (resp. j, k belong to) the partition cell corresponding to a (resp. e), so we have M_{cont}, M_{boun} and M_{belo} in Figure 7 (e).

(3) *Intra-paths calculation* (lines 13-16): In Figure 7 (d), for POI a, $P_{dest}(e) = \{f, g, h, \dots\}$. For each POI a, b, c, d, e, we calculate the intra-paths in green lines using algorithm *FastFly*, and store the key-value pairs in $M_{intra\text{-}path}$ in Figure 7 (f).

(4) *Inter-paths calculation* (line 18): In Figure 7 (g), for each boundary point, we regard them as POI in *RC-Oracle* to calculate the inter-paths in light blue lines, and store the output as $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ in Figure 7 (h).

5.3.2 ***Shortest Path Query Phase****.* We give the shortest path query phase of *TI-Oracle*. Suppose that we need to answer the shortest path query between a source s that can be any point on C and a destination t that is a POI in P.

Notation: Given a pair of boundary points p and q in B, let $\Pi_{inter\text{-}path}(p, q|C)$ be the inter-path between p and q calculated using the shortest path query phase of *RC-Oracle* with $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$. In Figure 7 (f), given h and i, $\Pi_{inter\text{-}path}(h, i|C)$ is the same as $\Pi^*(h, i|C)$, and given g and i, $\Pi_{inter\text{-}path}(g, i|C)$ is approximated by $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$.

Detail and example: Algorithm 4 shows the shortest path query phase of *TI-Oracle* in detail, and the following illustrates it with an example.

(1) *Same partition cell* (lines 1-2): In Figures 7 (d) and (f), given k as a source that is not a POI, and e as a destination that is a POI, since $\langle e, k\rangle \in M_{intra\text{-}path}.key$, we directly retrieve $\Pi^*(e, k|C)$.

Algorithm 3 *TI-Oracle-Construction* (C, P)

Input: a point cloud C and a set of POIs P

Output: a set of boundary points B, a containing point map table M_{cont}, a boundary point map table M_{boun}, a belonging point map table M_{belo}, , an intra-path map table $M_{intra\text{-}path}$, an inter-path map table $M_{inter\text{-}path}$ and an inter-endpoint map table $M_{inter\text{-}end}$

1: project C in the xy coordinate 2D plane, build a Voronoi diagram in Euclidean space using the grid-based 2D point cloud and P with the sweep line algorithm to generate a set of Voronoi cells, correct the boundary of each Voronoi cell, project the partition cells in the xy coordinate 2D plane back to the 3D space to obtain the partition cells of C, $B \leftarrow$ the boundary points, $M_{cont} \leftarrow \emptyset$, $M_{boun} \leftarrow \emptyset$, $M_{belo} \leftarrow \emptyset$, $M_{intra\text{-}path} \leftarrow \emptyset$, $M_{inter\text{-}path} \leftarrow \emptyset$, $M_{inter\text{-}end} \leftarrow \emptyset$, $M_{cout} \leftarrow \emptyset$
2: **for** each POI $u \in P$ **do**
3:     $value_1 \leftarrow \emptyset$, $value_2 \leftarrow \emptyset$
4:     **for** each point v on C such that **do**
5:         **if** v is a point that belongs to the partition cell corresponding to u **then**
6:             $value_1 \leftarrow value_1 \cup \{v\}$
7:         **if** v is a boundary point of the partition cell corresponding to u **then**
8:             $value_2 \leftarrow value_2 \cup \{v\}$
9:         $key \leftarrow u$, $M_{cont} \leftarrow M_{cont} \cup \{key, value_1\}$, $M_{boun} \leftarrow M_{boun} \cup \{key, value_2\}$
10: **for** each point u on C such that $u \notin B$ **do**
11:     **for** each POI $v \in P$ such that u belongs to the partition cell corresponding to v **do**
12:         $key \leftarrow u$, $value \leftarrow v$, $M_{belo} \leftarrow M_{belo} \cup \{key, value\}$
13: **for** each POI $u \in P$ **do**
14:     $P_{dest}(u) \leftarrow$ retrieved from M_{cont} using u as key $\cup$ retrieved from M_{boun} using u as key
15:     calculate the exact shortest paths passing on C from u to each point in $P_{dest}(u)$ simultaneously using algorithm *FastFly*
16:     **for** each point $v \in P_{dest}(u)$ **do**
17:         $key \leftarrow \langle u, v \rangle$, $value \leftarrow \Pi^*(u, v|C)$, $M_{intra\text{-}path} \leftarrow M_{intra\text{-}path} \cup \{key, value\}$
18: $\{M_{inter\text{-}path}, M_{inter\text{-}end}\} \leftarrow$ *RC-Oracle-Construction* (C, B)
19: **return** B, M_{cont}, M_{boun}, M_{belo}, $M_{intra\text{-}path}$, $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$

(2) *Different partition cell* (lines 3-16): In Figures 7 (i) and (j), given k as a source that is not a POI, and a as a destination that is a POI, since $\langle a, k \rangle \notin M_{intra\text{-}path}.key$, there are two steps.

(i) *Source and destination selection* (lines 5-15): In Figures 7 (e), (i) and (j), $k \in M_{belo}.key$, we retrieve the POI e from M_{belo} using key k, retrieve $\{j, \dots\}$ (except k) from M_{belo} and $\{i, \dots\}$ from M_{boun} using key e, so we have $P_{dest}(k) = \{i, j, \dots\}$, we calculate the intra-paths in orange lines using algorithm *FastFly*, and store the key-value pairs in $M_{intra\text{-}path}$ in Figure 7 (j). In Figure 7 (i), we have $B_1 = \{i, \dots\}$ of points in red around k, and $B_2 = \{g, h, \dots\}$ of points in red around a.

(ii) *Shortest path query* (line 16): In Figures 7 (k), we use the intra-paths $\Pi^*(a, h|C)$ and $\Pi^*(k, i|C)$ in green and orange lines, and the inter-path $\Pi_{inter\text{-}path}(h, i|C) = \Pi^*(h, i|C)$ in light blue line to approximate $\Pi_{TI\text{-}Oracle}(a, k|C)$.

5.4 Implementation Details of Proximity Query Algorithms

Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k, and a range value r, we can answer other proximity queries, i.e., the *kNN* and range queries using the two oracles. Since the proximity query algorithms for *TI-Oracle* and *TI-Oracle-A2A* are similar, we use *TI-Oracle* as an example for illustration.

Notation: Let $M'_{inter\text{-}path}$ be an additional inter-path map table, which is similar to $M_{inter\text{-}path}$, but $M'_{inter\text{-}path}$ not only stores the exact inter-paths, but also store all other inter-paths. In Figure 7 (g),

Algorithm 4 *TI-Oracle-Query* $(C, P, s, t, B, M_{cont}, M_{boun}, M_{belo}, M_{intra\text{-}path}, M_{inter\text{-}path}, M_{inter\text{-}end})$

Input: a point cloud C, a set of POIs P, a source s that can be any point on C, a destination t that is a POI in P, the set of boundary points B, the containing point map table M_{cont}, the boundary point map table M_{boun}, the belonging point map table M_{belo}, the intra-path map table $M_{intra\text{-}path}$, the inter-path map table $M_{inter\text{-}path}$ and the inter-endpoint map table $M_{inter\text{-}end}$

Output: the updated intra-path map table $M_{intra\text{-}path}$ and the shortest path $\Pi_{TI\text{-}Oracle}(s, t|C)$ between s and t passing on C

1: **if** $\langle s, t\rangle \in M_{intra\text{-}path}.key$ **then**
2: retrieve $\Pi^*(s, t|C)$ as $\Pi_{TI\text{-}Oracle}(s, t|C)$ using $\langle s, t\rangle$
3: **else if** $\langle s, t\rangle \notin M_{intra\text{-}path}.key$ **then**
4: $B_1 \leftarrow \emptyset, B_2 \leftarrow \emptyset$
5: **if** $s \in M_{belo}.key$ **then**
6: $u \leftarrow$ retrieved from M_{belo} using s as key
7: $P_{dest}(s) \leftarrow$ retrieved from M_{cont} using u as key (except s) $\cup$ retrieved from M_{boun} using u as key
8: calculate the exact shortest paths passing on C from s to each point in $P_{dest}(s)$ simultaneously using algorithm *FastFly*
9: **for** each point $v \in P_{dest}(s)$ such that $\langle s, v\rangle \notin M_{intra\text{-}path}.key$ **do**
10: $key \leftarrow \langle s, v\rangle$, $value \leftarrow \Pi^*(s, v|C)$, $M_{intra\text{-}path} \leftarrow M_{intra\text{-}path} \cup \{key, value\}$
11: $B_1 \leftarrow$ retrieved from M_{boun} using u as key
12: **else if** $s \in B$ (resp. $s \in P$) **then**
13: $B_1 \leftarrow \{s\}$ (resp. retrieved from M_{boun} using s as key)
14: **if** $t \in B$ (resp. $t \in P$) **then**
15: $B_2 \leftarrow \{t\}$ (resp. retrieved from M_{boun} using t as key)
16: calculate $\Pi_{TI\text{-}Oracle}(s, t|C)$ by concatenating $\Pi^*(s, s'|C)$, $\Pi_{inter\text{-}path}(s', t'|C)$ and $\Pi^*(t', t|C)$ such that $|\Pi_{TI\text{-}Oracle}(s, t|C)| = \min_{s' \in B_1, t' \in B_2}[|\Pi^*(s, s'|C)| + |\Pi_{inter\text{-}path}(s', t'|C)| + |\Pi^*(t', t|C)|]$, where $\Pi^*(s, s'|C)$ and $\Pi^*(t', t|C)$ are retrieved from $M_{intra\text{-}path}$ using $\langle s, s'\rangle$ and $\langle t', t\rangle$ as key (if $s = s'$, $\Pi^*(s, s'|C)$ is just a point s, and similar for t and t'), $\Pi_{inter\text{-}path}(s', t'|C)$ is calculated using the shortest path query phase of *RC-Oracle* with $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ using s' and t' as input
17: **return** $M_{intra\text{-}path}$ and $\Pi_{TI\text{-}Oracle}(s, t|C)$

$M_{inter\text{-}path}$ can only store 6 exact inter-paths in light blue lines with h as a source, but apart from these, $M_{inter\text{-}path}$ can also store the inter-path between g and i.

Detail and example: We perform a linear scan on the shortest path query result between q and each object in O. We first initialize $M'_{inter\text{-}path}$ to be empty. For each shortest path query, we follow Algorithm 4, there is only one change in line 16. For finding the inter-path $\Pi_{inter\text{-}path}(s', t'|C)$, we first search in $M'_{inter\text{-}path}$. There are two cases:

(1) *Inter-path retrieval by $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$*: If $\langle s', t'\rangle \notin M'_{intra\text{-}path}.key$, we calculate $\Pi_{inter\text{-}path}(s', t'|C)$ using the shortest path query phase of *RC-Oracle* with $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ using s' and t' as input, and store $\langle s', t'\rangle$ as key and $\Pi_{inter\text{-}path}(s', t'|C)$ as value in $M'_{inter\text{-}path}$ (in Figures 7 (f) - (h), suppose that $M'_{intra\text{-}path}$ is empty, given g and i, $\Pi_{inter\text{-}path}(g, i|C)$ is approximated by $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$ using $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$, we store $\langle g, i\rangle$ as key and $\Pi_{inter\text{-}path}(g, i|C)$ as value in $M'_{inter\text{-}path}$).

(2) *Inter-path retrieval by $M'_{inter\text{-}path}$*: If $\langle s', t'\rangle \in M'_{intra\text{-}path}.key$, we retrieve $\Pi_{inter\text{-}path}(s', t'|C)$ from $M'_{inter\text{-}path}$ using $\langle s', t'\rangle$ (in Figure 7 (g), we may need to find the inter-path between g and i again, but since $\langle g, i\rangle \in M'_{intra\text{-}path}.key$, we can directly retrieve $\Pi_{inter\text{-}path}(g, i|C)$ using one table $M'_{inter\text{-}path}$, without searching in two tables $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$).

5.5 Theoretical Analysis

5.5.1 ***TI-Oracle and TI-Oracle-A2A.*** The analysis of *TI-Oracle* and *TI-Oracle-A2A* are in Theorem 5.1.

Theorem 5.1. *The oracle construction time, oracle size and shortest path query time of (1) TI-Oracle are $O(\frac{N\log N}{\epsilon}+Nn+n\log n)$, $O(\frac{N}{\epsilon})$, $O(1)$ and (2) TI-Oracle-A2A are $O(\frac{N\log N}{\epsilon}+N\sqrt{N}+\sqrt{N}\log\sqrt{N})$, $O(\frac{N}{\epsilon})$, $O(1)$, respectively. TI-Oracle-A2P-SmCon always have $|\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any s on C and any t in P, and RC-Oracle-A2A always have $|\Pi_{RC\text{-}Oracle\text{-}A2A}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pairs of points s and t on C.*

Proof. We give the proof for *TI-Oracle* as follows.

Firstly, we show the *oracle construction time.* (1) In *partition cells calculation* step, it needs $O(n\log n)$ time. Since there are n POIs, and use them to construct a 2D Voronoi diagram. (2) In *M_{cont}, M_{boun} and M_{belo} calculation* step, it needs $O(Nn)$ time. Since there are N points and n POIs (i.e., partition cells), we need to check which point belongs to which partition cells for each of them. (3) In *intra-paths calculation* step, it needs $O(n)$ time. It needs to use $O(n)$ POIs as a source to run algorithm *FastFly* for the exact shortest paths calculation to the corresponding boundary points of the POI, and each algorithm *FastFly* needs $O(1)$ time (since these boundary points are close to their corresponding POI). (4) In *inter-paths calculation* step, it needs $O(\frac{N\log N}{\epsilon})$ time. Since there are at most $O(N)$ boundary points and we use *RC-Oracle* for the calculation of inter-paths, we just need to change n to N in the oracle construction time of *RC-Oracle.* (5) So the oracle construction time is $O(\frac{N\log N}{\epsilon}+Nn+n\log n)$.

Secondly, we show the *oracle size.* (1) For M_{cont}, M_{bond}, M_{belo} and $M_{intra\text{-}path}$, their sizes are all $O(N)$ since there are N points on C. (2) For $M_{inter\text{-}end}$ and $M_{inter\text{-}end}$, their sizes are $O(N)$ and $O(\frac{N}{\epsilon})$. Since there are at most $O(N)$ boundary points, $M_{inter\text{-}end}$ and $M_{inter\text{-}end}$ correspond to M_{end} and M_{end} in *RC-Oracle*, we just need to change n to N for these two tables in *RC-Oracle.* (3) So the oracle size is $O(\frac{N}{\epsilon})$.

Thirdly, we show the *shortest path query time.* (1) If $\Pi^*(s,t|C) \in M_{intra\text{-}path}$, the shortest path query time is $O(1)$. (2) If $\Pi^*(s,t|C) \notin M_{intra\text{-}path}$, we need to run algorithm *FastFly* for the exact shortest paths calculation to the corresponding boundary points of s or t in $O(1)$ time, and run the shortest path query phase of *RC-Oracle* in $O(1)$ time. Thus, the shortest path query time of *TI-Oracle* is $O(1)$.

Fourthly, we show the *error bound.* Given s and t, let s' and t' be the boundary points on of the partition cell that s and t belong to, such that we concatenate $\Pi^*(s,s'|C)$, $\Pi_{inter\text{-}path}(s',t'|C)$ and $\Pi^*(t',t|C)$ to calculate $\Pi_{TI\text{-}Oracle}(s,t|C)$, let p and q be the boundary points on of the partition cell that s and t belong to, such that they lie on $\Pi^*(s,t|C)$. We have $|\Pi_{TI\text{-}Oracle}(s,t|C)| = |\Pi^*(s,s'|C)|+|\Pi_{inter\text{-}path}(s',t'|C)|+|\Pi^*(t',t|C)| \leq |\Pi^*(s,p|C)|+|\Pi_{inter\text{-}path}(p,q|C)|+|\Pi^*(q,t|C)| \leq |\Pi^*(s,p|C)| + (1+\epsilon)|\Pi^*(p,q|C)| + |\Pi^*(q,t|C)| \leq (1+\epsilon)|\Pi^*(s,p|C)| + (1+\epsilon)|\Pi^*(p,q|C)| + (1+\epsilon)|\Pi^*(q,t|C)| = (1+\epsilon)|\Pi^*(s,t|C)|$. The first equation is because $\Pi_{TI\text{-}Oracle}(s,t|C)$ is calculated by the $\Pi^*(s,s'|C)$, $\Pi_{inter\text{-}path}(s',t'|C)$ and $\Pi^*(t',t|C)$. The second inequality is because s' and t' are the boundary points that result in the shortest distance of $\Pi_{TI\text{-}Oracle}(s,t|C)$. The third inequality is because $|\Pi_{inter\text{-}path}(p,q|C)| \leq (1+\epsilon)|\Pi^*(p,q|C)|$, i.e., the error bound of *RC-Oracle* for the inter-path. The fourth inequality is because $|\Pi^*(s,p|C)| \leq (1+\epsilon)|\Pi^*(s,p|C)|$ and $|\Pi^*(q,t|C)| \leq (1+\epsilon)|\Pi^*(q,t|C)|$. The fifth inequality is because p and q are the boundary points that result in the shortest distance of $\Pi^*(s,t|C)$.

We give the proof for *TI-Oracle-A2A* as follows. We need to change (1) n to $\sqrt{N}$ in the oracle construction time and oracle size since we select $\sqrt{N}$ points as POIs, and (2) for any s on C and any t in P to any pairs of points s and t on C in the error bound. The other analysis is the same as *TI-Oracle*. □

5.5.2 ***Proximity query algorithms***. We provide analysis on the proximity query algorithms, including the *kNN* and range queries using *TI-Oracle* and *TI-Oracle-A2A* in Theorem 5.2.

THEOREM 5.2. *The query time and error rate of both the kNN and range queries by using TI-Oracle and TI-Oracle-A2A are both* $O(n')$ *and* $1 + \epsilon$*, respectively.*

PROOF SKETCH. The *query time* of *TI-Oracle* and *TI-Oracle-A2A* is due to the usages of the shortest path query phase of them for n' times. The *error rate* is due to its definition and the error of *TI-Oracle* and *TI-Oracle-A2A*. The detailed proof appears in our technical report [13]. □

6 EMPIRICAL STUDIES

6.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.2 GHz CPU and 512GB memory. All algorithms were implemented in C++. Our experimental setup generally follows the setups in the literature [35, 36, 45, 62, 63]. We conducted experiments with point clouds and *TIN*s as input, separately.

Datasets: (1) Point cloud datasets: We conducted our experiment based on 34 real point cloud datasets in Table 2, where the subscript p means a point cloud. For BH_p and EP_p datasets, they are represented as a point cloud with 8km × 6km covered region. For GF_p, LM_p and RM_p, we first obtained the satellite map from Google Earth [2] with 8km × 6km covered region, and then used Blender [1] to generate the point cloud. These five original datasets have a resolution of 10m × 10m [24, 45, 58, 62, 63]. We extracted 500 POIs using OpenStreetMap [62, 63] for these datasets in the P2P query. For small-version datasets, we use the same region of the original datasets with a (lower) resolution of 70m × 70m and the dataset generation procedure in [45, 62, 63] to generate them. This procedure can be found in our technical report [13]. In addition, we have six sets of multi-resolution datasets with different numbers of points generated using the original and small-version datasets with the same procedure. (2) *TIN* datasets: Based on the 34 point cloud datasets, we triangulate [53] them and generate another 34 *TIN* datasets, and use t as the subscript. For example, BH_t means a *TIN* dataset generated using the BH_p point cloud dataset.

Algorithms: (1) Algorithms that support the shortest path query (and also other proximity queries) on a point cloud (i.e., algorithms for solving the problem studied in this paper): We adapted existing algorithms, originally designed for the problem on *TIN*s, for our problem on point clouds by performing the triangulation approach on the point cloud to obtain a *TIN* [53] (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory) so that the existing algorithm could be used. Their algorithm names are appended by "*-Adapt*". We have four on-the-fly algorithms, i.e., (i) *DIO-Adapt* [64], (ii) *Kaul-Adapt* [35], (iii) *Dijk-Adapt* [36], and (iv) *FastFly*: our algorithm. We have eleven oracles, i.e., (v) *SE-Oracle-Adapt*: the best-known oracle [62, 63] for the P2P query on a point cloud, (vi) *EAR-Oracle-Adapt*: the best-known oracle [33] for the A2A query on a point cloud, (vii) *RC-Oracle-Naive*: the naive version of *RC-Oracle* without shortest paths approximation step for the P2P query on a point cloud, (viii) *RC-Oracle*: the oracle for the P2P query on a point cloud proposed in previous conference paper [69], (ix) *SE-Oracle-Adapt-A2A*: the adapted *SE-Oracle-Adapt* (by creating POIs as Steiner points on faces of the constructed *TIN* by the point cloud, see more details in [62, 63]) for the A2A query on a point cloud, (x) *RC-Oracle-Naive-A2A*: the adapted *RC-Oracle-Naive* in a similar way of *RC-Oracle-A2A* for the A2A query on a point cloud, (xi) *RC-Oracle-A2A*: the oracle for the A2A query on a point cloud proposed in previous conference paper [69], (xii, xiii, xv) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle*: the oracles for

Table 2. Point cloud datasets

Name	$\|N\|$
Original dataset	
BearHead (BH_p) [5, 62, 63]	0.5M
EaglePeak (EP_p) [5, 62, 63]	0.5M
GunnisonForest (GF_p) [7]	0.5M
LaramieMount (LM_p) [8]	0.5M
RobinsonMount (RM_p) [3]	0.5M
Small-version dataset	
BH_p*-small*	10k
EP_p*-small*	10k
GF_p*-small*	10k
LM_p*-small*	10k
RM_p*-small*	10k
Multi-resolution dataset	
BH_p multi-resolution	1M, 1.5M, 2M, 2.5M
EP_p multi-resolution	1M, 1.5M, 2M, 2.5M
GF_p multi-resolution	1M, 1.5M, 2M, 2.5M
LM_p multi-resolution	1M, 1.5M, 2M, 2.5M
RM_p multi-resolution	1M, 1.5M, 2M, 2.5M
EP_p*-small* multi-resolution	20k, 30k, 40k, 50k

the A2P query on a point cloud proposed in this journal paper, and (xvi) *TI-Oracle-A2A*: the oracle for the A2A query on a point cloud proposed in this journal paper.

(2) Algorithms that support the shortest path query (and also other proximity queries) on a *TIN* (i.e., algorithms for solving the problem studied by previous studies [33, 62, 63]): Similarly, we have four on-the-fly algorithms, i.e., (i) *DIO* [64], (ii) *Kaul* [35], (iii) *Dijk* [36], (iv) *FastFly-Adapt*: our adapted algorithm (for the queries on a *TIN*) that calculates the shortest path passing on a conceptual graph of a *TIN*, where the vertices of this conceptual graph are formed by the vertices of the given *TIN*, and the edges of this graph are formed by adding edges between each vertex and its 8 neighbor vertices (this conceptual graph is similar to the one in Figure 2 (c), we store it as a data structure in the memory and clear the given *TIN* from the memory). We have eleven oracles, i.e., (v) *SE-Oracle*: the best-known oracle [62, 63] for the P2P query on a *TIN*, (vi) *EAR-Oracle*: the best-known oracle [33] for the AR2AR query on a *TIN*, (vii) *RC-Oracle-Naive-Adapt*: the adapted naive version of our oracle without shortest paths approximation step for the P2P query on a *TIN* that calculates the shortest path passing on a conceptual graph of a *TIN*, (viii) *RC-Oracle-Adapt*: the adapted oracle for the P2P query on a *TIN* that calculates the shortest path passing on a conceptual graph of a *TIN* proposed in previous conference paper [69], (ix) *SE-Oracle-AR2AR*: the adapted *SE-Oracle* (by creating POIs as Steiner points on faces of the *TIN*, see more details in [62, 63]) for the AR2AR query on a *TIN*, (x) *RC-Oracle-Naive-Adapt-AR2AR*: the adapted *RC-Oracle-Naive* in a similar way of *RC-Oracle-Naive-A2A* for the AR2AR query on a *TIN* which calculates the shortest path passing on a conceptual graph of a *TIN*, (xi) *RC-Oracle-Adapt-AR2AR*: the adapted oracle in a similar way of *RC-Oracle-A2A* for the AR2AR query on a *TIN* which calculates the shortest path passing on a conceptual graph of a *TIN* proposed in previous conference paper [69], (xii, xiii, xv) *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*: the adapted oracles in a similar way of *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* for the AR2P query on a a *TIN* which calculates the shortest path passing on a conceptual graph of a *TIN* proposed in this journal paper, and (xvi) *TI-Oracle-Adapt-AR2AR*: the adapted oracle in a similar way of *TI-Oracle-A2A* for the AR2AR query on a *TIN* which calculates the shortest path passing on a conceptual graph of a *TIN* proposed in this journal paper.

Query Generation: We conducted all proximity queries, i.e., (1) shortest path query, (2) all objects *kNN* query, and (3) all objects range query. (1) For the shortest path query, we issued 100 query instances where for each instance, we randomly chose (i) two points in P for the P2P query on a point cloud or a *TIN*, (ii) one point in P and one point on the point cloud (resp. *TIN*) for the A2P query on a point cloud (resp. the AR2P query on a *TIN*), or (iii) two points on the point cloud (resp. *TIN*) for the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), one as a source and the other as a destination. The average, minimum, and maximum results were reported. In the experimental result figures, the vertical bar and the points mean the minimum, maximum, and average results. (2 & 3) For all objects *kNN* query and range query, we perform the proximity query algorithms for our oracles in Sections 4.5 and 5.4, and a linear scan for other baselines (as described in [63]) using all objects as query objects. In the P2P query on a point cloud or a *TIN*, A2P query on a point cloud or AR2P query on a *TIN*, these objects are POIs in P. In the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), we randomly select 2500 points on the point cloud (resp. *TIN*) as objects. Since we perform linear scans or use the calculated distance stored in M_{path}, $M_{intra\text{-}path}$ or $M_{inter\text{-}path}$ for proximity query algorithms, the value of k and r will not affect their query time, we set $k = 3$ and $r = 1$km.

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

6.2 Experimental Results for *TIN*s

We first study proximity queries on *TIN*s (studied by previous studies [33, 62, 63]) to justify why our proximity queries on *point clouds* are useful in practice. We have the following settings. (1) The distance of the path calculated by *DIO* is used for distance error calculation since the path is the exact shortest surface path passing on the *TIN*. (2) For the P2P query on a *TIN*, we compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *DIO*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets (with default 50 POIs), and compared *RC-Oracle-Adapt*, *DIO*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets (with default 500 POIs) since *SE-Oracle*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours). (3) For the AR2P and AR2AR queries on a *TIN*, we compared *SE-Oracle-AR2AR*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on small-version datasets (with default 50 POIs for the AR2P query), and compared *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on large-version datasets (with default 500 POIs for the AR2P query) since *SE-Oracle-AR2AR*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt-AR2AR* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours).

6.2.1 ***Baseline comparisons***. We study the effect of ϵ and n for the P2P, AR2P and AR2AR queries on a *TIN* in this subsection. We study the effect of N for these queries in our technical report [13].

Effect of ϵ for the P2P query on a *TIN*. In Figure 9, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the P2P query on a *TIN*. Although a *TIN* is given as input, *RC-Oracle-Adapt* performs better than *SE-Oracle*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt* in terms of the oracle construction time, oracle size and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *DIO* (although *FastFly-Adapt* needs to construct a conceptual graph from the given *TIN*, and there is no other additional steps for *DIO*), since the query region of the path calculated by *FastFly-Adapt* is smaller than that of *DIO*. The distance error of *FastFly-Adapt* (i.e., 0.002) is very small compared with that of *DIO* (i.e., without error), and much much smaller than that of *Dijk* (i.e., 0.1). This motivates us to conduct experiments on point clouds. The *kNN* query error and range query error are all equal to 0 (due to the small distance error), so their results are omitted.

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

Effect of ϵ for the AR2P query on a *TIN*. In Figure 11, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the AR2P query on a *TIN*. The oracle construction time, memory usage and oracle size of *RC-Oracle-Adapt-AR2P-SmCon* is the smallest in all oracles since it has the same oracle construction process as of *RC-Oracle-Adapt*, but its shortest path query time is larger than other oracles (but still smaller than *FastFly-Adapt*) since it can terminate earlier when using *FastFly-Adapt* in the shortest path query phase. Thus, it performs well in the case of fewer proximity queries. The oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* are also very small and their shortest path query time are also very small due to their earlier termination during oracle construction and tight information stored in the oracles.

Effect of n for the AR2P query on a *TIN*. In Figure 12, we tested 5 values of n from {50, 100, 150, 200, 250} on EP_t dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons for the AR2P query on a *TIN*. When $n < 100$ (resp. $n \geq 100$), the oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle-Adapt*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).

AR2AR query on a *TIN*. In Figures 11 and 12, we also compared oracles for the AR2AR query on a *TIN*. *SE-Oracle-AR2AR*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR* and *TI-Oracle-Adapt-AR2AR* can answer the AR2AR query on a *TIN*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time due to their earlier termination during oracle construction and tight information stored in the oracles.

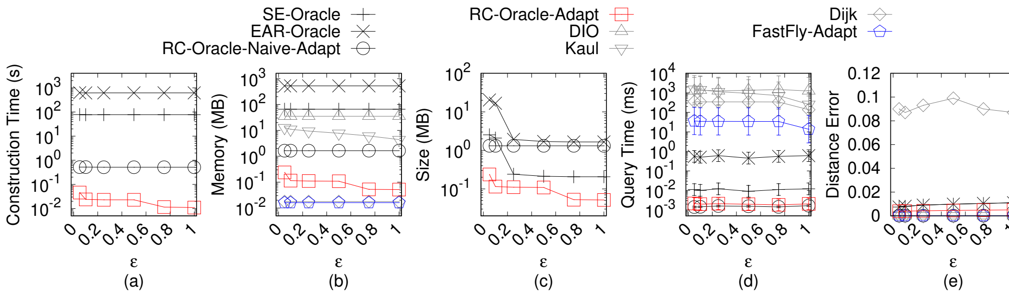

Fig. 9. Baseline comparisons (effect of ϵ on *BH_t-small TIN* dataset for the P2P query)

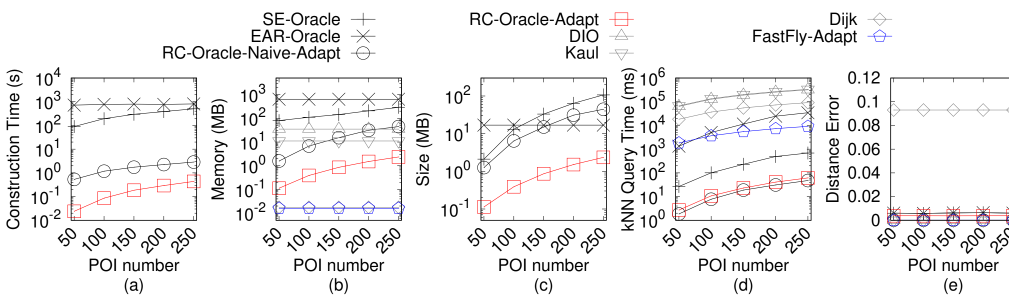

Fig. 10. Baseline comparisons (effect of n on *EP_t-small TIN* dataset for the P2P query)

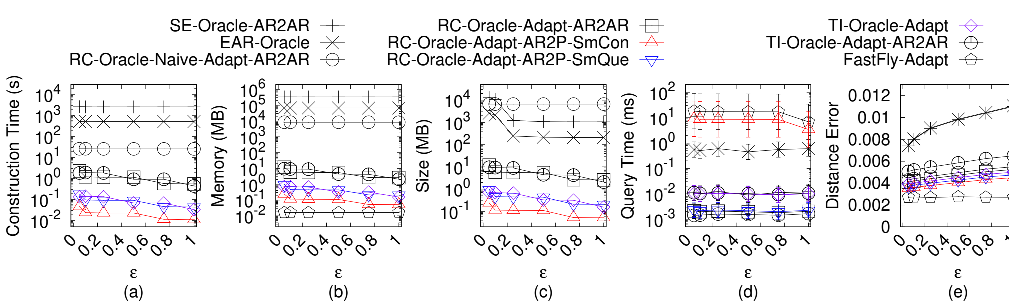

Fig. 11. Baseline comparisons (effect of ϵ on *BH_t-small TIN* dataset for the AR2P query)

6.3 Experimental Results for Point Clouds

Now, we understand the effectiveness of proximity queries on *point clouds*. In this section, we then study proximity queries on *point clouds* using the algorithms in Table 1. We have the following setting. (1) The distance of the path calculated by *FastFly* is used for distance error calculation since the path is the exact shortest path passing on the point cloud. (2) We compared similar algorithms on small/large-version datasets with the same reasons for *TIN*s, we just need to append "*-Adapt*" for *SE-Oracle*, *EAR-Oracle*, *DIO*, *Kaul*, *Dijk*, *SE-Oracle-AR2AR*, remove "*-Adapt*" for *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *FastFly-Adapt*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR*, and change AR2P (and AR2AR) to A2P (and A2A) for these algorithms, and change *TIN* to point cloud.

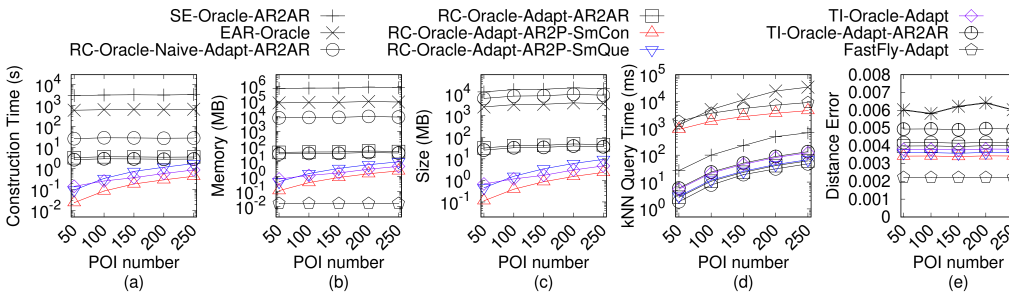

Fig. 12. Baseline comparisons (effect of n on EP_t-*small TIN* dataset for the AR2P query)

6.3.1 ***Baseline comparisons.*** We study the effect of ϵ, n and N for the P2P, A2P and A2A queries on a point cloud in this subsection.

Effect of ϵ for the P2P query on a point cloud. In Figure 13, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on GF_p-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the P2P query on a point cloud. (1) For *RC-Oracle* and the best-known oracle *SE-Oracle-Adapt*, (i) the oracle construction time and memory consumption, (ii) oracle size, and (iii) shortest path query time of *RC-Oracle* are all smaller than *SE-Oracle-Adapt*, since (i) *SE-Oracle-Adapt* has the *bad criterion for algorithm earlier termination* drawback, it cannot terminate the *SSAD* algorithm earlier, so it requires more time and memory, (ii) *RC-Oracle* can terminate the *SSAD* algorithm earlier and store fewer paths, (iii) *RC-Oracle*'s shortest path query time is $O(1)$, while the time is $O(h^2)$ for *SE-Oracle-Adapt*. (2) *RC-Oracle* performs better than other on-the-fly algorithms in terms of the shortest path query time since it is an oracle. (3) Algorithm *FastFly* performs better than other on-the-fly algorithms in terms of the shortest path query time since it calculates the shortest path passing on a point cloud. (4) In Figures 13 (a) & (b), regarding the oracle construction time and memory consumption, the variation of ϵ (i) has a large effect on *RC-Oracle*, but due to the log scale used in the experimental figures, the effect is not obvious (e.g., the oracle construction time and memory consumption of *RC-Oracle* with $\epsilon = 1$ are both up to 5 times smaller than that of the case when $\epsilon = 0.05$), (ii) has a small effect on *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*, because even when ϵ is large, they cannot terminate the *SSAD* algorithm earlier for most of the cases due to their *bad criterion for algorithm earlier termination* drawback, and (iii) has no effect on *RC-Oracle-Naive* since it is independent of ϵ. (5) The *kNN* and range queries time of *RC-Oracle* are much smaller than the on-the-fly algorithms. (6) The distance error of *RC-Oracle* is close to 0.

Effect of n for the P2P query on a point cloud. In Figure 14, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on LM_p dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons for the P2P query on a point cloud. Since *RC-Oracle* is an oracle, its *kNN* query time is smaller than on-the-fly algorithms.

Effect of N (scalability test) for the P2P query on a point cloud. In Figure 15, 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 0.25 for baseline comparisons for the P2P query on a point cloud. The oracle construction time of *RC-Oracle* is only 80s $\approx$ 1.3 min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of *RC-Oracle*. The range query time of *RC-Oracle* is the smallest.

Effect of ϵ for the A2P query on a point cloud. In Figure 16, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on GF_p-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the A2P query on a point cloud. (1) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon* and *TI-Oracle-A2P-SmCon* perform better than the best-known oracle *EAR-Oracle-Adapt* for the A2P query on a point cloud in terms of oracle construction time, oracle size and shortest path query time due to the *bad criterion for algorithm earlier termination* drawback of *EAR-Oracle-Adapt*. (2) We also include *RC-Oracle-A2A* and *TI-Oracle-A2A* as baseline oracles to show that *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon* and *TI-Oracle-A2P-SmCon* perform better in terms of oracle construction time and oracle size, since are they are designed for the A2P query on a point cloud.

Effect of n for the A2P query on a point cloud. In Figure 17, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on LM_p dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons for the A2P query on a point cloud. (1) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon* and *TI-Oracle-A2P-SmCon* also perform better than *RC-Oracle-A2A*. (2) The oracle construction time, memory usage and oracle size of *RC-Oracle-AR2P-SmCon* are the smallest, but its shortest path query time is larger than other oracles (but still smaller than *FastFly*) due to the same reason as that of *RC-Oracle-Adapt-AR2P-SmCon* for *TIN*s. Thus, it performs well in the case of fewer proximity queries. (3) The oracle construction time of *RC-Oracle-AR2P-SmQue* and *TI-Oracle* are also very small and their shortest path query time are also very small due to the same reason as those of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* for *TIN*s. When $n < 500$ (resp. $n \geq 500$), the oracle construction time of *RC-Oracle-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).

Effect of N (scalability test) for the A2P query on a point cloud. In Figure 18, 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 0.25 for baseline comparisons for the A2P query on a point cloud. The oracle construction time of *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* are only 80s $\approx$ 1.3 min, 310s $\approx$ 5.1 min and 250s $\approx$ 4.1min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of them.

A2A query on a point cloud. In Figures 16, 17 and 18, we also compared oracles for the A2A query on a point cloud. *SE-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A* can answer the A2A query on a point cloud. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time.

6.3.2 ***Ablation study***. We denote *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *SE-Oracle-FastFly-Adapt-A2A* to be other adapted oracles of *SE-Oracle-Adapt*, *EAR-Oracle-Adapt* and *SE-Oracle-Adapt-A2A* that use algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without constructing a *TIN*. In Figure 19, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on LM_p dataset by setting N to be 0.5M and n to be 500 for ablation study among *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle* for the P2P query on a point cloud, such that they only differ by the oracle construction. The oracle construction time and shortest path query time of *RC-Oracle* perform better than *SE-Oracle-FastFly-Adapt* and *EAR-Oracle-FastFly-Adapt*. In Figure 20, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on RM_p dataset by setting N to be 0.5M and n to be 500 for ablation study among *SE-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A* for the A2P query on a point cloud, such that they only differ by the oracle construction. The oracle construction time, oracle size and shortest path query time of *RC-Oracle-A2P-SmQue* and *TI-Oracle* also perform better than *SE-Oracle-FastFly-Adapt-A2A* and *EAR-Oracle-FastFly-Adapt*. In Figure 20, we also compared oracles for the A2A query on a point cloud since *SE-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A* and *TI-Oracle-A2A* can answer the A2A query on a point cloud. *RC-Oracle-A2A* and *TI-Oracle-A2A* also perform better.

6.3.3 ***Comparisons with other proximity queries oracles and variation oracles on a point cloud***. We compared *SU-Oracle-Adapt* [58] (i.e., the oracle designed for the *kNN* query) and some variations of our oracles related to *SU-Oracle-Adapt* in our technical report [13]. (1) For the P2P

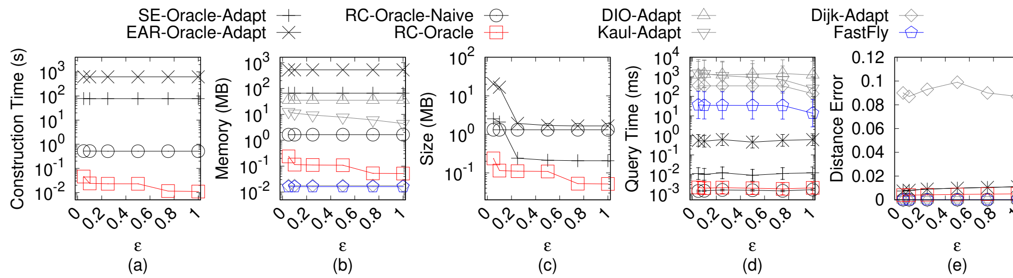

Fig. 13. Baseline comparisons (effect of ϵ on *GF_p-small* point cloud dataset for the P2P query)

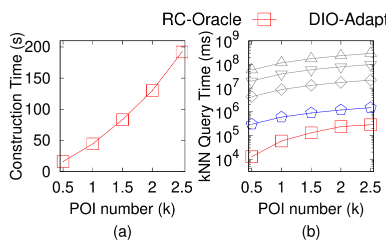

Fig. 14. Baseline comparisons (effect of n on *LM_p* point cloud dataset for the P2P query)

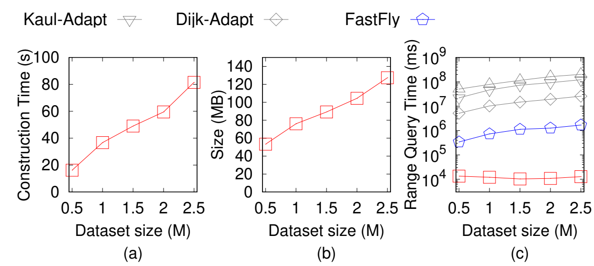

Fig. 15. Baseline comparisons (effect of N on *RM_p* point cloud dataset for the P2P query)

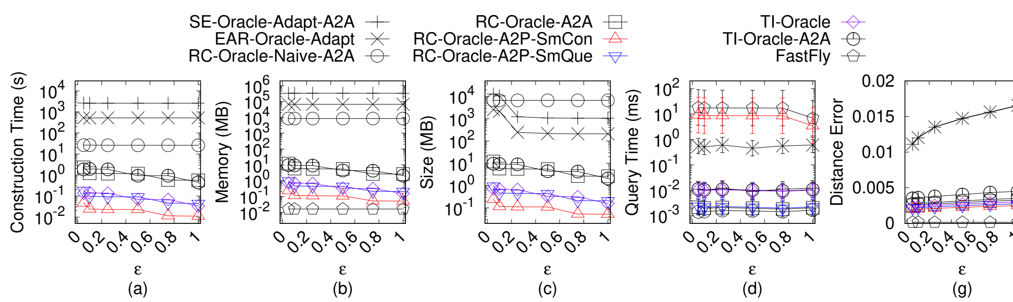

Fig. 16. Baseline comparisons (effect of ϵ on *GF_p-small* point cloud dataset for the A2P query)

query on a point cloud, we compared *SU-Oracle-Adapt*, *RC-Oracle* using the naive proximity query algorithm as mentioned in Section 4.2.5 and *RC-Oracle*. For a point cloud with 2.5M points and 500 POIs, the *kNN* query time of *RC-Oracle* is 12.5s, but the time is 1520s $\approx$ 25 min for *SU-Oracle-Adapt*, and 25s for *RC-Oracle* with the naive proximity query algorithm (since the shortest path query time of *RC-Oracle* is $O(1)$, and we do not need to perform linear scans over all the POIs in our efficient proximity query algorithm of *RC-Oracle*). (2) For the A2P (and also A2A) query, we compared *SU-Oracle-Adapt*, *TI-Oracle* using the naive proximity query algorithm as mentioned in Section 5.2.3, *TI-Oracle* without using M_{belo} while using an R-tree and a set of 2D boxes as tree nodes to index the belonging point information (used in study [58]), *TI-Oracle* without using partition cells while using tight/loose surface indexes (used in study [58]), *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A*. Since we have shown the usefulness of the proximity algorithm of *RC-Oracle* for the P2P query on a point cloud, there is no need to compare

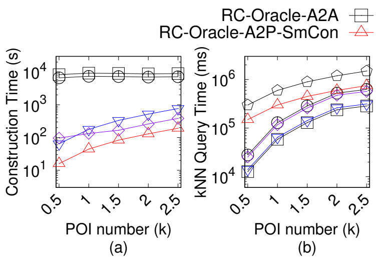

Fig. 17. Baseline comparisons (effect of n on LM_p point cloud dataset for the A2P query)

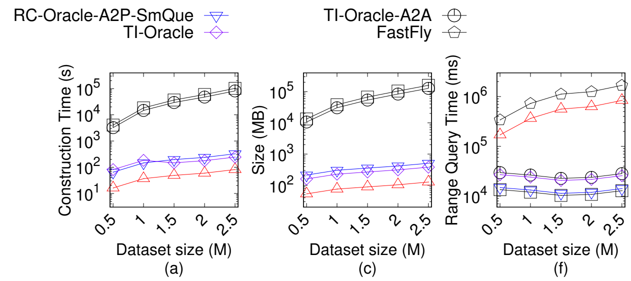

Fig. 18. Baseline comparisons (effect of N on RM_p point cloud dataset for the A2P query)

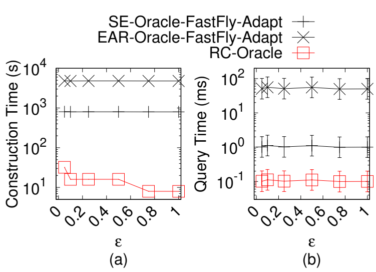

Fig. 19. Ablation study on LM_p point cloud dataset for the P2P query

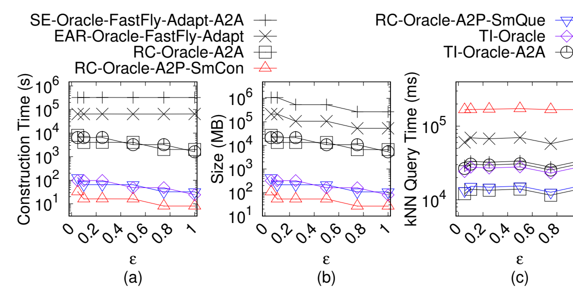

Fig. 20. Ablation study on RM_p point cloud dataset for the A2P query

RC-Oracle-A2A, *RC-Oracle-A2P-SmCon* and *RC-Oracle-A2P-SmQue* using the naive proximity query algorithm as mentioned in Section 4.2.5. (i) Our oracles and variations still perform better than *SU-Oracle-Adapt*. (ii) For a point cloud with 2.5M points and 500 POIs, the oracle size and *kNN* query time of *TI-Oracle* are 350MB and 23s, but are 348MB and 104s of *TI-Oracle* which uses the R-tree. The latter one needs to store the R-tree, 2D boxes and points coordinate information, so it can only slightly reduce the oracle size compared with the former one. But, the latter one needs to use the R-tree to find which partition cell of a query point belongs to, and the leaf nodes of 2D boxes contain more than one endpoint used for creating the partition cell (see Figure 9 of study [58]), so it significantly increases the shortest path query time compared with the former one. (iii) For a point cloud with 2.5M points and 500 POIs, the oracle construction time and *kNN* query time of *TI-Oracle* are 250s ≈ 4.1 min and 23s, but are 500s ≈ 8.2 min and 20s of *TI-Oracle* which uses tight/loose surface indexes. The latter one uses tight/loose surface indexes, so it can gradually expand in *kNN* queries to slightly reduce *kNN* query time compared with the former one. But, the latter one contains more (almost twice compared with *TI-Oracle* using partition cells) boundary points between tight/loose surface indexes and the point cloud, so it significantly increases the oracle construction time compared with the former one.

6.3.4 ***Case study (snowfall evacuation).*** We conducted a case study on an evacuation simulation in Mount Rainier [54] due to the frequent heavy snowfall [55]. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [12] and one may be dead due to asphyxiation [40] if s/he gets buried in the snow. In the case of snowfall, staffs will evacuate tourists in the mountain to the closest hotels immediately for tourists' safety. The time of a human being buried in the snow is expected to be 2.4 hours[1]. The average distance between the viewing platforms and hotels in Mount Rainier National Park is 11.2km [6], and the average human walking speed is 5.1 km/h [9], so the evacuation (i.e., the time of human's walking from the viewing platform to hotels) can be finished in 2.2 (= $\frac{11.2\text{km}}{5.1\text{km/h}}$) hours. Thus, the calculation of the shortest paths is expected to be finished within 12 min (= $2.4 - 2.2$ hours). Our experimental results show that for the P2P query on a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels), (1) the oracle construction time for (i) *RC-Oracle* is 80s $\approx$ 1.3 min and (ii) the best-known oracle for the P2P query on a point cloud *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours, and (2) the query time for calculating 10 nearest hotels of each viewing platform for (i) *RC-Oracle* is 6s, (ii) *SE-Oracle-Adapt* is 75s, and (iii) the best-known on-the-fly approximate shortest surface path query algorithm *Kaul-Adapt* is 80,500s $\approx$ 22.5 hours. Thus, *RC-Oracle* is the best one in the evacuation since 1.3 min + 6s $\leq$ 12 min, but 21.7 hours + 75s $\geq$ 12 min and 22.5 hours $\geq$ 12 min. In addition, for the A2P query on a point cloud under the same setting, (1) the oracle construction time for (i) *TI-Oracle* is 250s $\approx$ 4.1 min, (ii) the best-known oracle for the A2P query on a point cloud *EAR-Oracle-Adapt* is 10,500,000s $\approx$ 121 days, and (iii) the oracle supports A2P query on a point cloud *RC-Oracle-A2A* is 21,000 $\approx$ 5.8 hours, and (2) the query time for calculating 10 nearest hotels of each viewing platform for (i) *TI-Oracle* is 11s, (ii) *EAR-Oracle-Adapt* is 300s, and (iii) *RC-Oracle-Adapt* is 6s. Thus, *TI-Oracle* is the best one in the evacuation since 4.1 min + 12s $\leq$ 12 min, but 121 days + 300s $\geq$ 12 min and 5.8 hours + 6s $\geq$ 12 min. *RC-Oracle* (resp. *TI-Oracle*) also supports real-time responses, i.e., it can construct the oracle in 0.4s (resp. 1.25s) and answer the *kNN* query and range query in both 7ms (resp. 14ms) for the P2P (resp. A2P) query on a point cloud with 10k points and 250 POIs.

6.3.5 ***Case study (solar storm).*** We conducted another case study on an evacuation simulation of Mars rovers due to the frequent solar storms [11], and Mars rovers need to find the shortest escape paths quickly from their current positions (which can be any position) on Mars to shelters or working stations (which are POIs) to avoid damage. The memory size of NASA's Mars 2020 rover is 256MB [10]. Our experimental results show for the A2P query on a point cloud with 250k points and 500 POIs, (1) the oracle construction time for (i) *TI-Oracle* is 25s, and (ii) *RC-Oracle-A2A* is 2,100 $\approx$ 35 min, (2) the oracle size for *TI-Oracle* is 28MB, and (ii) *RC-Oracle-A2A* is 10GB. Thus, *TI-Oracle* only is suitable since 28MB $\leq$ 256MB, but 10GB $\geq$ 256MB.

6.3.6 ***Summary.*** In terms of the oracle construction time, oracle size and shortest path query time, *RC-Oracle* is up to 975 times, 30 times and 6 times better than the best-known oracle *SE-Oracle-Adapt* for the P2P query on a point cloud, and *TI-Oracle* is up to 42,000 times, 10,800 times and 27 times better than the best-known oracle *EAR-Oracle-Adapt* for the A2A query on a point cloud. With the assistance of *RC-Oracle* (resp. *TI-Oracle*), our algorithm for both the *kNN* and range queries are up to 6 (resp. 27) times faster than *SE-Oracle-Adapt* (resp. *EAR-Oracle-Adapt*). For the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time, oracle size and all POIs *kNN* query time for *RC-Oracle* are 80s $\approx$ 1.3 min, 50MB and 12.5s, but the values are 78,000s $\approx$ 21.7 hours, 1.5GB and 150s for the best-known oracle *SE-Oracle-Adapt*. For the A2P query on a point cloud with 250k points and 500 POIs, the oracle construction time, oracle size and all POIs *kNN* query time for *TI-Oracle* are 25s, 28MB and 11s, but the values are 1,050,000s $\approx$ 12 days, 300GB and 300s for the best-known oracle *EAR-Oracle-Adapt*.

[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 [19, 57]) in Mount Rainier is 1 meter per 24 hours [56], 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].

7 CONCLUSION

In our paper, we propose six efficient $(1 + \epsilon)$-approximate shortest path oracle called *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2A*, *TI-Oracle* and *TI-Oracle-A2A* for the P2P, A2P and A2A queries on a point cloud, which have good performances (in terms of the oracle construction time, oracle size and shortest path query time) compared with the best-known oracle. With their assistance, we propose algorithms for answering other proximity queries, i.e., the *kNN* and range queries. For the future work, we can explore how to simplify the point cloud to further reduce the oracle construction time of these oracles.

REFERENCES

[1] 2022. *Blender.* https://www.blender.org
[2] 2022. *Google Earth.* https://earth.google.com/web
[3] 2022. *Robinson Mountain.* https://www.mountaineers.org/activities/routes-places/robinson-mountain
[4] 2023. *Cyberpunk 2077.* https://www.cyberpunk.net
[5] 2023. *Data Geocomm.* http://data.geocomm.com/
[6] 2023. *Google Map.* https://www.google.com/maps
[7] 2023. *Gunnison National Forest.* https://gunnisoncrestedbutte.com/visit/places-to-go/parks-and-outdoors/gunnison-national-forest/
[8] 2023. *Laramie Mountain.* https://www.britannica.com/place/Laramie-Mountains
[9] 2023. *Preferred walking speed.* https://en.wikipedia.org/wiki/Preferred_walking_speed
[10] 2024. *Mars 2020 mission perseverance rover brains.* https://mars.nasa.gov/mars2020/spacecraft/rover/brains/
[11] 2024. *NASA's MAVEN Observes Martian Light Show Caused by Major Solar Storm.* https://www.nasa.gov/missions/nasas-maven-observes-martian-light-show-caused-by-major-solar-storm/
[12] 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
[13] Anonymous and Anonymous. 2024. Efficient path oracles for proximity queries on point clouds (technical report). https://anonymous.4open.science/r/PointCloudOracle-xxxx/TechnicalReport.pdf
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG.* New York, NY, USA, 360–369.
[19] 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
[20] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.
[21] 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.
[22] Mark De Berg. 2000. *Computational geometry: algorithms and applications.* Springer Science & Business Media.
[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 (SIGMOD)* 1, 1, 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, 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, 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 ACM on Management of Data (SIGMOD).* 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] Niall McCarthy. 2021. *Exploring the red planet is a costly undertaking.* https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

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

[49] 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

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

[51] 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.

[52] 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.

[53] 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).

[54] National Park Service. 2022. *Mount Rainier.* https://www.nps.gov/mora/index.htm
[55] National Park Service. 2022. *Mount Rainier Annual Snowfall Totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm
[56] National Park Service. 2022. *Mount Rainier Frequently Asked Questionss.* https://www.nps.gov/mora/faqs.htm
[57] National Weather Service. 2023. *Measuring Snow.* https://www.weather.gov/dvn/snowmeasure
[58] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. *Proceedings of the VLDB Endowment* 1, 1, 1020–1031.
[59] 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.
[60] 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).
[61] 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
[62] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *Proceedings of the ACM on Management of Data (SIGMOD).* 1211–1226.
[63] 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).
[64] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2024. On Efficient Shortest Path Computation on Terrain Surface: A Direction-Oriented Approach. *IEEE Transactions on Knowledge & Data Engineering (TKDE)* 1 (2024), 1–14.
[65] 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.
[66] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment* 2, 1, 1114–1125.
[67] 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.
[68] 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, 2683–2686.
[69] Yinzhao Yan and Raymond Chi-Wing Wong. 2024. Proximity queries on point clouds using rapid construction path oracle. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 1, 1–26.
[70] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering* (2014).

A SUMMARY OF ALL NOTATIONS

Table 3 shows a summary of all notations.

B AR2P AND AR2AR QUERIES ON *TINS*

Apart from the P2P query on *TIN*s that we discussed in the main body of this paper, we also present oracles to answer AR2P and AR2AR queries on *TIN*s (such that the arbitrary point may lie on the faces of the *TIN*) based on *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A*, *TI-Oracle* and *TI-Oracle-A2A*. These oracles are *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *RC-Oracle-Adapt-AR2AR*, *TI-Oracle-Adapt* and *TI-Oracle-Adapt-AR2AR*. These adaptions are similar, so we use *RC-Oracle-Adapt-AR2AR* as an example.

For *RC-Oracle-Adapt-AR2AR*, there are two differences between *RC-Oracle-A2A*. The first difference is that we need to calculate the shortest path passing on a conceptual graph of a *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 that have the same coordinate values as all vertices in the *TIN*, *RC-Oracle-Adapt-AR2AR* can answer the AR2AR query. (2) For the second case, we denote the face that s lies in to be f_s and the face that t lies in to be f_t. We denote the set of three vertices of f_s to be V_s, and the set of three vertices of f_t to be V_t. After creating POIs that have the same coordinate values as all vertices in the *TIN*, we

Table 3. Summary of all notations

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
$N(p)$	A set of neighbor points of p
$d_E(p, p')$	The Euclidean distance between point p and p'
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
M_{path}	The path map table
M_{end}	The endpoint map table
P_{remain}	A set of remaining POIs of P on C that we have not used algorithm *FastFly* to calculate the exact shortest path passing on C with $p_i \in P_{remain}$ as source
$P_{dest}(q)$	A set of POIs of P on C that we need to use algorithm *FastFly* to calculate the exact shortest path passing on C from q to $p_i \in P_{dest}(q)$ as destinations
M_{cont}	The containing point map table
M_{boun}	The boundary point map table
M_{belo}	The belonging point map table
$M_{intra\text{-}path}$	The intra-path map table
$M_{inter\text{-}path}$	The inter-path map table
$M_{inter\text{-}end}$	The inter-endpoint map table
T	The *TIN* constructed by C
h	The height of the compressed partition tree
β	The largest capacity dimension
θ	The minimum inner angle of any face in T
l_{max}/l_{min}	The length of the longest / shortest edge of T
λ	The number of highway nodes covered by a minimum square
ξ	The square root of the number of boxes
m	The number of Steiner points per face
$\Pi^*(s, t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s, t\|C)\|$	The distance of $\Pi^*(s, t\|C)$
$\Pi_A(s, t\|C)$	The shortest path passing on C between s and t returned by oracle A, where $A \in \{$*RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A*, *TI-Oracle*, *TI-Oracle-A2A*$\}$
$\Pi^*(s, t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_N(s, t\|T)$	The shortest network path passing on T between s and t
$\Pi_E(s, t\|T)$	The shortest path passing on the edges of T between s and t where these edges belongs to the faces that $\Pi^*(s, t\|T)$ passes

need to find the shortest path between each vertex $u \in V_s$ and each vertex $v \in V_t$, then concatenate the line segment (s, u) and (v, t) with the path. After calculating nine paths, we select the path with the smallest distance as the result path. (3) For the third case, it is similar to the second case. When s lies on the vertices of the *TIN* and t lies on the faces of the *TIN*, we set $V_s = \{s\}$. When t lies on the vertices of the *TIN* and s lies on the faces of the *TIN*, we set $V_t = \{t\}$. Then, we can use the second case to answer the shortest path between s and t.

All the theoretical analysis of *RC-Oracle-Adapt-AR2AR* is the same as *RC-Oracle-A2A*. We mainly discuss the error bound. For the AR2AR query on *TIN*s, *RC-Oracle-Adapt-AR2AR* always has $|\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pairs of vertices s and t on the faces T, where $\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)$ is the calculated shortest path of *RC-Oracle-Adapt-AR2AR* passing on a conceptual graph of T between s and t, where the vertices of this conceptual graph are formed by the vertices of T, and the edges of this graph are formed by adding edges between each vertex and its 8 neighbor vertices, and $\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)$ is the exact shortest path passing on this conceptual graph between s and t. This is because we let $p \in V_s$ and $q \in V_t$ be two vertices that lie on the path $\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)$, so $|\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)| = |(s,p)| + |\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(p,q|T)| + |(q,t)| \leq |(s,p')| + |\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| + |(q',t)|$. We let $p' \in V_s$ and $q' \in V_t$ be two vertices that lie on the path $\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)$, so $|\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)| = |(s,p')| + |\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| + |(q',t)|$. Since *RC-Oracle-Adapt-AR2AR* always has $|\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| \leq (1+\epsilon)|\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)|$, we obtain $|\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)| = |(s,p)| + |\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(p,q|T)| + |(q,t)| \leq |(s,p')| + |\Pi_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| + |(q',t)| \leq |(s,p')| + (1+\epsilon)|\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| + |(q',t)| \leq (1+\epsilon)|(s,p')| + (1+\epsilon)|\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(p',q'|T)| + (1+\epsilon)|(q',t)| = (1+\epsilon)|\Pi^*_{\textit{RC-Oracle-Adapt-AR2AR}}(s,t|T)|$.

C EMPIRICAL STUDIES

C.1 Experimental Results for *TIN*s

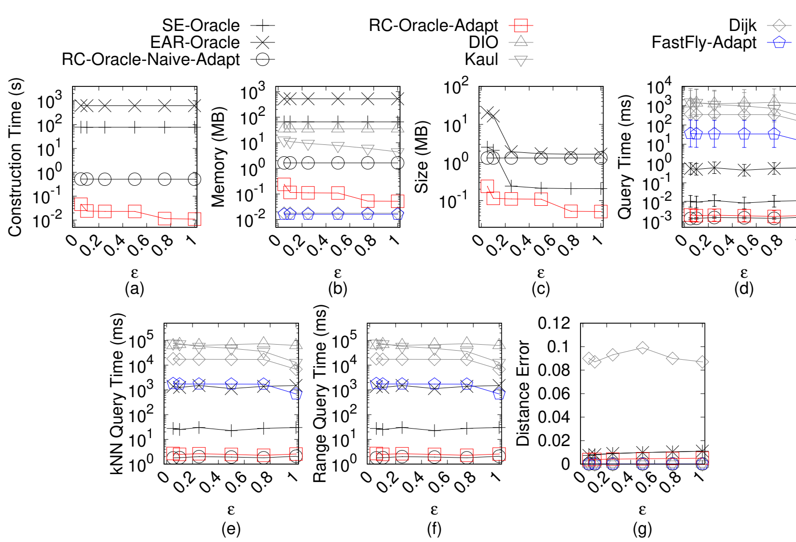

Fig. 21. Baseline comparisons (effect of ϵ on BH_t*-small TIN* dataset for the P2P query)

C.1.1 ***Baseline comparisons for the P2P query***. We study the P2P query on *TIN*s. We (1) compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *DIO*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle-Adapt*, *DIO*, *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. (1) Figure 21 and Figure 22 show the result on BH_t*-small TIN* dataset for the P2P query when varying ϵ and n, respectively. (2) Figure 23, Figure 24 and Figure 25

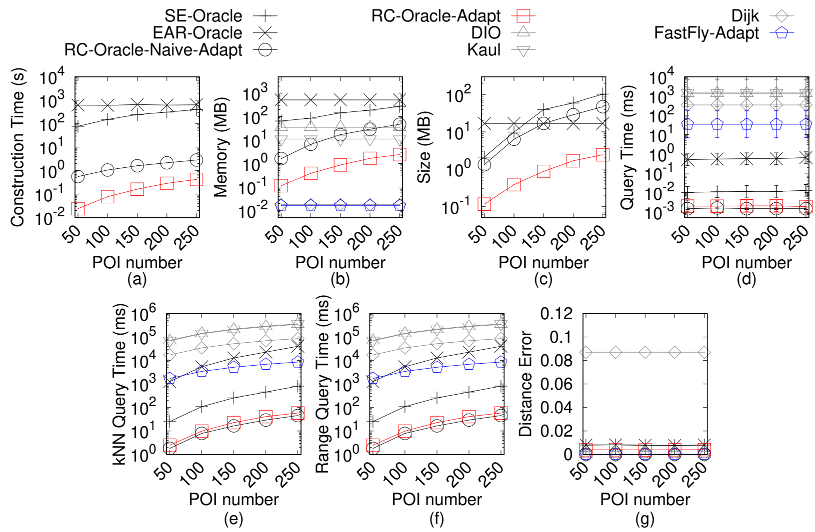

Fig. 22. Baseline comparisons (effect of n on BH_t*-small TIN* dataset for the P2P query)

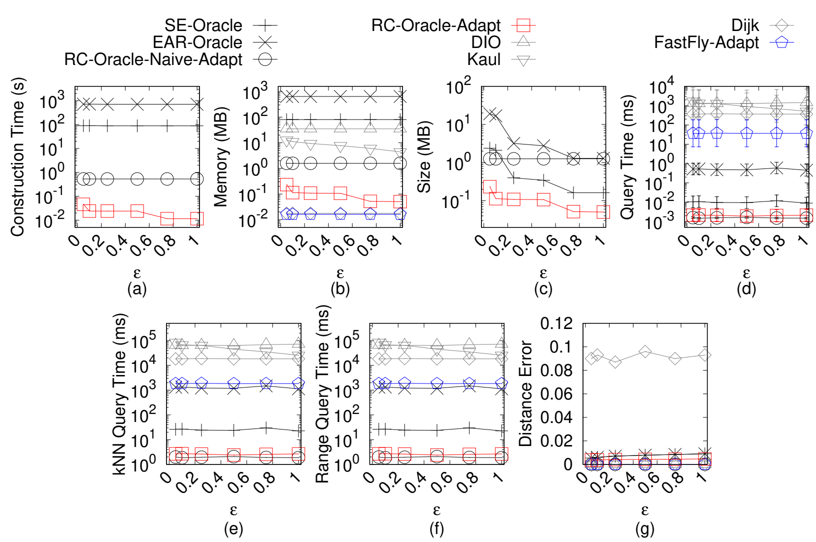

Fig. 23. Baseline comparisons (effect of ϵ on EP_t*-small TIN* dataset for the P2P query)

show the result on EP_t*-small TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (3) Figure 26 and Figure 27 show the result on GF_t*-small TIN* dataset for the P2P query when varying ϵ and n, respectively. (4) Figure 28 and Figure 29 show the result on LM_t*-small TIN* dataset for the P2P query when varying ϵ and n, respectively. (5) Figure 30 and Figure 31 show the result on RM_t*-small TIN* dataset for the P2P query when varying ϵ and n, respectively. (6) Figure 32, Figure 33 and Figure 34 show the result on BH_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (6) Figure 32, Figure 33 and Figure 34 show the result on BH_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (7) Figure 35, Figure 36 and Figure 37 show the result on EP_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (8) Figure 38,

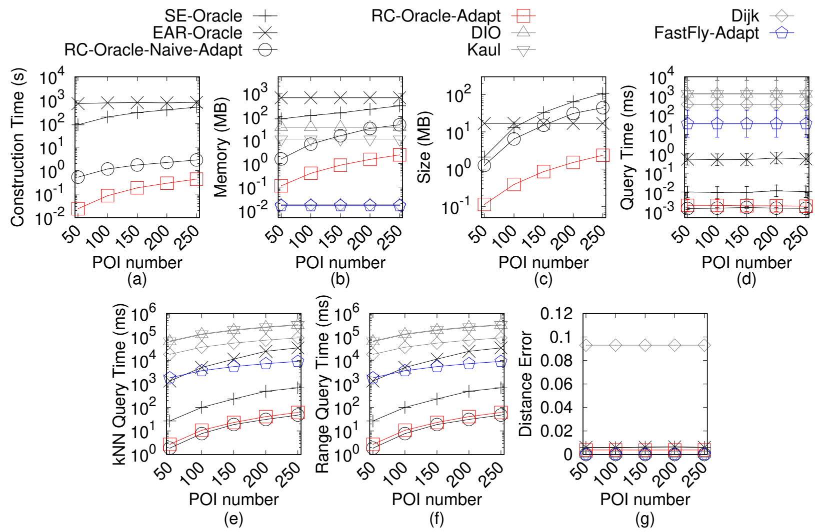

Fig. 24. Baseline comparisons (effect of n on EP_t*-small TIN* dataset for the P2P query)

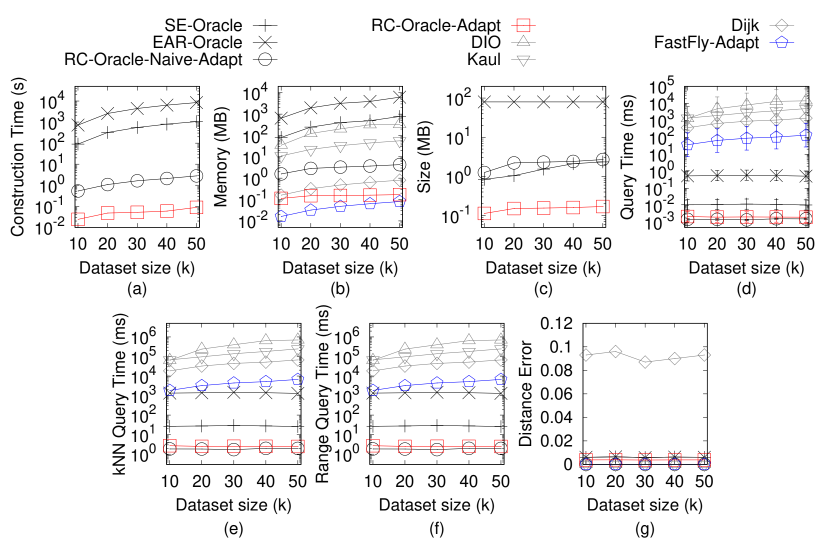

Fig. 25. Baseline comparisons (effect of N on EP_t*-small TIN* dataset for the P2P query)

Figure 39 and Figure 40 show the result on GF_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (9) Figure 41, Figure 42 and Figure 43 show the result on LM_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively. (10) Figure 44, Figure 45 and Figure 46 show the result on RM_t *TIN* dataset for the P2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 21, Figure 23, Figure 26, Figure 28 and Figure 30, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting N to be 10k and n to be 50. In Figure 32, Figure 35, Figure 38, Figure 41 and Figure 44, 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,

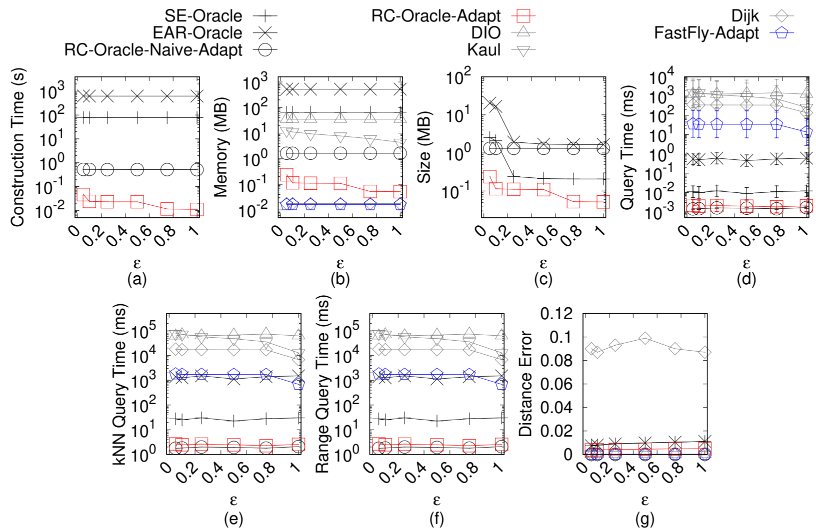

Fig. 26. Baseline comparisons (effect of ϵ on GF_t-*small TIN* dataset for the P2P query)

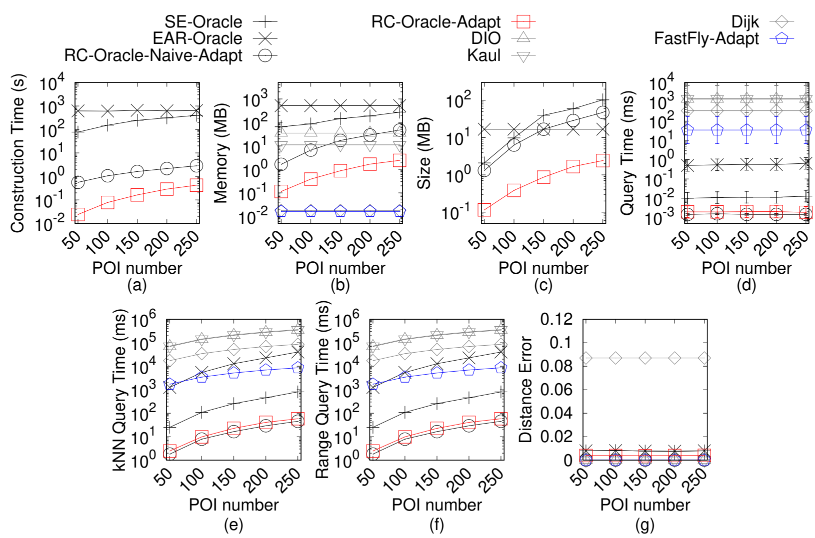

Fig. 27. Baseline comparisons (effect of n on GF_t-*small TIN* dataset for the P2P query)

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

Effect of n. In Figure 22, Figure 24, Figure 27, Figure 29 and Figure 31, 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 33, Figure 36, Figure 39, Figure 42 and Figure 45, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time

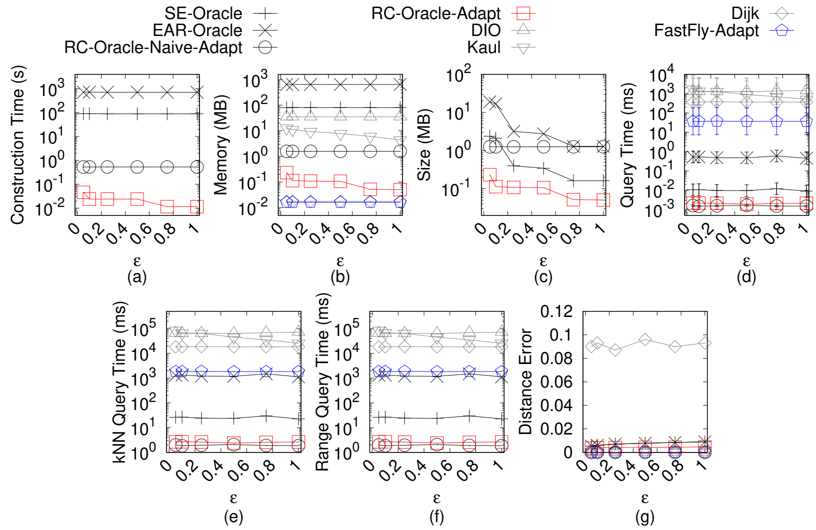

Fig. 28. Baseline comparisons (effect of ϵ on LM_t*-small TIN* dataset for the P2P query)

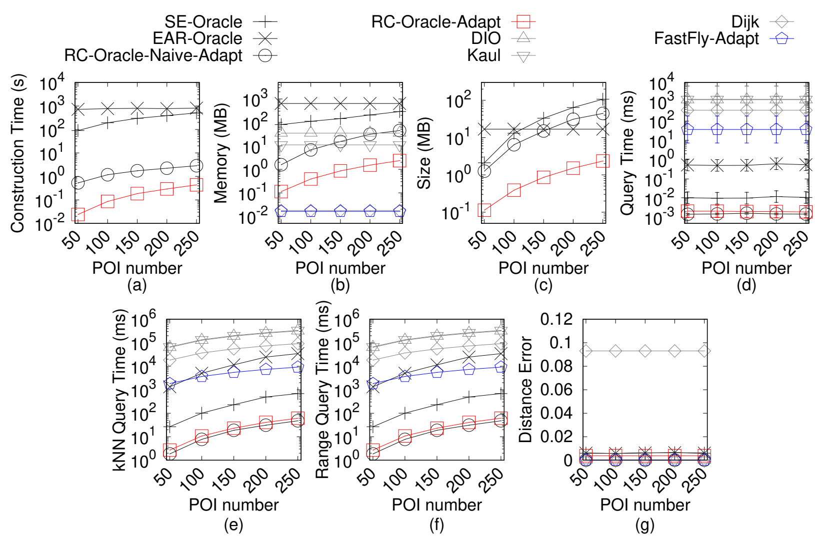

Fig. 29. Baseline comparisons (effect of n on LM_t*-small TIN* dataset for the P2P query)

for *SE-Oracle* is large compared with *RC-Oracle-Adapt*, which shows the superior performance of *RC-Oracle-Adapt* in terms of the oracle construction and shortest path querying.

Effect of N (scalability test). In Figure 25, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_t*-small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 34, Figure 37, Figure 40, Figure 43 and Figure 46, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle-Adapt* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size and shortest path query time. The shortest path query

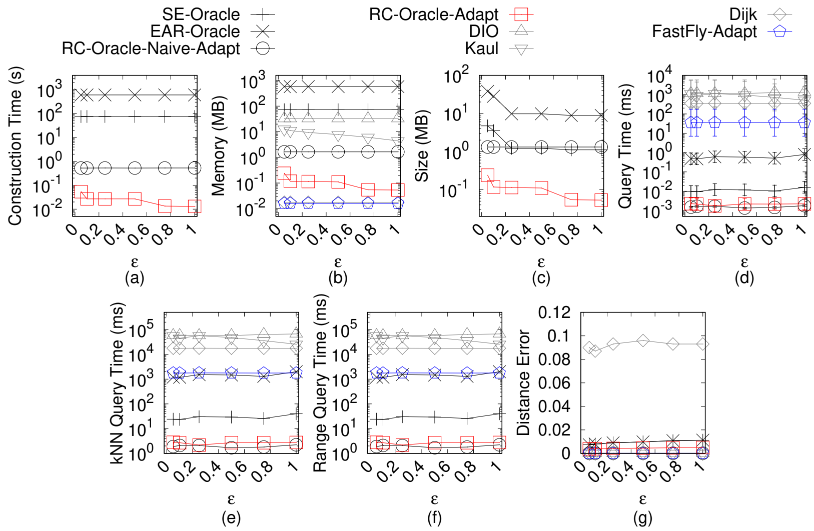

Fig. 30. Baseline comparisons (effect of ϵ on RM_t*-small TIN* dataset for the P2P query)

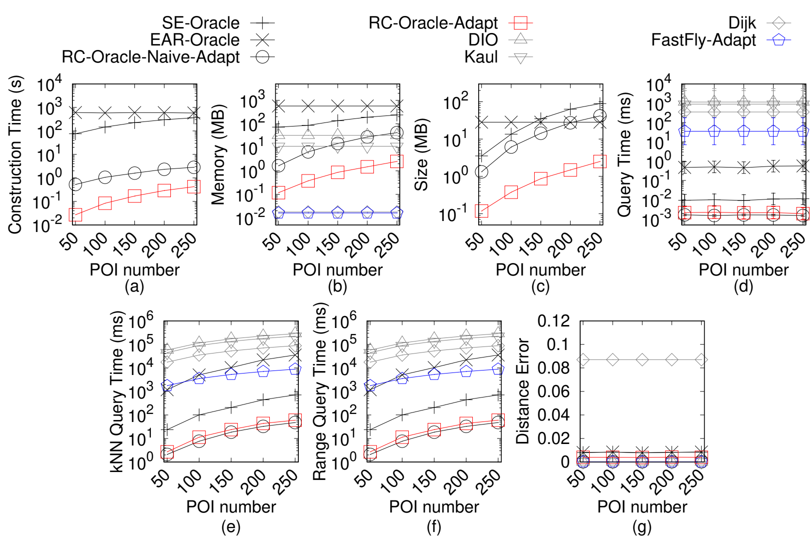

Fig. 31. Baseline comparisons (effect of n on RM_t*-small TIN* dataset for the P2P query)

time of *FastFly-Adapt* is 100 times smaller than that of *DIO*, and the distance error of *FastFly-Adapt* (with distance error close to 0) is much smaller than that of *Dijk* (with distance error 0.03).

C.1.2 ***Baseline comparisons for the AR2P query***. We study the AR2P query on *TIN*s. We (1) compared *SE-Oracle-AR2AR*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR* 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

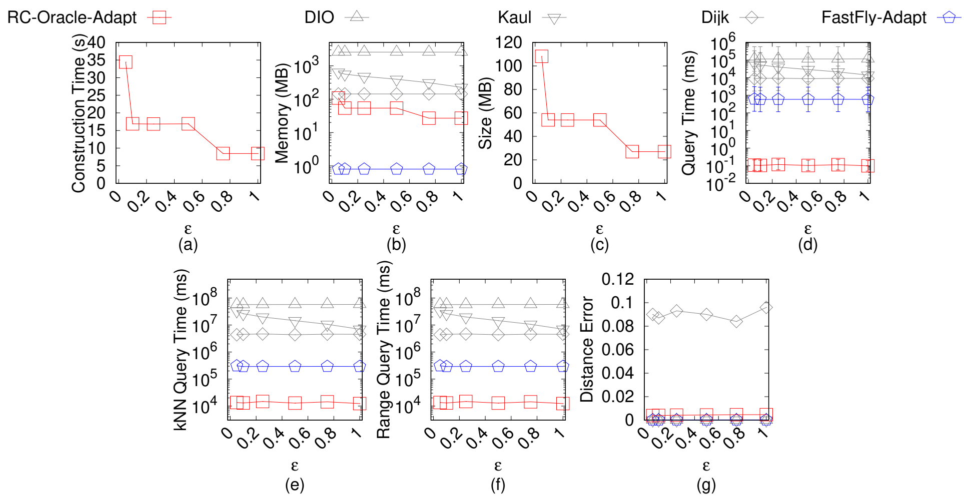

Fig. 32. Baseline comparisons (effect of ϵ on BH_t *TIN* dataset for the P2P query)

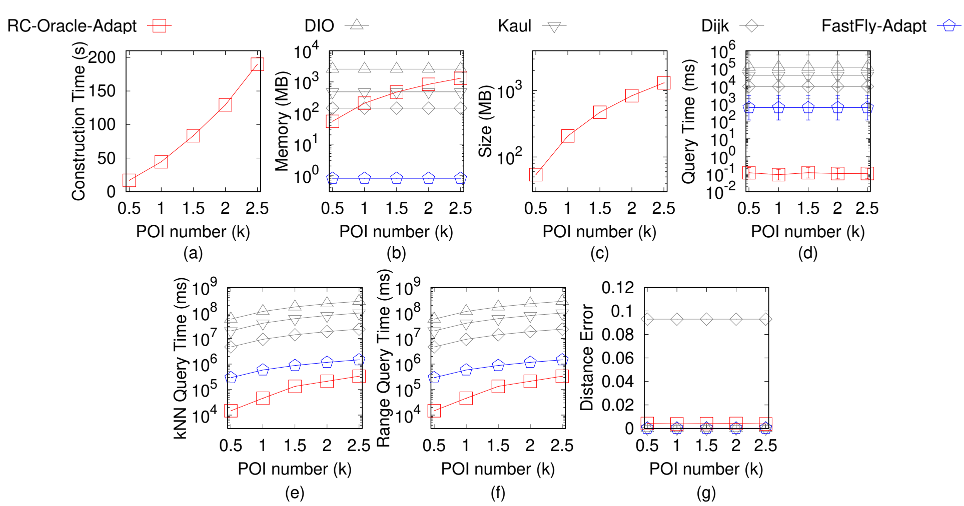

Fig. 33. Baseline comparisons (effect of n on BH_t *TIN* dataset for the P2P query)

error is very small), so their results are omitted. (1) Figure 47 and Figure 48 show the result on BH_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (2) Figure 49, Figure 50 and Figure 51 show the result on EP_t-*small TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (3) Figure 52 and Figure 53 show the result on GF_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (4) Figure 54 and Figure 55 show the result on LM_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (5) Figure 56 and Figure 57 show the result on RM_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (6) Figure 58, Figure 59 and Figure 60 show the result on BH_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (6) Figure 58, Figure 59 and Figure 60 show the result on BH_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (7) Figure 61,

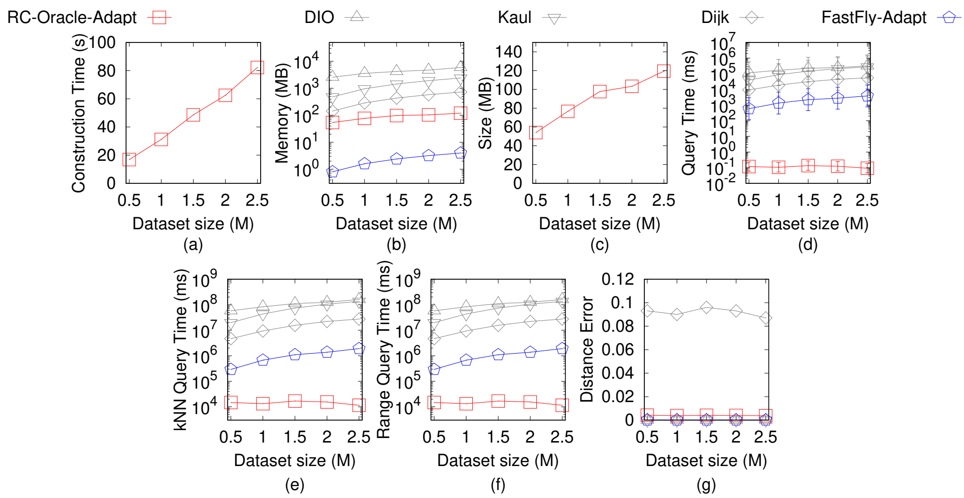

Fig. 34. Baseline comparisons (effect of N on BH_t *TIN* dataset for the P2P query)

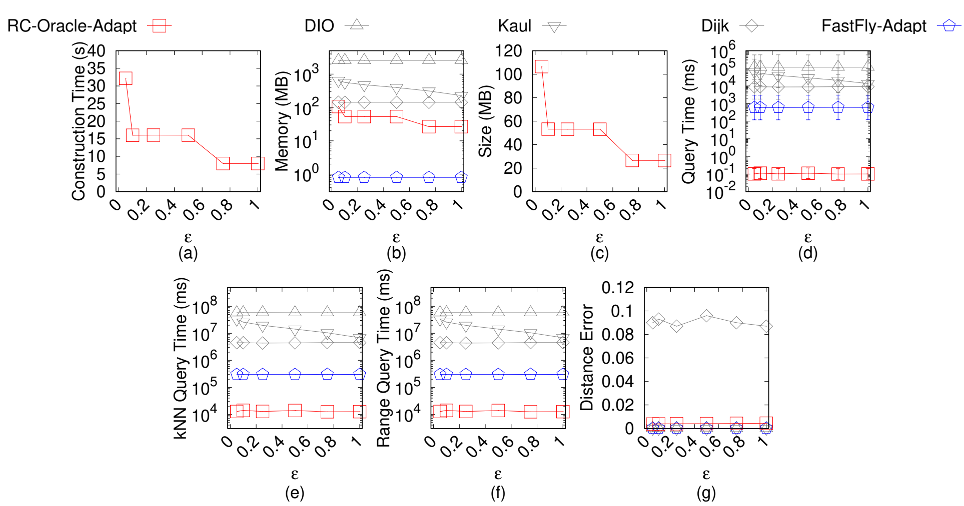

Fig. 35. Baseline comparisons (effect of ϵ on EP_t *TIN* dataset for the P2P query)

Figure 62 and Figure 63 show the result on EP_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (8) Figure 64, Figure 65 and Figure 66 show the result on GF_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (9) Figure 67, Figure 68 and Figure 69 show the result on LM_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (10) Figure 70, Figure 71 and Figure 72 show the result on RM_t *TIN* dataset for the AR2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 47, Figure 49, Figure 52, Figure 54 and Figure 56, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting N to be 10k and n to be 50. In Figure 58, Figure 61, Figure 64, Figure 67 and Figure 70, 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

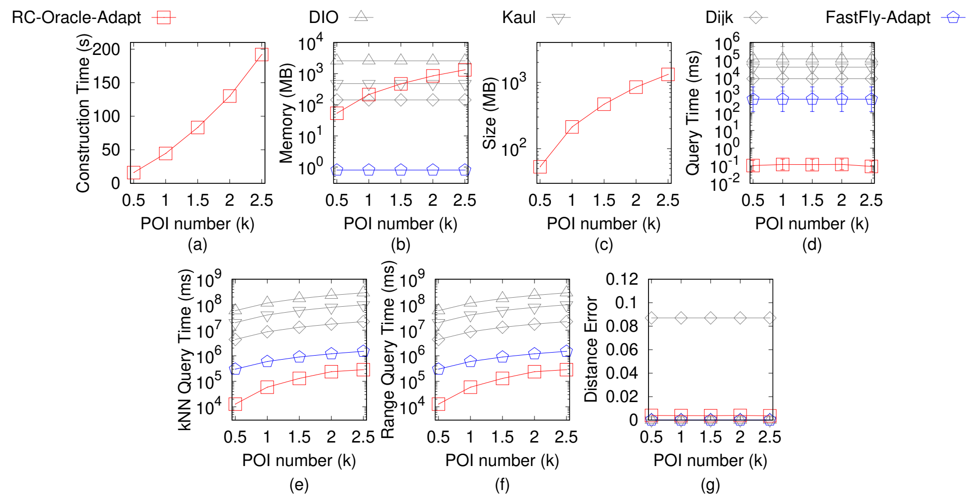

Fig. 36. Baseline comparisons (effect of n on EP_t *TIN* dataset for the P2P query)

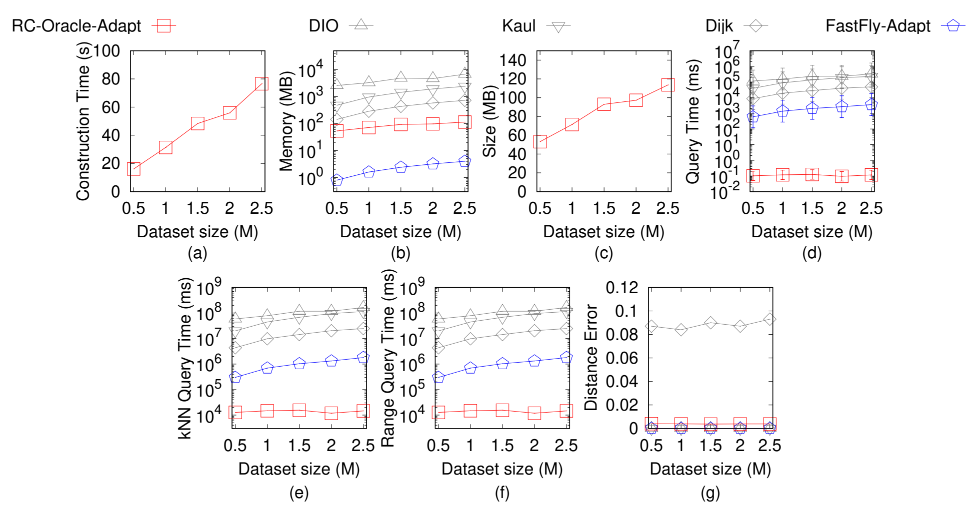

Fig. 37. Baseline comparisons (effect of N on EP_t *TIN* dataset for the P2P query)

dataset by setting N to be 0.5M and n to be 500. The oracle construction time, memory usage and oracle size of *RC-Oracle-Adapt-AR2P-SmCon* is the smallest in all oracles since it has the same oracle construction process as of *RC-Oracle-Adapt*, but its shortest path query time is larger than other oracles (but still smaller than *FastFly-Adapt*) since it can terminate earlier when using *FastFly-Adapt* in the shortest path query phase. Thus, it performs well in the case of fewer proximity queries. The oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* are also very small and their shortest path query time are also very small due to their earlier termination during oracle construction and tight information stored in the oracles.

Effect of n. In Figure 48, Figure 50, Figure 53, Figure 55 and Figure 57, 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

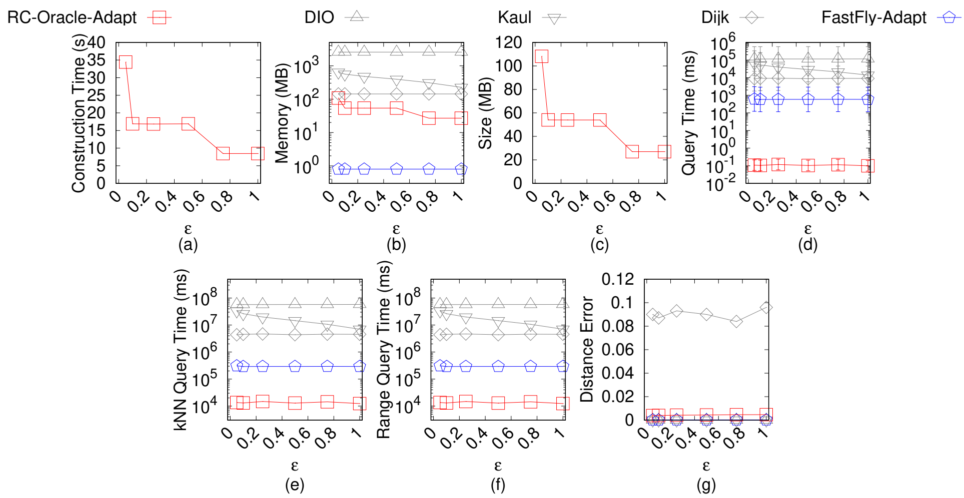

Fig. 38. Baseline comparisons (effect of ϵ on GF_t *TIN* dataset for the P2P query)

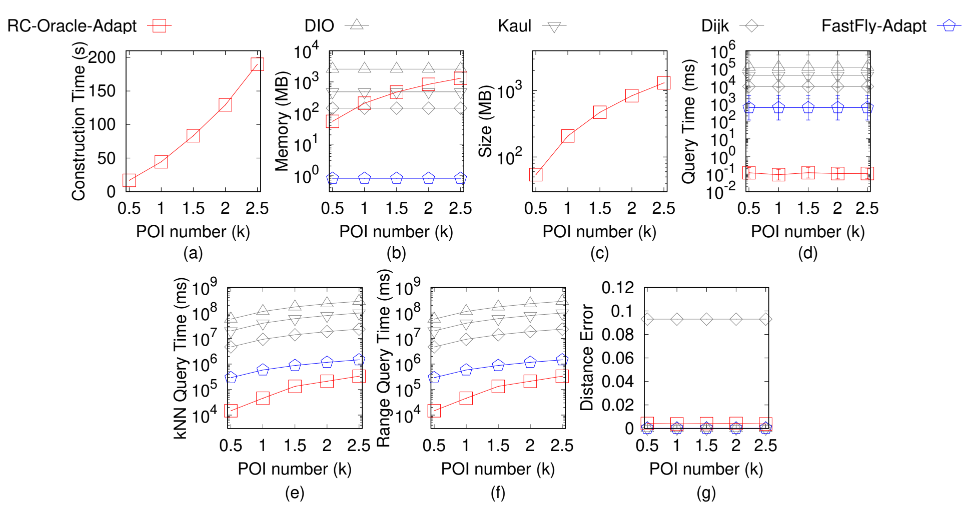

Fig. 39. Baseline comparisons (effect of n on GF_t *TIN* dataset for the P2P query)

by setting N to be 10k and ϵ to be 0.1. In Figure 59, Figure 62, Figure 65, Figure 68 and Figure 71, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and ϵ to be 0.25. When $n < 100$ (resp. $n \geq 100$), the oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle-Adapt*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).

Effect of N (scalability test). In Figure 51, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_t*-small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 60, Figure 63, Figure 66, Figure 69 and Figure 72, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t*-small*, EP_t*-small*, GF_t*-small*, LM_t*-small* and RM_t*-small* dataset by setting n to be 500

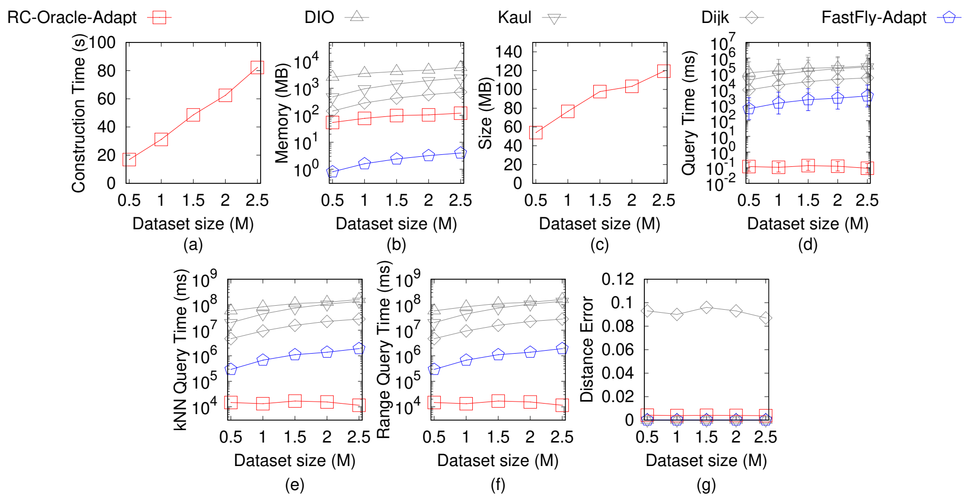

Fig. 40. Baseline comparisons (effect of N on GF_t *TIN* dataset for the P2P query)

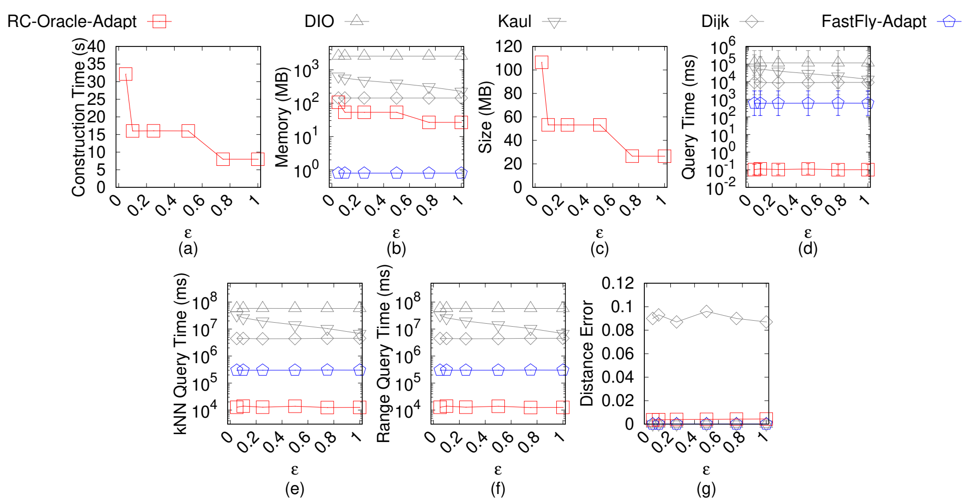

Fig. 41. Baseline comparisons (effect of ϵ on LM_t *TIN* dataset for the P2P query)

and ϵ to be 0.25 for scalability test. The oracle construction time of *RC-Oracle-Adapt-A2P-SmCon*, *RC-Oracle-Adapt-A2P-SmQue* and *TI-Oracle-Adapt* are only 80s ≈ 1.3 min, 310s ≈ 5.1 min and 250s ≈ 4.1min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of them.

C.1.3 ***Baseline comparisons for the AR2AR query.*** We study the AR2AR query on *TIN*s. In the same figures of baseline comparisons for the AR2P query on *TIN*s, we compared *SE-Oracle-AR2AR*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR* and *TI-Oracle-Adapt-AR2AR*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time due to their earlier termination during oracle construction and tight information stored in the oracles.

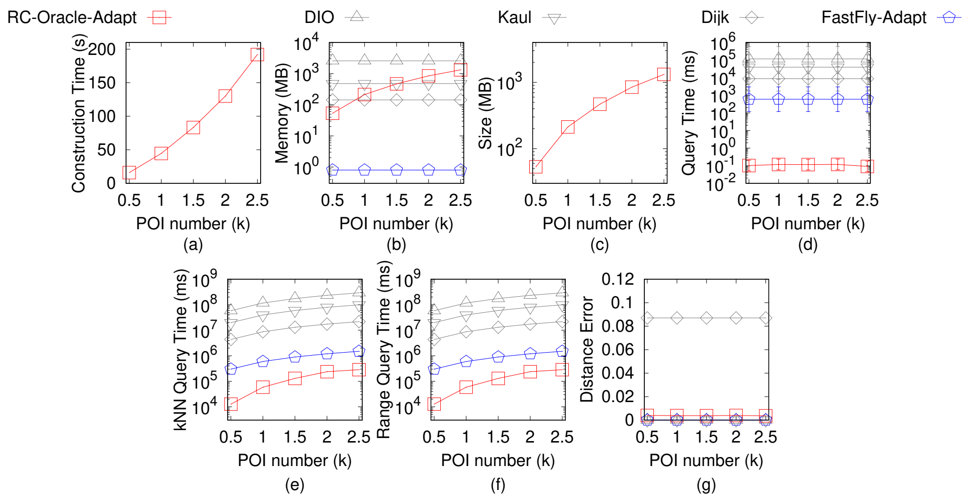

Fig. 42. Baseline comparisons (effect of n on LM_t *TIN* dataset for the P2P query)

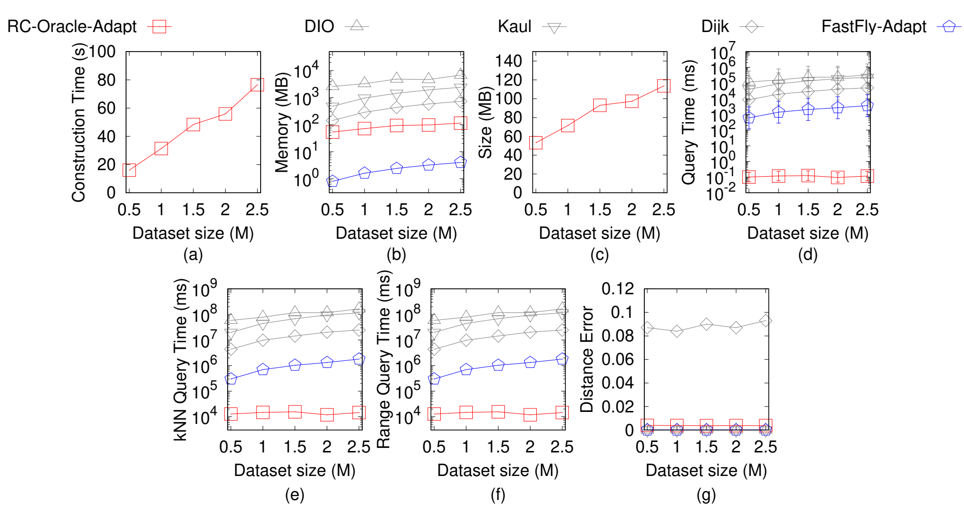

Fig. 43. Baseline comparisons (effect of N on LM_t *TIN* dataset for the P2P query)

C.2 Experimental Results for Point Clouds

C.2.1 ***Baseline comparisons for the P2P query.*** We study the P2P query on *point clouds* for baseline comparisons. We (1) compared *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *DIO-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle*, *DIO-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets with default 500 POIs. (1) Figure 73 and Figure 74 show the result on BH_p*-small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (2) Figure 75, Figure 76 and Figure 77 show the result on EP_p*-small* point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (3) Figure 78 and Figure 79 show the result on GF_p*-small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (4) Figure 80 and Figure 81 show the result on

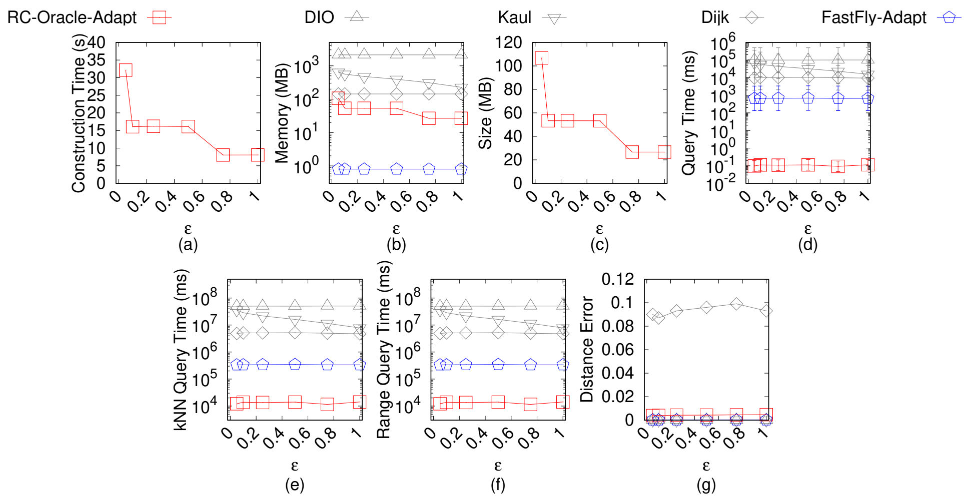

Fig. 44. Baseline comparisons (effect of ϵ on RM_t *TIN* dataset for the P2P query)

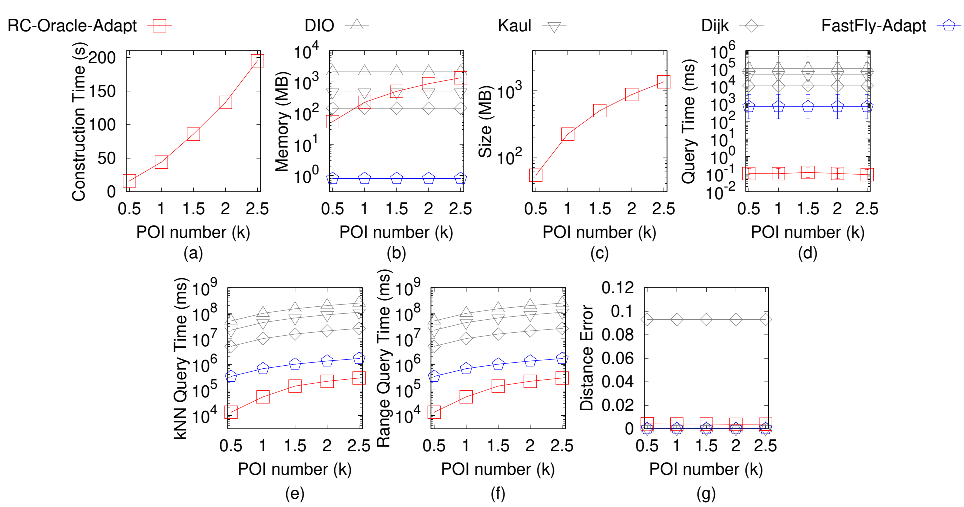

Fig. 45. Baseline comparisons (effect of n on RM_t *TIN* dataset for the P2P query)

LM_p*-small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (5) Figure 82 and Figure 83 show the result on RM_p*-small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (6) Figure 84, Figure 85 and Figure 86 show the result on BH_p point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (6) Figure 84, Figure 85 and Figure 86 show the result on BH_p point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (7) Figure 87, Figure 88 and Figure 89 show the result on EP_p point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (8) Figure 90, Figure 91 and Figure 92 show the result on GF_p point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (9) Figure 93, Figure 94 and Figure 95 show the result on LM_p point cloud dataset for the P2P query when varying

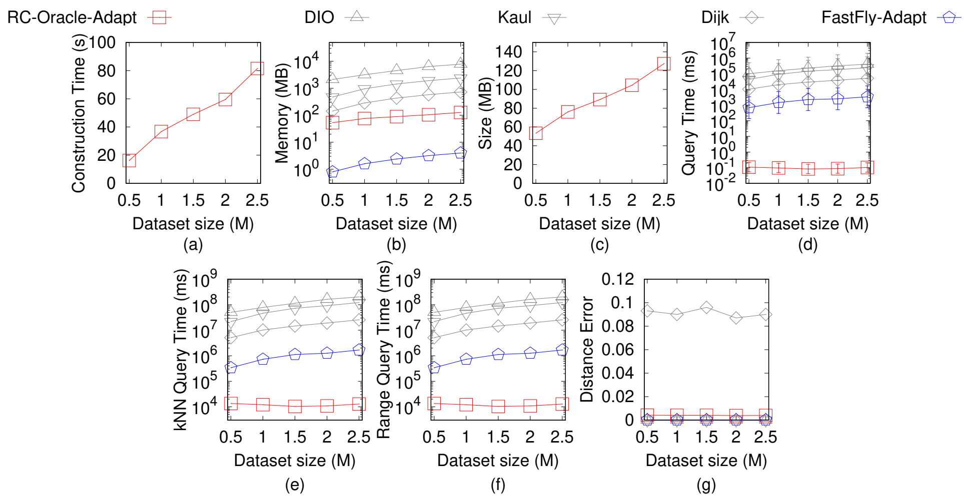

Fig. 46. Baseline comparisons (effect of N on RM_t *TIN* dataset for the P2P query)

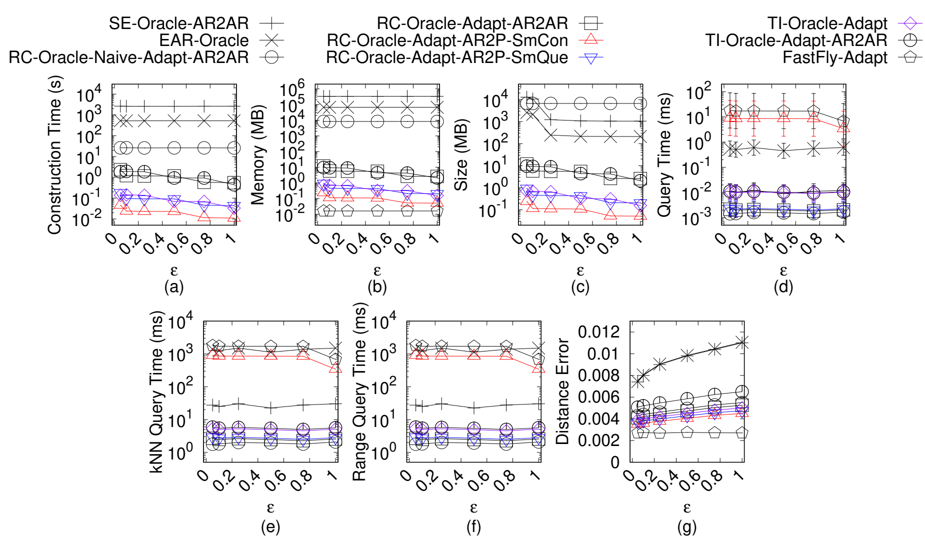

Fig. 47. Baseline comparisons (effect of ϵ on BH_t*-small TIN* dataset for the AR2P query)

ϵ, n and N, respectively. (10) Figure 96, Figure 97 and Figure 98 show the result on RM_p point cloud dataset for the P2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 73, Figure 75, Figure 78, Figure 80 and Figure 82, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p*-small*, EP_p*-small*, GF_p*-small*, LM_p*-small* and RM_p*-small* dataset by setting N to be 10k and n to be 50. In Figure 84, Figure 87, Figure 90, Figure 93 and Figure 96, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle* a lot, the oracle construction time, memory consumption, oracle size, shortest path query time, all POIs *kNN* query time and all POIs range query time of *RC-Oracle* still perform much better than the best-known oracle *SE-Oracle-Adapt*, and other algorithms / oracles.

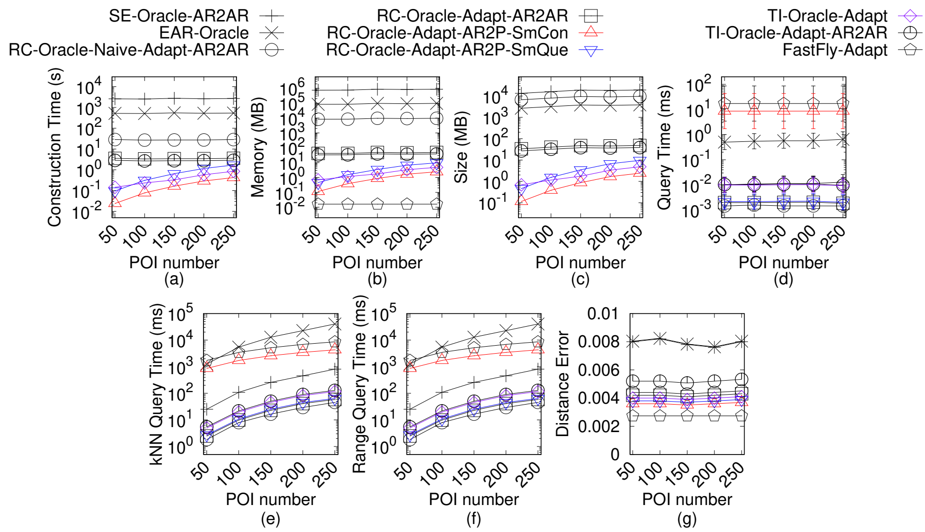

Fig. 48. Baseline comparisons (effect of n on BH_t-*small TIN* dataset for the AR2P query)

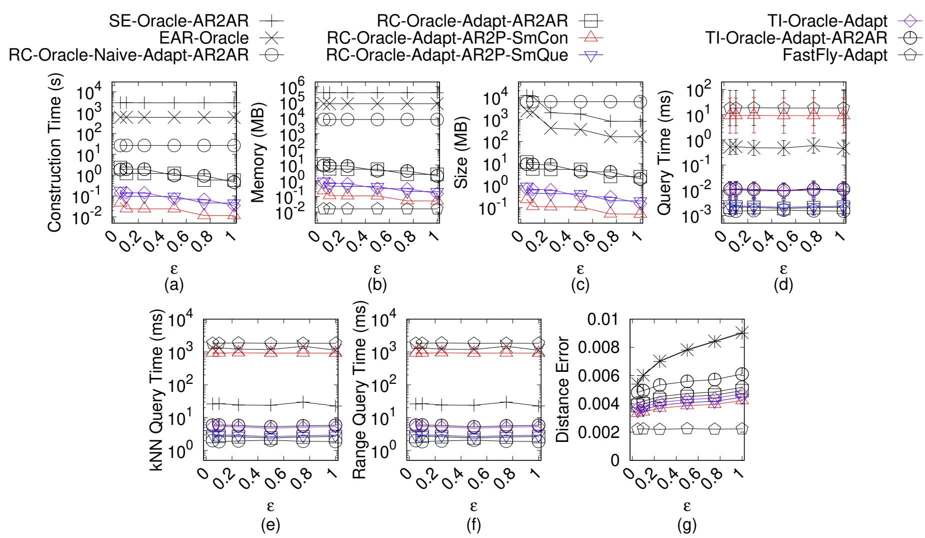

Fig. 49. Baseline comparisons (effect of ϵ on EP_t-*small TIN* dataset for the AR2P query)

Effect of n. In Figure 74, Figure 76, Figure 79, Figure 81 and Figure 83, 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 85, Figure 88, Figure 91, Figure 94 and Figure 97, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time for *SE-Oracle* is large compared with *RC-Oracle*, which shows the superior performance of *RC-Oracle* in terms of the oracle construction and shortest path query.

Effect of N (scalability test). In Figure 77, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_p-*small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 86,

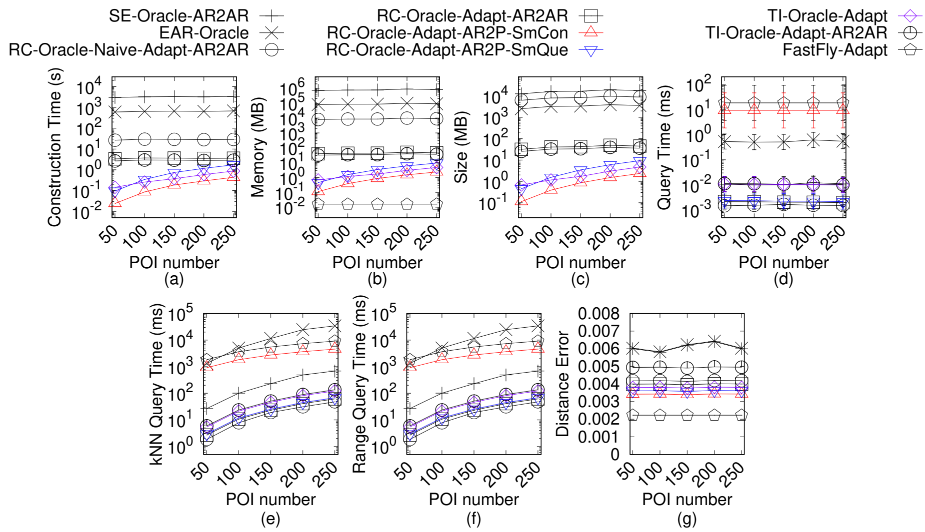

Fig. 50. Baseline comparisons (effect of n on EP_t*-small TIN* dataset for the AR2P query)

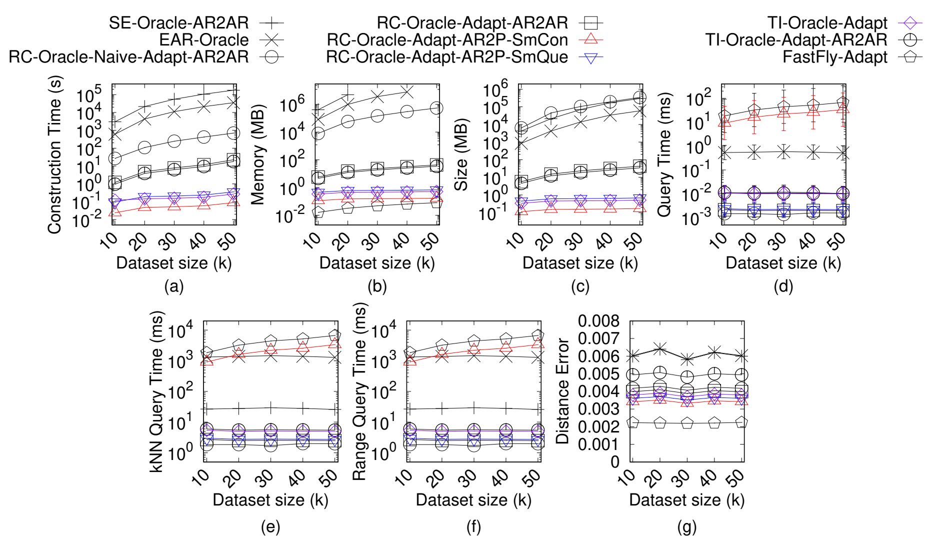

Fig. 51. Baseline comparisons (effect of N on EP_t*-small TIN* dataset for the AR2P query)

Figure 89, Figure 92, Figure 95 and Figure 98, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_p*-small*, EP_p*-small*, GF_p*-small*, LM_p*-small* and RM_p*-small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size and shortest path query time.

C.2.2 ***Ablation study for the P2P query***. We study the P2P query on *point clouds* for ablation study. We compared *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle*.

In Figure 99, Figure 100, Figure 101, Figure 102 and Figure 103, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size and shortest path query time of *RC-Oracle* perform

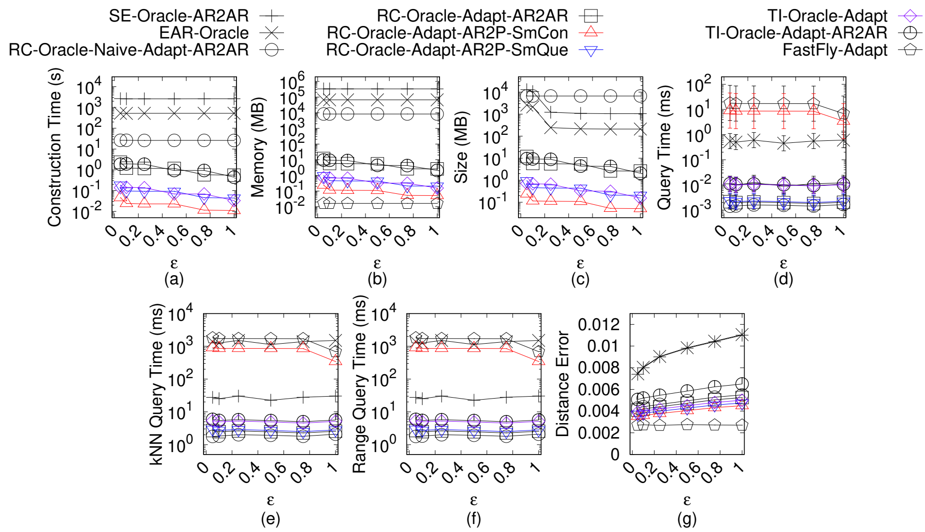

Fig. 52. Baseline comparisons (effect of ϵ on GF_t*-small TIN* dataset for the AR2P query)

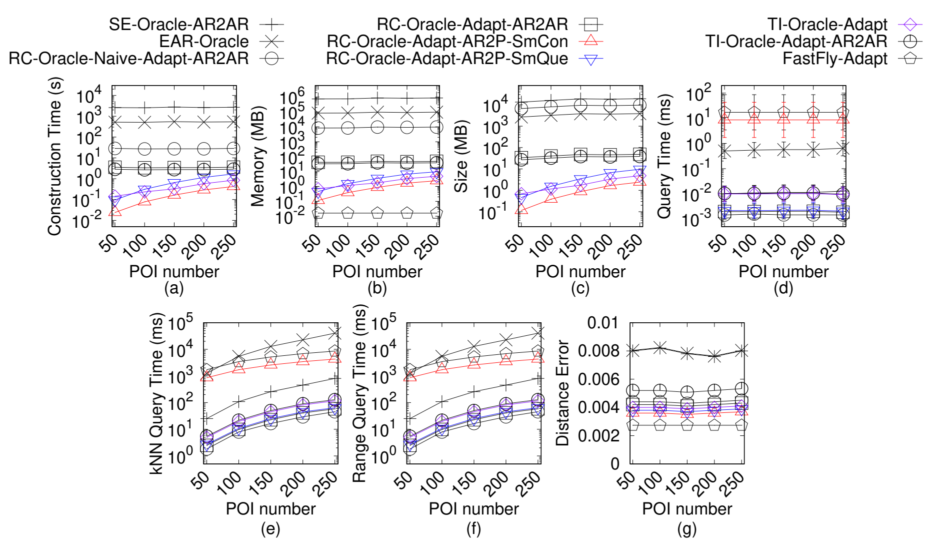

Fig. 53. Baseline comparisons (effect of n on GF_t*-small TIN* dataset for the AR2P query)

better than *SE-Oracle-FastFly-Adapt* and *EAR-Oracle-FastFly-Adapt*, which shows the usefulness of the oracle part of *RC-Oracle*.

C.2.3 ***Comparisons with other proximity queries oracles and variation oracles for the P2P query***. We study the P2P query on *point clouds* for comparisons with other proximity queries oracles and variation oracles. We denote *RC-Oracle-NaiveProx* to be *RC-Oracle* using the naive proximity query algorithm. We compared *SU-Oracle-Adapt*, *RC-Oracle-NaiveProx* and *RC-Oracle*.

In Figure 104, Figure 105, Figure 106, Figure 107 and Figure 108, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size and *kNN* query time of *RC-Oracle* perform better

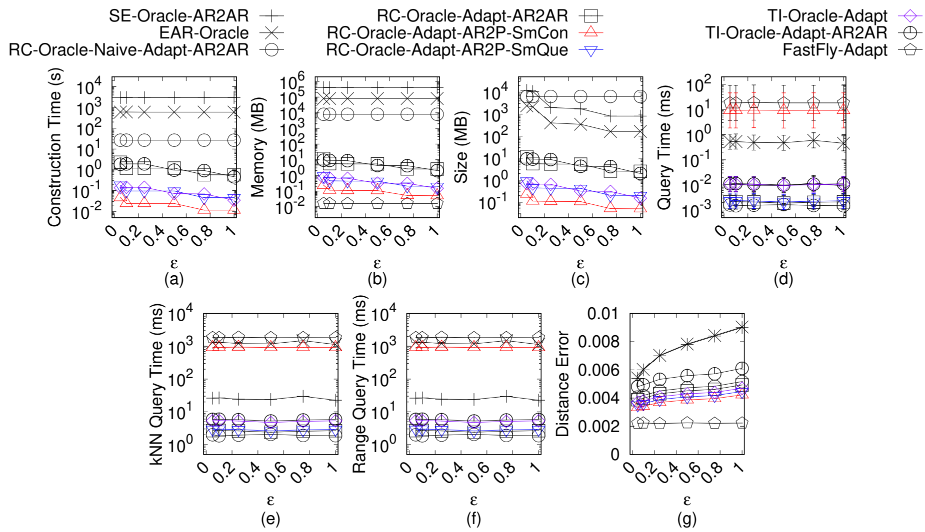

Fig. 54. Baseline comparisons (effect of ϵ on LM_t*-small TIN* dataset for the AR2P query)

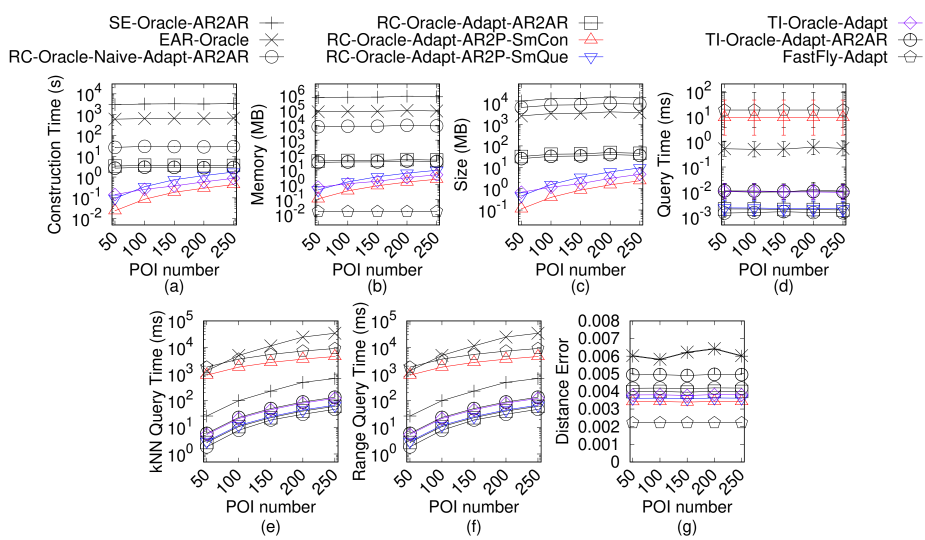

Fig. 55. Baseline comparisons (effect of n on LM_t*-small TIN* dataset for the AR2P query)

than *SU-Oracle-Adapt* and *RC-Oracle-NaiveProx*. Specifically, the *kNN* query time of *RC-Oracle* is 200 times smaller than that of *SU-Oracle-Adapt*. This is because the shortest path query time of *RC-Oracle* is $O(1)$, so even with the linear scan of the proximity query algorithm (in the worst case), the *kNN* query time of *RC-Oracle* is still fast.

C.2.4 ***Baseline comparisons for the A2P query***. We study the A2P query on *point clouds* for baseline comparisons. We (1) compared *SE-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle*, *TI-Oracle-A2A* and *FastFly* on small-version datasets with default 50 POIs, and (2) compared *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle*, *TI-Oracle-A2A* and *FastFly* on large-version

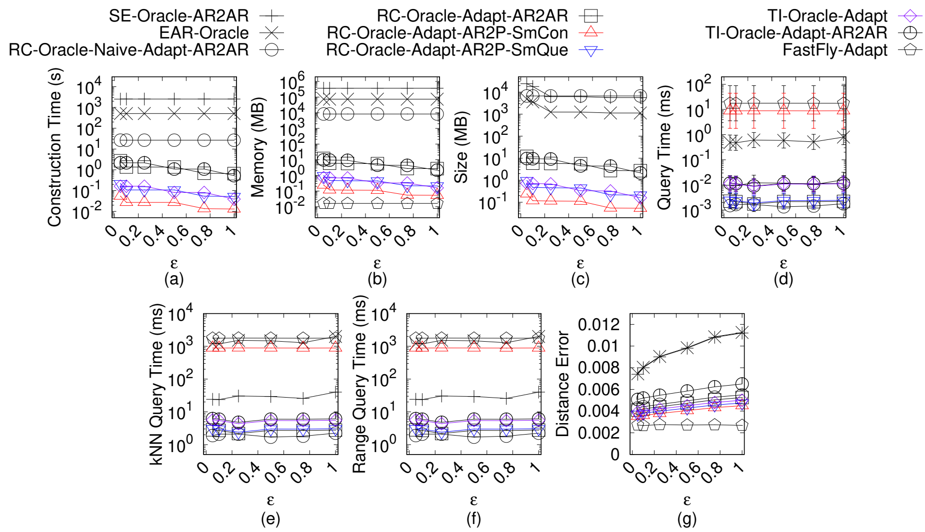

Fig. 56. Baseline comparisons (effect of ϵ on *RM_t-small TIN* dataset for the AR2P query)

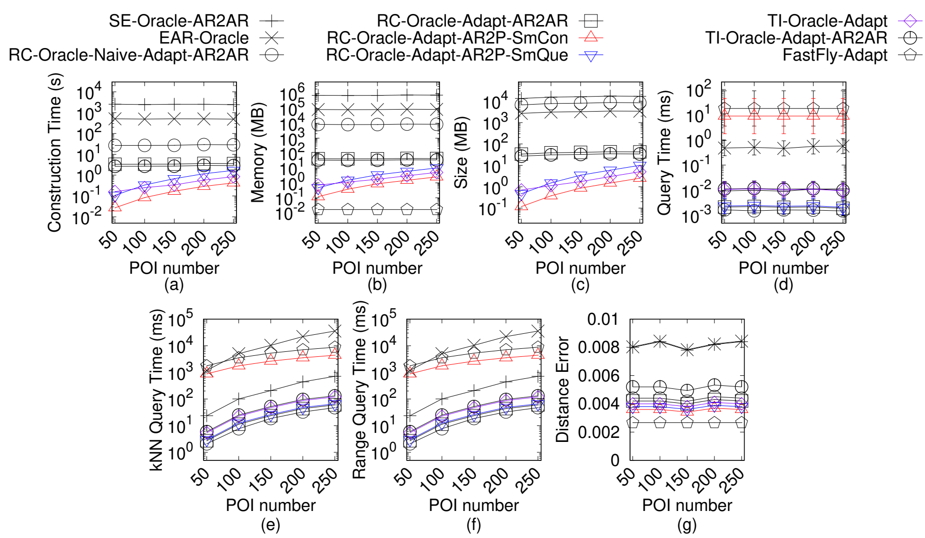

Fig. 57. Baseline comparisons (effect of n on *RM_t-small TIN* dataset for the AR2P query)

datasets with default 500 POIs. (1) Figure 109 and Figure 110 show the result on *BH_p-small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (2) Figure 111, Figure 112 and Figure 113 show the result on *EP_p-small* point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (3) Figure 114 and Figure 115 show the result on *GF_p-small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (4) Figure 116 and Figure 117 show the result on *LM_p-small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (5) Figure 118 and Figure 119 show the result on *RM_p-small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (6) Figure 120, Figure 121 and Figure 122 show the result on *BH_p* point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (6) Figure 120, Figure 121

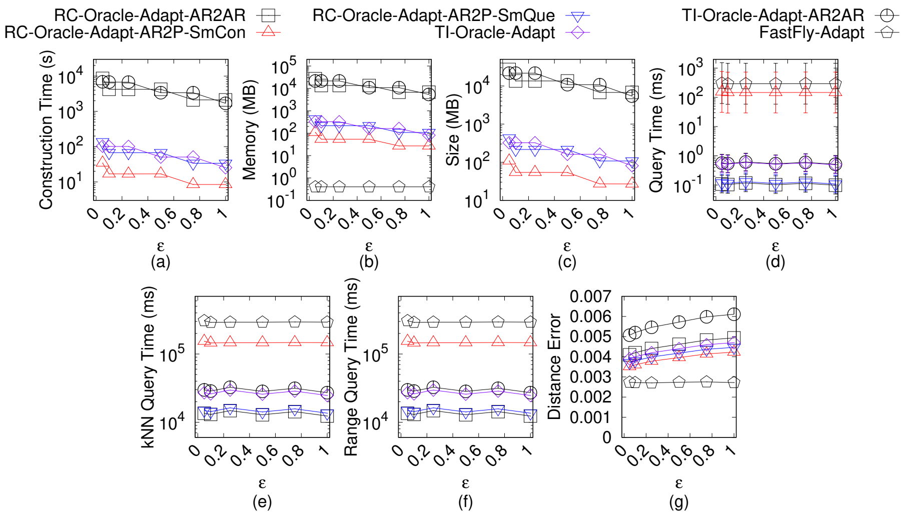

Fig. 58. Baseline comparisons (effect of ϵ on BH_t *TIN* dataset for the AR2P query)

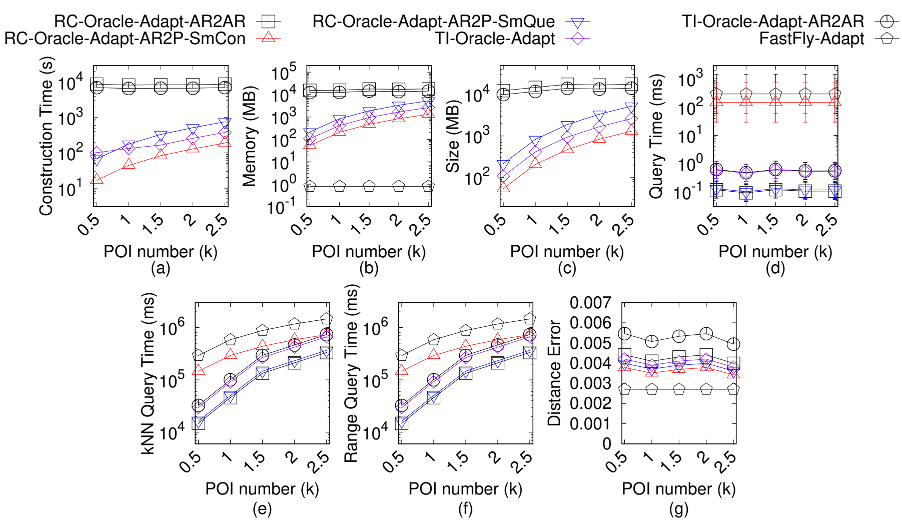

Fig. 59. Baseline comparisons (effect of n on BH_t *TIN* dataset for the AR2P query)

and Figure 122 show the result on BH_p point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (7) Figure 123, Figure 124 and Figure 125 show the result on EP_p point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (8) Figure 126, Figure 127 and Figure 128 show the result on GF_p point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (9) Figure 129, Figure 130 and Figure 131 show the result on LM_p point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (10) Figure 132, Figure 133 and Figure 134 show the result on RM_p point cloud dataset for the A2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 109, Figure 111, Figure 114, Figure 116 and Figure 118, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_p*-small*, EP_p*-small*, GF_p*-small*, LM_p*-small* and RM_p*-small* dataset by setting N to be 10k and n to be 50. In Figure 120, Figure 123, Figure 126, Figure 129 and

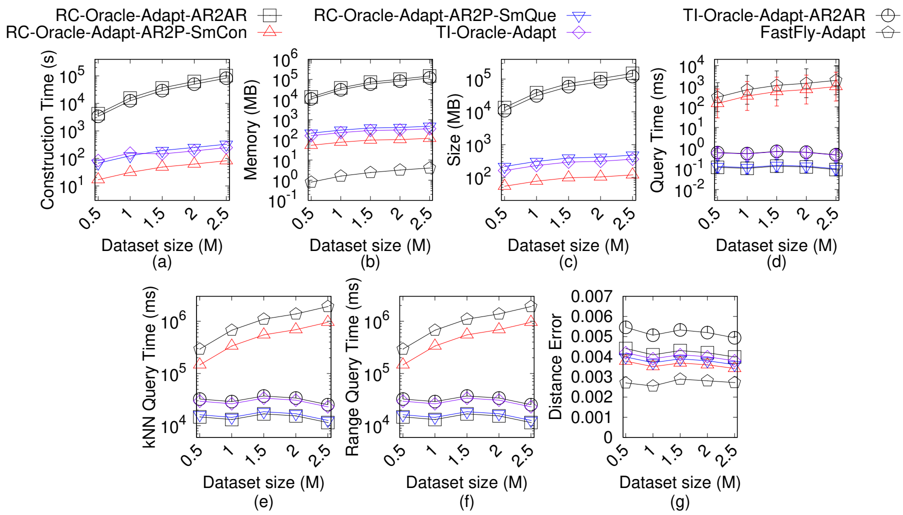

Fig. 60. Baseline comparisons (effect of N on BH_t *TIN* dataset for the AR2P query)

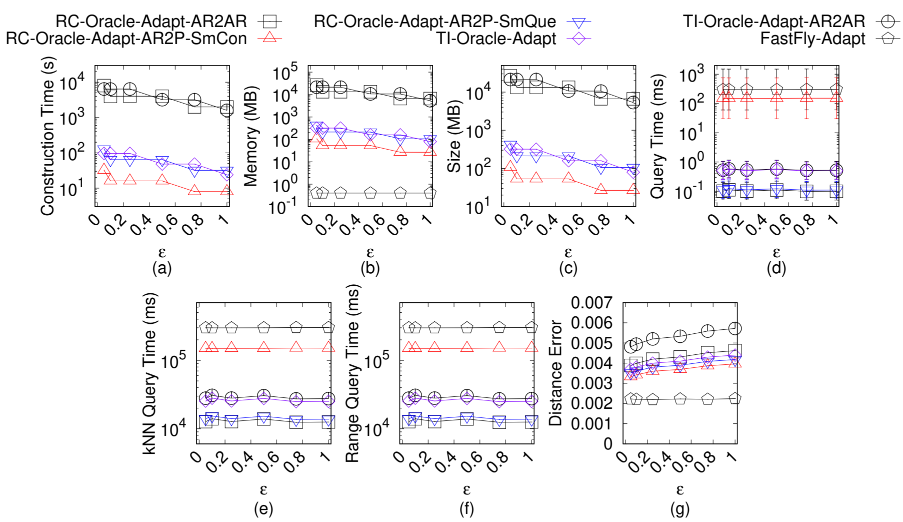

Fig. 61. Baseline comparisons (effect of ϵ on EP_t *TIN* dataset for the AR2P query)

Figure 132, 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. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon* and *TI-Oracle-A2P-SmCon* perform better than the best-known oracle *EAR-Oracle-Adapt* for the A2P query on a point cloud in terms of oracle construction time, oracle size and shortest path query time due to the *bad criterion for algorithm earlier termination* drawback of *EAR-Oracle-Adapt*

Effect of n. In Figure 110, Figure 112, Figure 115, Figure 117 and Figure 119, 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 121, Figure 124, Figure 127, Figure 130 and Figure 133, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and ϵ to be 0.25. (1) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon*

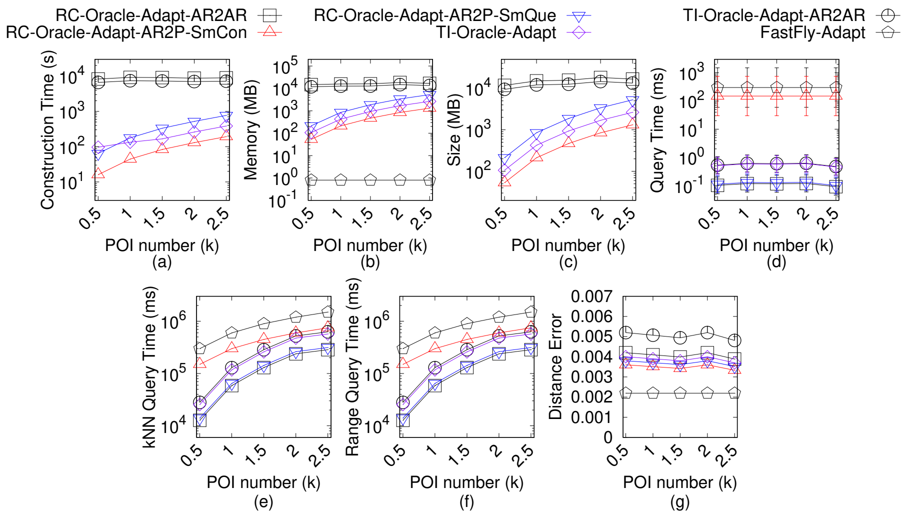

Fig. 62. Baseline comparisons (effect of n on EP_t *TIN* dataset for the AR2P query)

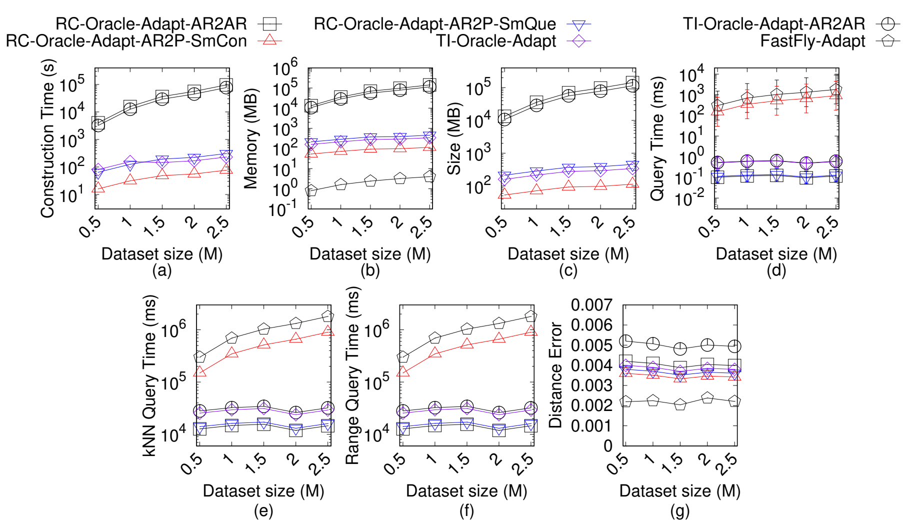

Fig. 63. Baseline comparisons (effect of N on EP_t *TIN* dataset for the AR2P query)

and *TI-Oracle-A2P-SmCon* also perform better than *RC-Oracle-A2A*. (2) The oracle construction time, memory usage and oracle size of *RC-Oracle-AR2P-SmCon* are the smallest, but its shortest path query time is larger than other oracles (but still smaller than *FastFly*) due to the same reason as that of *RC-Oracle-Adapt-AR2P-SmCon* for *TIN*s. Thus, it performs well in the case of fewer proximity queries. (3) The oracle construction time of *RC-Oracle-AR2P-SmQue* and *TI-Oracle* are also very small and their shortest path query time are also very small due to the same reason as those of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* for *TIN*s. When $n < 500$ (resp. $n \geq 500$), the oracle construction time of *RC-Oracle-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).

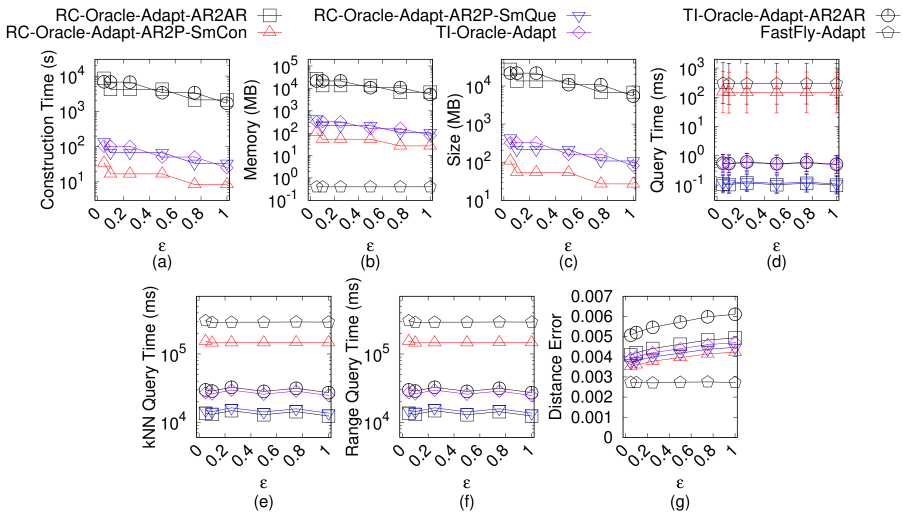

Fig. 64. Baseline comparisons (effect of ϵ on GF_t *TIN* dataset for the AR2P query)

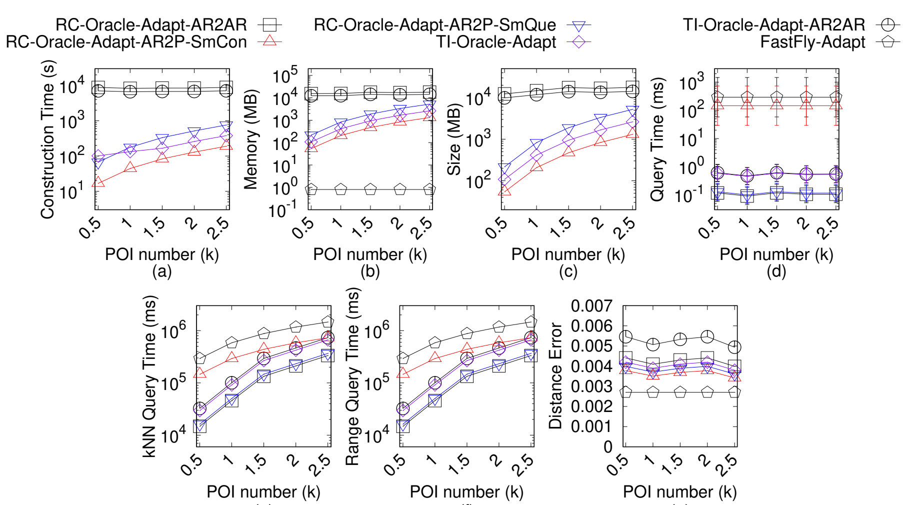

Fig. 65. Baseline comparisons (effect of n on GF_t *TIN* dataset for the AR2P query)

Effect of N (scalability test). In Figure 113, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_p*-small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 122, Figure 125, Figure 128, Figure 131 and Figure 134, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_p*-small*, EP_p*-small*, GF_p*-small*, LM_p*-small* and RM_p*-small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. The oracle construction time of *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* are only 80s ≈ 1.3 min, 310s ≈ 5.1 min and 250s ≈ 4.1min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of them.

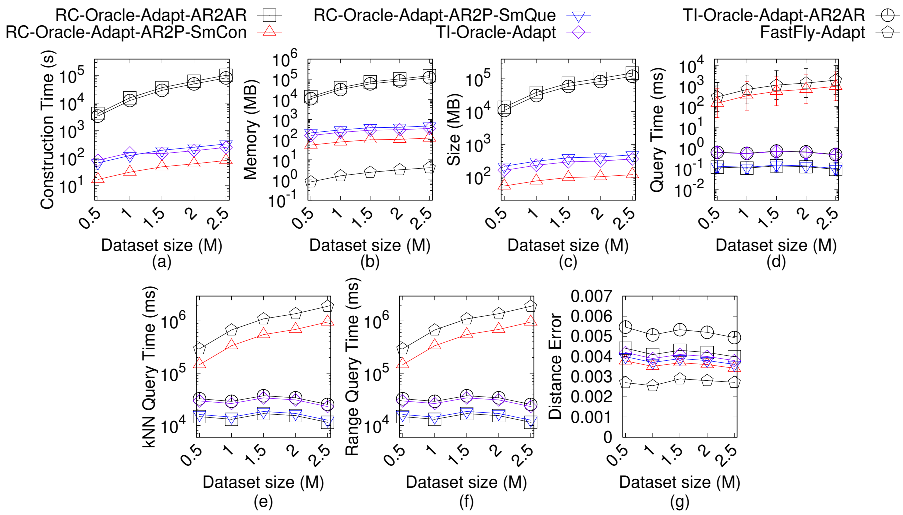

Fig. 66. Baseline comparisons (effect of N on GF_t *TIN* dataset for the AR2P query)

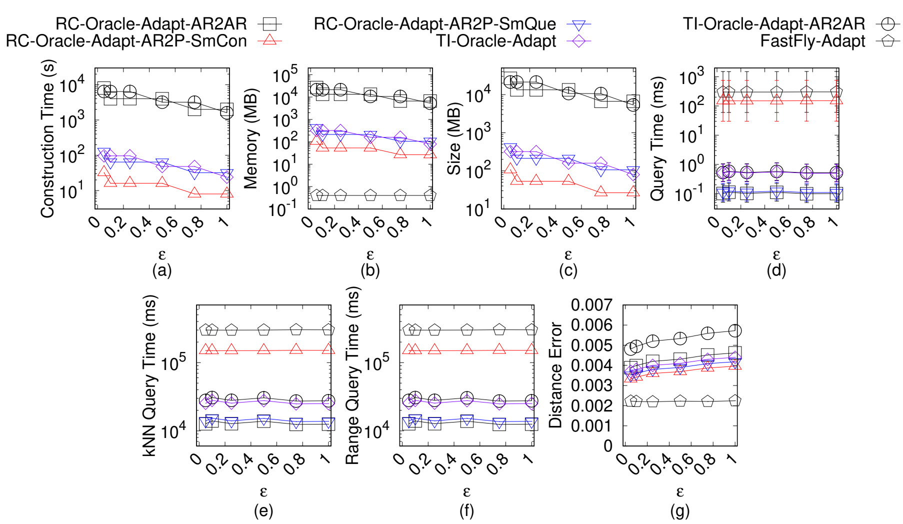

Fig. 67. Baseline comparisons (effect of ϵ on LM_t *TIN* dataset for the AR2P query)

C.2.5 ***Ablation study for the A2P query.*** We study the A2P query on *point clouds* for ablation study. We compared *SE-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A*.

In Figure 135, Figure 136, Figure 137, Figure 138 and Figure 139, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p, EP_p, GF_p, LM_p and RM_p dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size and shortest path query time of *RC-Oracle-A2P-SmQue* and *TI-Oracle* also perform better than *SE-Oracle-FastFly-Adapt-A2A* and *EAR-Oracle-FastFly-Adapt*.

C.2.6 ***Comparisons with other proximity queries oracles and variation oracles for the A2P query.*** We study the A2P query on *point clouds* for comparisons with other proximity queries

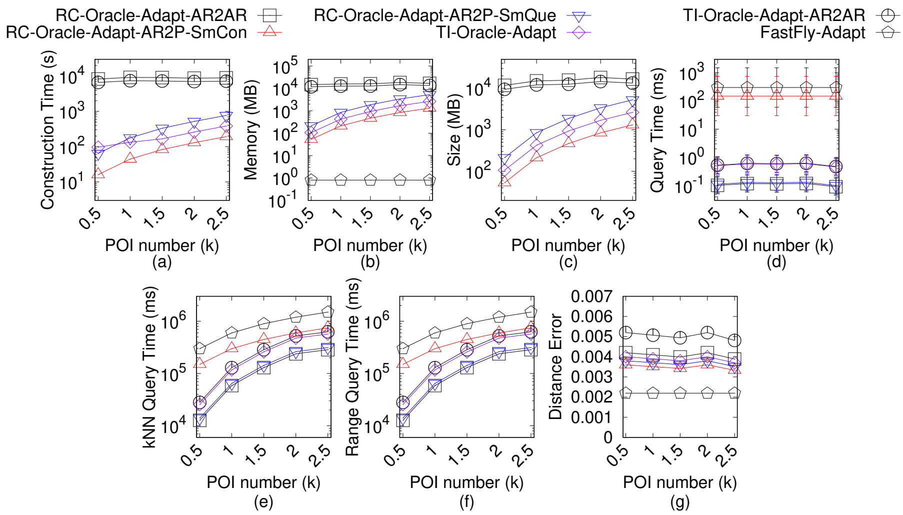

Fig. 68. Baseline comparisons (effect of n on LM_t *TIN* dataset for the AR2P query)

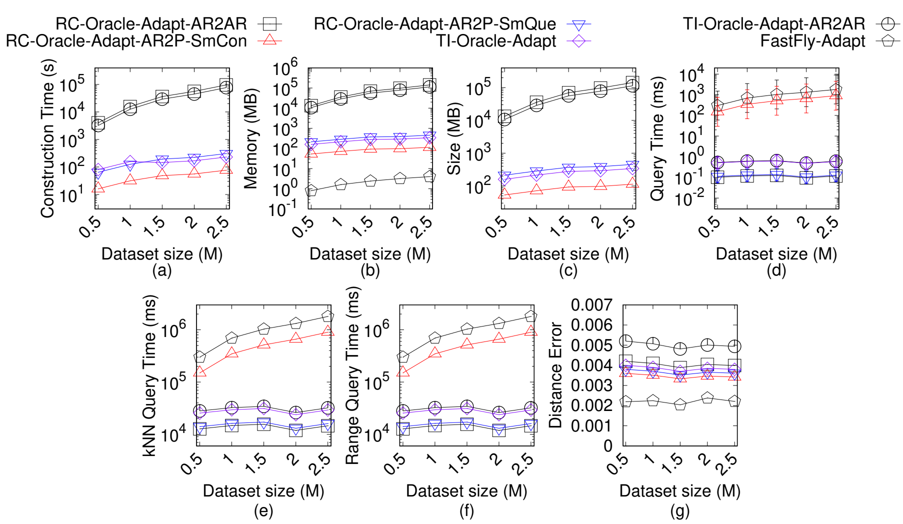

Fig. 69. Baseline comparisons (effect of N on LM_t *TIN* dataset for the AR2P query)

oracles and variation oracles. We adapt *SU-Oracle-Adapt* to be *SU-Oracle-Adapt-A2A* in the similar way of of *TI-Oracle-A2A* for the A2A query on a point cloud. We denote *TI-Oracle-NaiveProx* to be *TI-Oracle* using the naive proximity query algorithm. We denote *TI-Oracle-Rtree* to be *TI-Oracle* using the R-tree and 2D boxes. We denote *TI-Oracle-TigLoo* to be *TI-Oracle* using tight/loose surface indexes. We compared *SU-Oracle-Adapt*, *SU-Oracle-Adapt-A2A*, *TI-Oracle-NaiveProx*, *TI-Oracle-Rtree*, *TI-Oracle-TigLoo*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmCon*.

In Figure 140, Figure 141, Figure 142, Figure 143 and Figure 144, 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. (1) Our oracles and variations still perform better than *SU-Oracle-Adapt* and *SU-Oracle-Adapt-A2A*. (2) For a point cloud with 2.5M points and 500 POIs, the oracle size and *kNN* query time of *TI-Oracle*

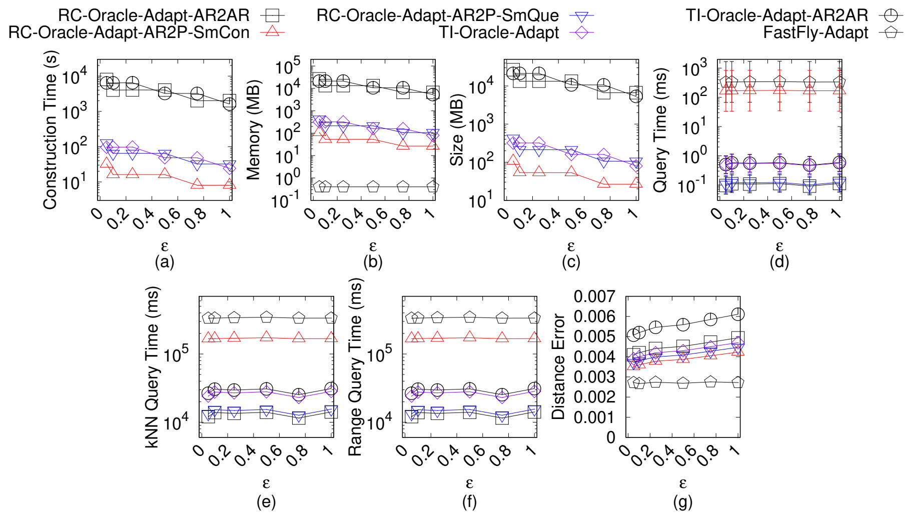

Fig. 70. Baseline comparisons (effect of ϵ on RM_t *TIN* dataset for the AR2P query)

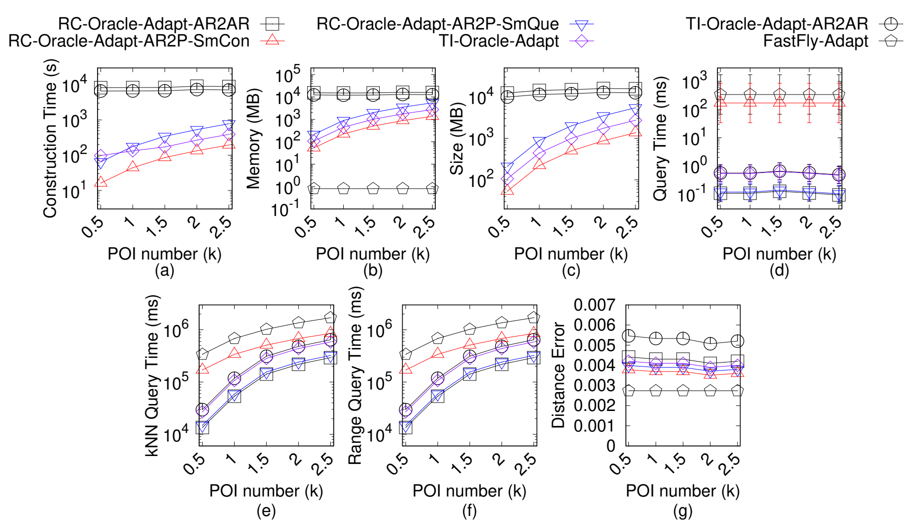

Fig. 71. Baseline comparisons (effect of n on RM_t *TIN* dataset for the AR2P query)

are 350MB and 23s, but are 348MB and 104s of *TI-Oracle-Rtree*. The latter one needs to store the R-tree, 2D boxes and points coordinate information, so it can only slightly reduce the oracle size compared with the former one. But, the latter one needs to use the R-tree to find which partition cell of a query point belongs to, and the leaf nodes of 2D boxes contain more than one endpoint used for creating the partition cell (see Figure 9 of study [58]), so it significantly increases the shortest path query time compared with the former one. (3) For a point cloud with 2.5M points and 500 POIs, the oracle construction time and *kNN* query time of *TI-Oracle* are 250s ≈ 4.1 min and 23s, but are 500s ≈ 8.2 min and 20s of *TI-Oracle-TigLoo*. The latter one uses tight/loose surface indexes, so it can gradually expand in *kNN* queries to slightly reduce *kNN* query time compared with the former one. But, the latter one contains more (almost twice compared with *TI-Oracle* using partition

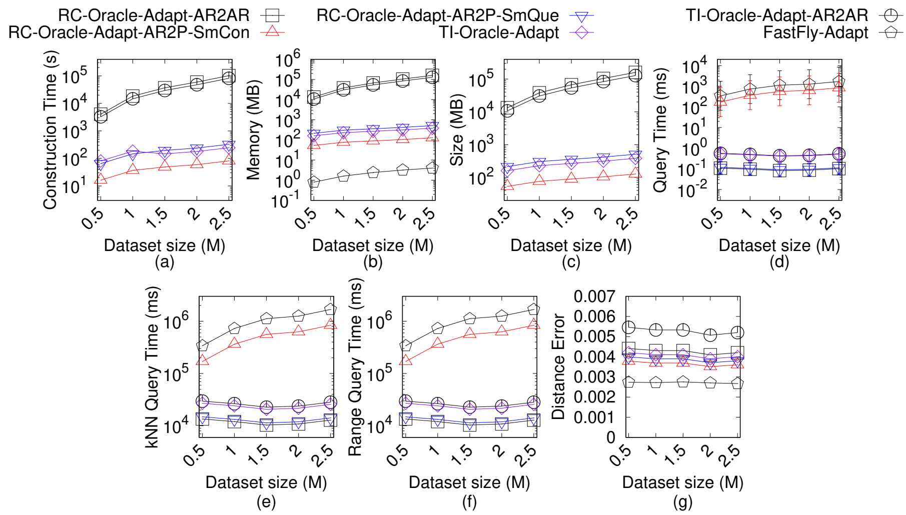

Fig. 72. Baseline comparisons (effect of N on RM_t *TIN* dataset for the AR2P query)

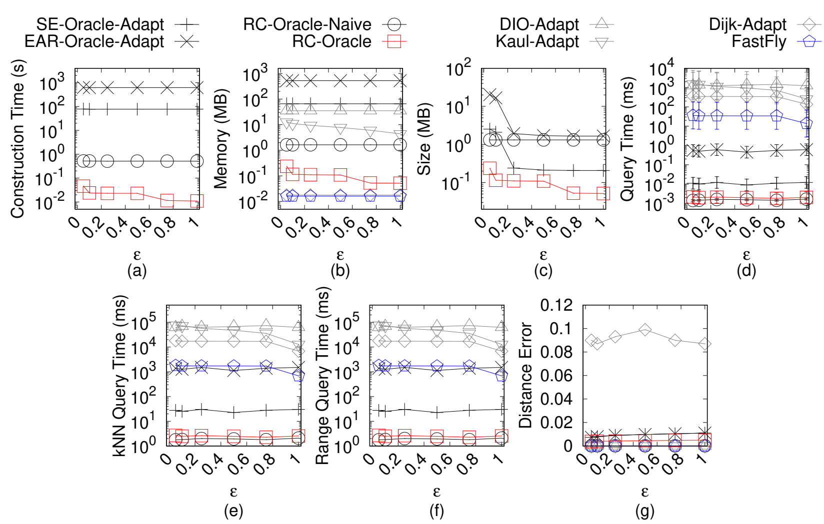

Fig. 73. Baseline comparisons (effect of ϵ on BH_p*-small* point cloud dataset for the P2P query)

cells) boundary points between tight/loose surface indexes and the point cloud, so it significantly increases the oracle construction time compared with the former one.

C.2.7 ***Baseline comparisons for the A2A query***. We study the A2A query on point clouds for baseline comparisons. In the same figures of baseline comparisons for the A2P query on point clouds, we compared *SE-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time.

C.2.8 ***Ablation study for the A2A query for ablation study***. We study the A2A query on point clouds for ablation study. In the same figures of ablation study for the A2P query on point

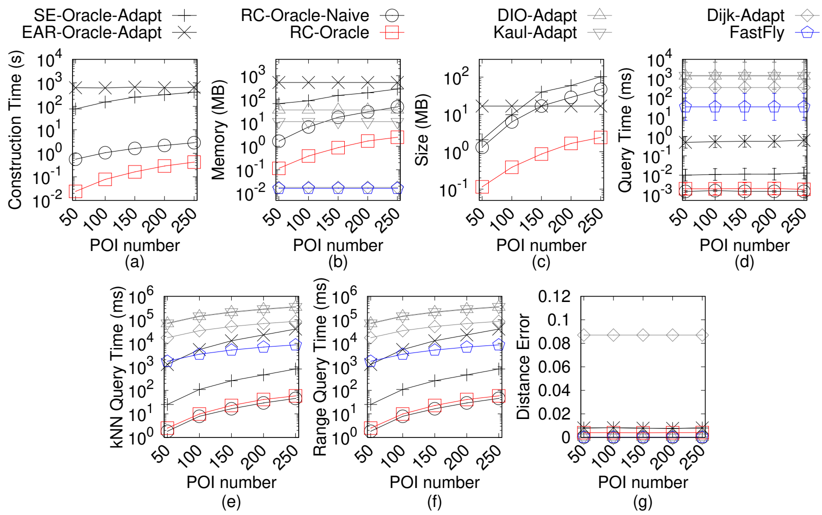

Fig. 74. Baseline comparisons (effect of n on BH_p*-small* point cloud dataset for the P2P query)

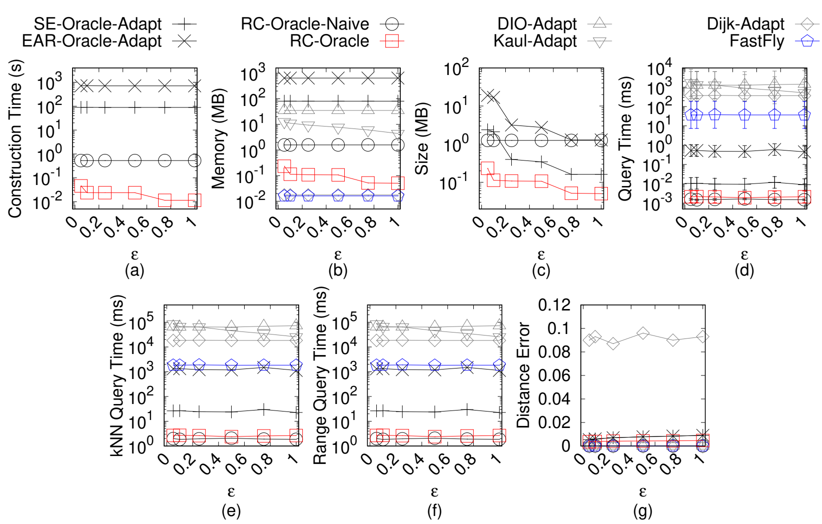

Fig. 75. Baseline comparisons (effect of ϵ on EP_p*-small* point cloud dataset for the P2P query)

clouds, we compared *SE-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time.

C.2.9 ***Comparisons with other proximity queries oracles and variation oracles for the A2A query.*** We study the A2A query on *point clouds* for comparisons with other proximity queries oracle. In the same figures of comparisons with other proximity queries oracles and variation oracles for the A2P query on point clouds, we compared *SU-Oracle-Adapt-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first oracle in terms of oracle construction time, oracle size and shortest path query time. Since in the comparisons with other

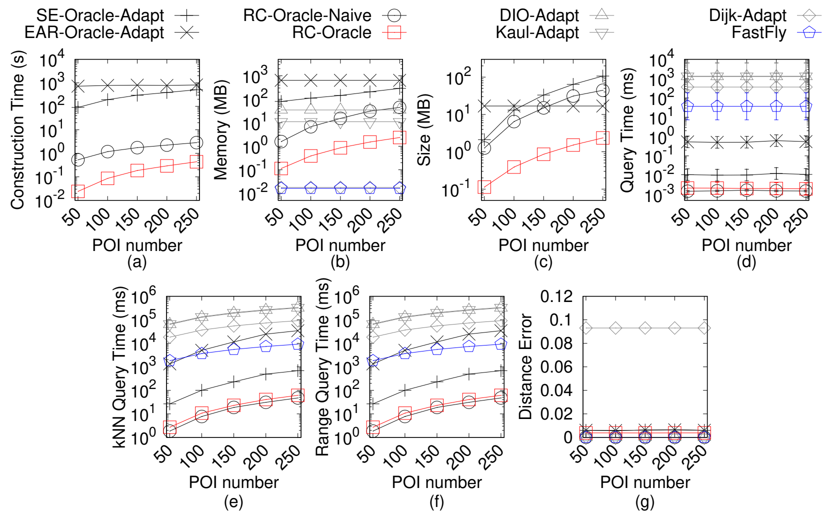

Fig. 76. Baseline comparisons (effect of n on EP_p*-small* point cloud dataset for the P2P query)

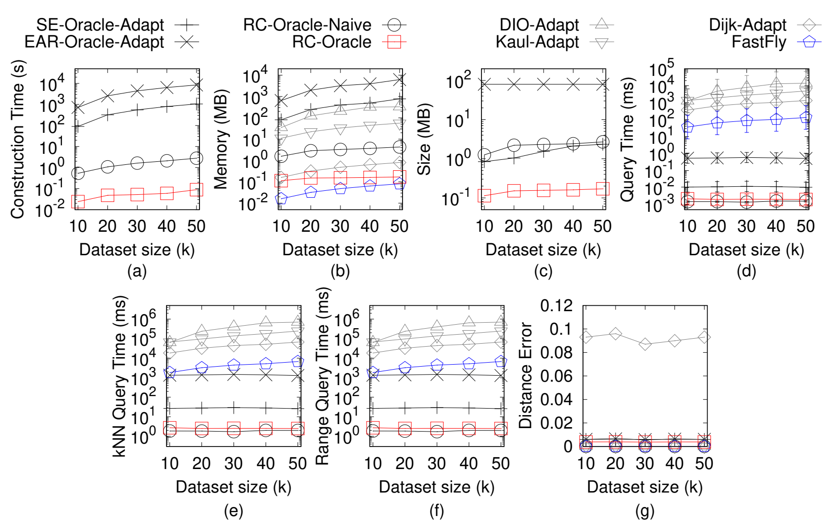

Fig. 77. Baseline comparisons (effect of N on EP_p*-small* point cloud dataset for the P2P query)

proximity queries oracles and variation oracles for the A2P query on point clouds, we have shown that *TI-Oracle* performs better than *TI-Oracle-NaiveProx*, *TI-Oracle-Rtree* and *TI-Oracle-TigLoo*, there is no need to to compare *TI-Oracle-A2A* using the naive proximity query algorithms, using the R-tree and using tight/loose surface indexes.

C.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 [45, 62, 63]. 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

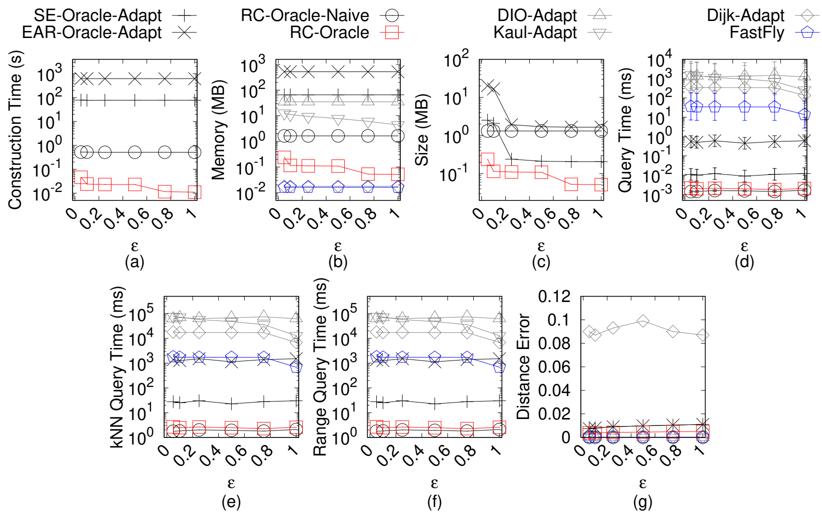

Fig. 78. Baseline comparisons (effect of ϵ on GF_p*-small* point cloud dataset for the P2P query)

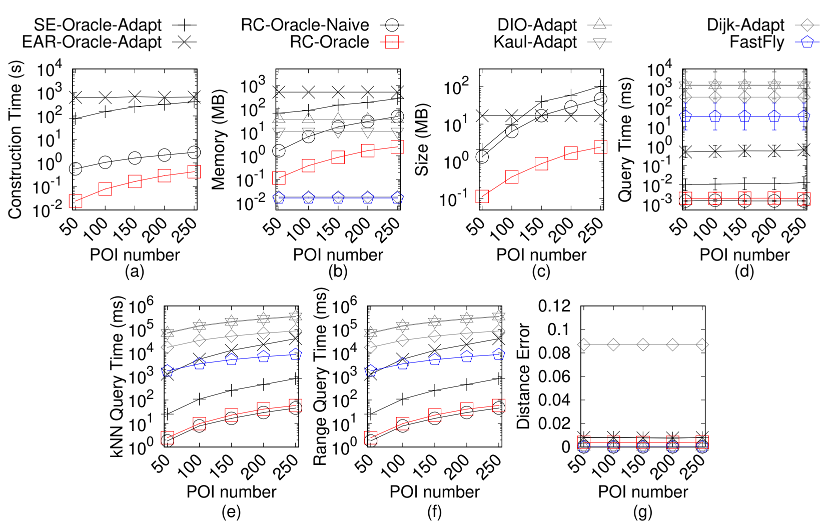

Fig. 79. Baseline comparisons (effect of n on GF_p*-small* point cloud dataset for the P2P query)

we currently have with qx_o edges along x-coordinate, qy_o edges along y-coordinate and N_o points, where $N_o = qx_o \cdot qy_o$. We then generate $qx_t \cdot qy_t$ 2D points (x, y) based on a Normal distribution $N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{q_o \in C_o} x_{q_o}}{qx_o \cdot qy_o}, \overline{y} = \frac{\sum_{q_o \in C_o} y_{q_o}}{qx_o \cdot qy_o})$ and $\sigma_N^2 = (\frac{\sum_{q_o \in C_o}(x_{q_o} - \overline{x})^2}{qx_o \cdot qy_o}, \frac{\sum_{q_o \in C_o}(y_{q_o} - \overline{y})^2}{qx_o \cdot qy_o})$. 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 generated C_t.

D COMPARISON OF ALL ALGORITHMS

Table 4 shows a comparison of all algorithms (support the shortest path query) in terms of the oracle construction time, oracle size and shortest path query time, and Table 5 shows a comparison

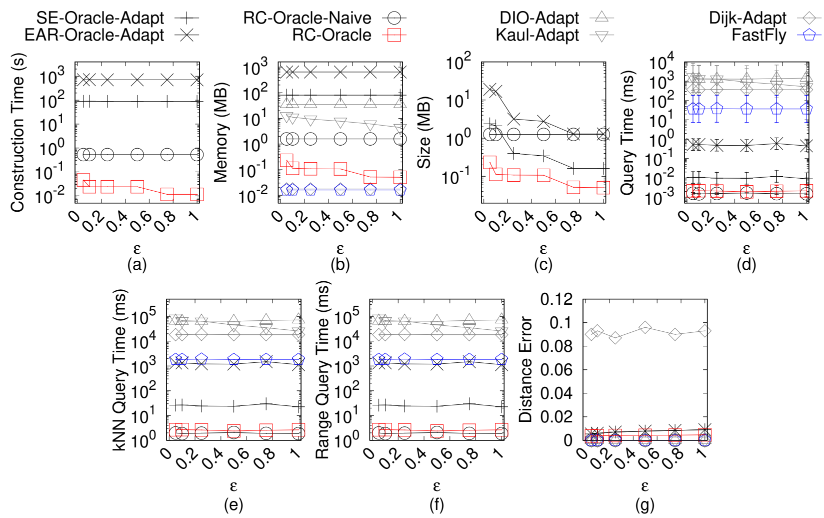

Fig. 80. Baseline comparisons (effect of ϵ on *LM_p-small* point cloud dataset for the P2P query)

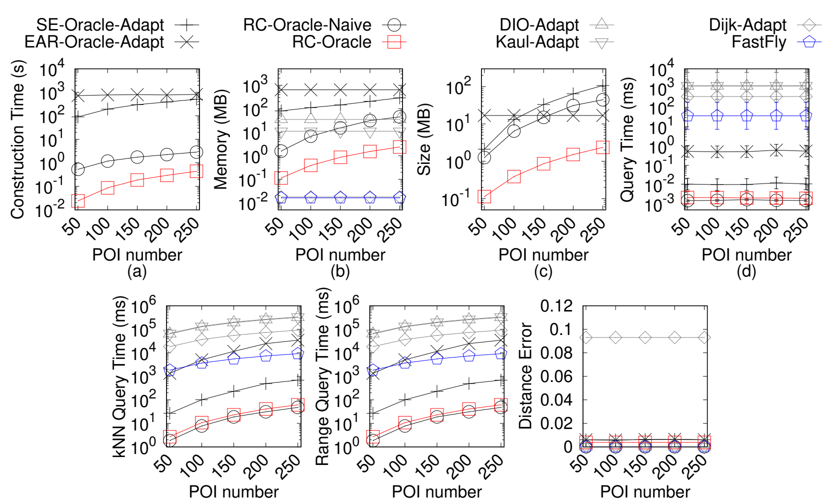

Fig. 81. Baseline comparisons (effect of n on *LM_p-small* point cloud dataset for the P2P query)

of other proximity queries oracles and their variation oracles in terms of the oracle construction time, oracle size and *kNN* query time.

E PROOF

Proof of Lemma 4.4. We give the proof for *RC-Oracle* as follows.

Firstly, we show the query time of both the *kNN* and range queries algorithm. Given a query object, when we need to perform the *kNN* query or the range query, the worst case is that we need to perform a linear scan to check the distance between this query object to all other objects using the shortest path query phase of *RC-Oracle* in $O(1)$ time. Since there are total n' objects, the query time is $O(n')$. However, the real query time is smaller than $O(n')$. This is because for most of the

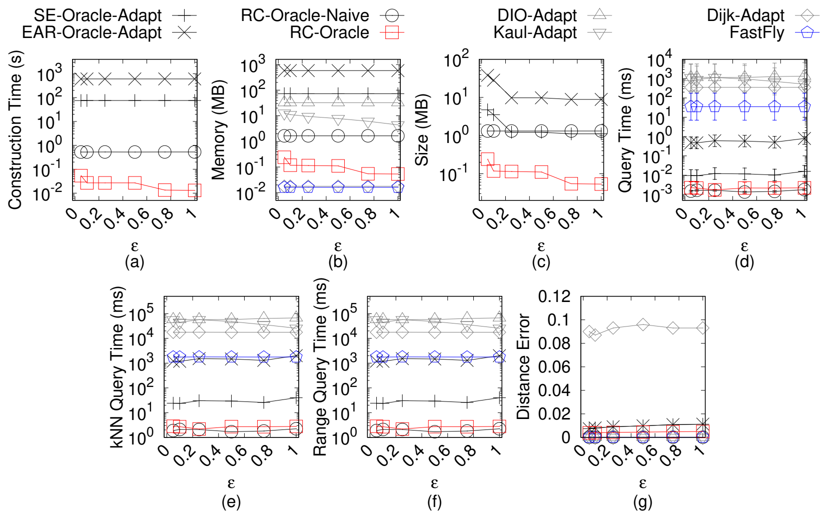

Fig. 82. Baseline comparisons (effect of ϵ on *RM_p-small* point cloud dataset for the P2P query)

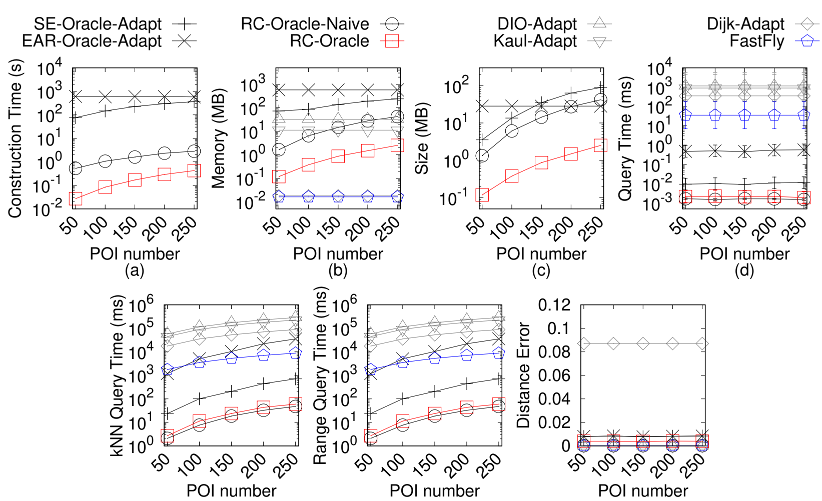

Fig. 83. Baseline comparisons (effect of n on *RM_p-small* point cloud dataset for the P2P query)

cases, we do not need to perform a linear scan (since we have already sorted some distances in order during the construction phase of *RC-Oracle*).

Secondly, we show the error rate of both the *kNN* and range queries algorithm for *RC-Oracle*. We give some definitions first. For simplicity, given a query POI $q \in P$, (1) we let X be a set of POIs containing the *exact* (i) k nearest POIs of q or (ii) POIs whose distance to q are at most r, calculated using the exact distance on C. Furthermore, given a query POI $q \in P$, (2) we let X' be a set of POIs containing (i) k nearest POIs of q or (ii) POIs whose distance to q are at most r, calculated using the approximated distance on C returned by *RC-Oracle*. In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d, i.e., $X = \{c, d\}$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c, i.e., $X' = \{b, c\}$. Recall that we let v_f (resp. v'_f) be the furthest object to q in X

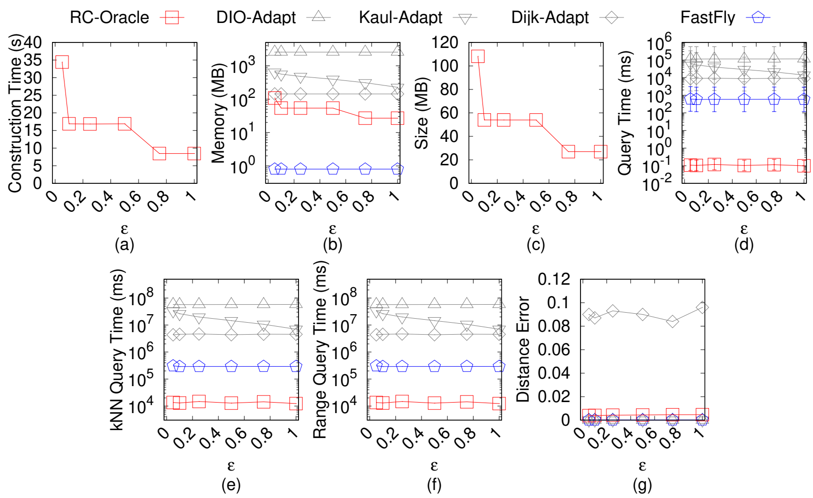

Fig. 84. Baseline comparisons (effect of ϵ on BH_p point cloud dataset for the P2P query)

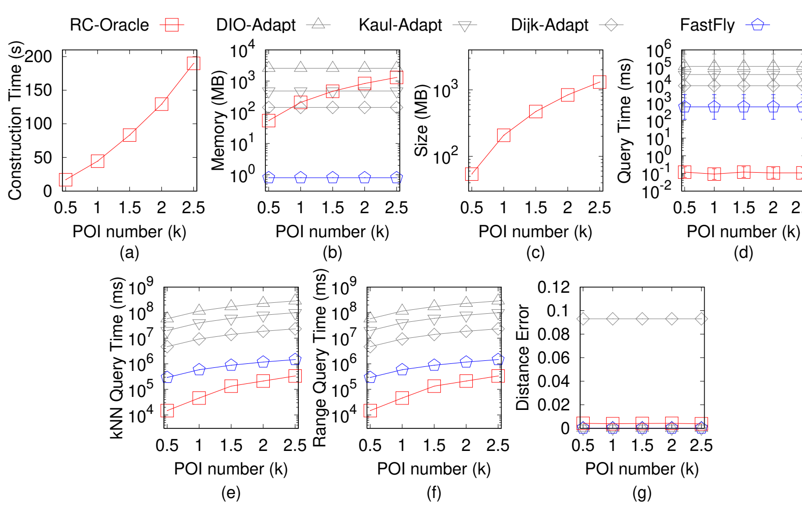

Fig. 85. Baseline comparisons (effect of n on BH_p point cloud dataset for the P2P query)

(resp. X'), i.e., $|\Pi^*(q, v_f|C)| \leq \max_{\forall v \in X} |\Pi^*(q, v|C)|$ (resp. $|\Pi^*(q, v'_f|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q, v'|C)|$). We further let w_f (resp. w'_f) be the furthest object to q in X (resp. X') based on the approximated distance on C returned by *RC-Oracle*, i.e., $|\Pi_{RC\text{-}Oracle}(q, w_f|C)| \leq \max_{\forall w \in X} |\Pi_{RC\text{-}Oracle}(q, w|C)|$ (resp. $|\Pi_{RC\text{-}Oracle}(q, w'_f|C)| \leq \max_{\forall w' \in X'} |\Pi_{RC\text{-}Oracle}(q, w'|C)|$). Recall the error rate of the *kNN* and range queries 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 $|\Pi_{RC\text{-}Oracle}(q, v'_f|C)| \geq |\Pi^*(q, v'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$. 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 $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q,v'_f|C)|}{|\Pi^*(q,w_f|C)|}$. By the definition of v'_f and w'_f, we have $|\Pi_{RC\text{-}Oracle}(q, v'_f|C)| \leq$

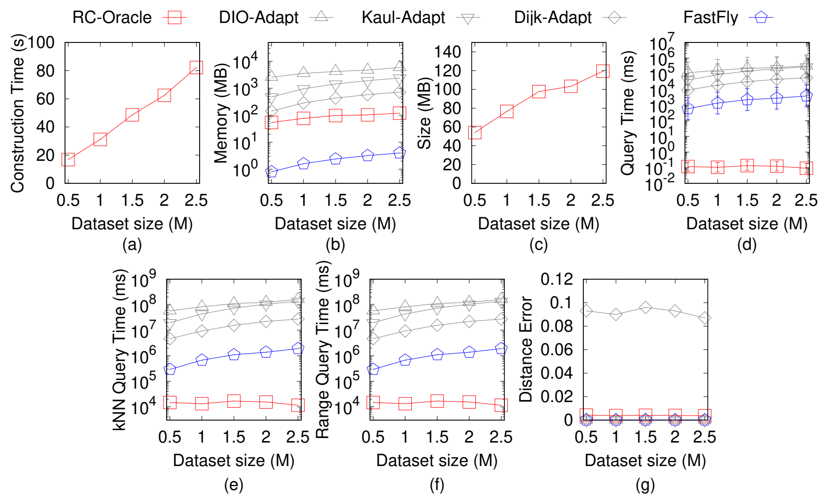

Fig. 86. Baseline comparisons (effect of N on BH_p point cloud dataset for the P2P query)

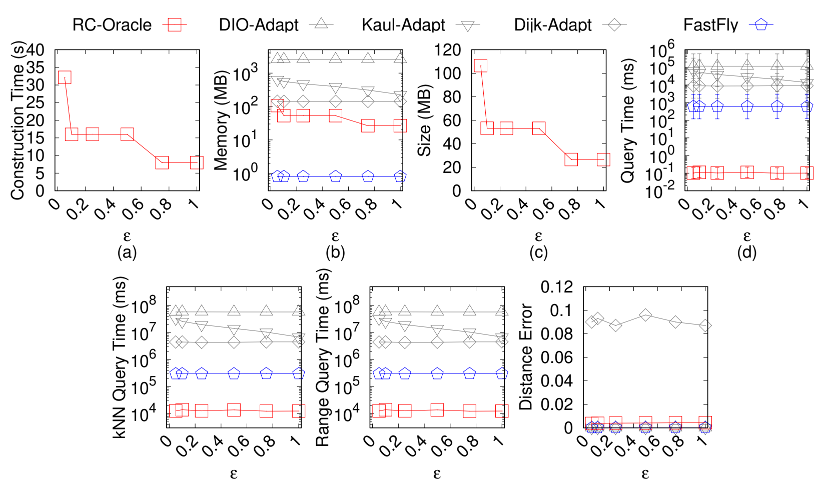

Fig. 87. Baseline comparisons (effect of ϵ on EP_p point cloud dataset for the P2P query)

$|\Pi_{RC\text{-}Oracle}(q, w'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, w'_f|C)|}{|\Pi^*(q, w_f|C)|}$. Since the error rate of the approximated distance on C returned by *RC-Oracle* is $1+\epsilon$, we have $|\Pi_{RC\text{-}Oracle}(q, w_f|C)| \leq (1+\epsilon)|\Pi^*(q, w_f|C)|$. Then, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, w'_f|C)|(1+\epsilon)}{|\Pi_{RC\text{-}Oracle}(q, w_f|C)|}$. By our *kNN* and range queries algorithm, we have $|\Pi_{RC\text{-}Oracle}(q, w'_f|C)| \leq |\Pi_{RC\text{-}Oracle}(q, w_f|C)|$. Thus, we have $\alpha \leq 1+\epsilon$.

We give the proof for *RC-Oracle-A2P-SmCon* as follows. For the query time of both the *kNN* and range queries algorithm, it is similar to *RC-Oracle*. But, *RC-Oracle-A2P-SmCon* also has the new shortest paths calculation step before the shortest path query step. *RC-Oracle-A2P-SmCon* just need to use the new shortest paths calculation step once in $O(N \log N)$ time. After that, it needs the shortest path query step in $O(n')$ time. Thus, the total query time of both the *kNN* and

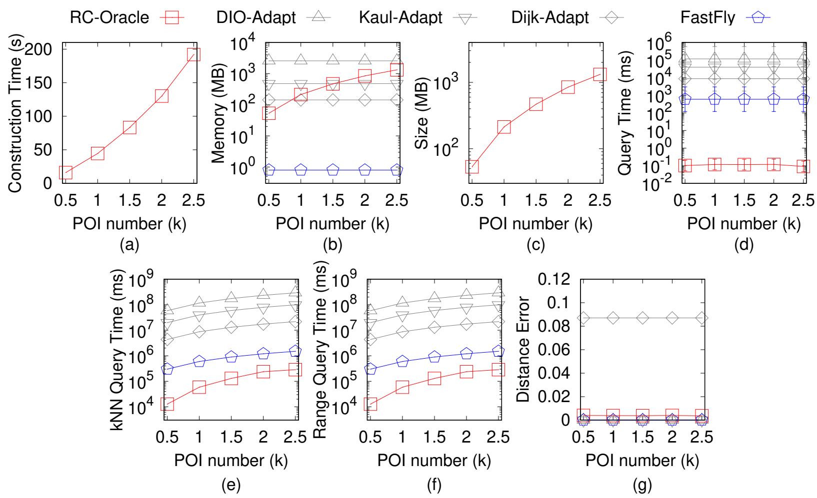

Fig. 88. Baseline comparisons (effect of n on EP_p point cloud dataset for the P2P query)

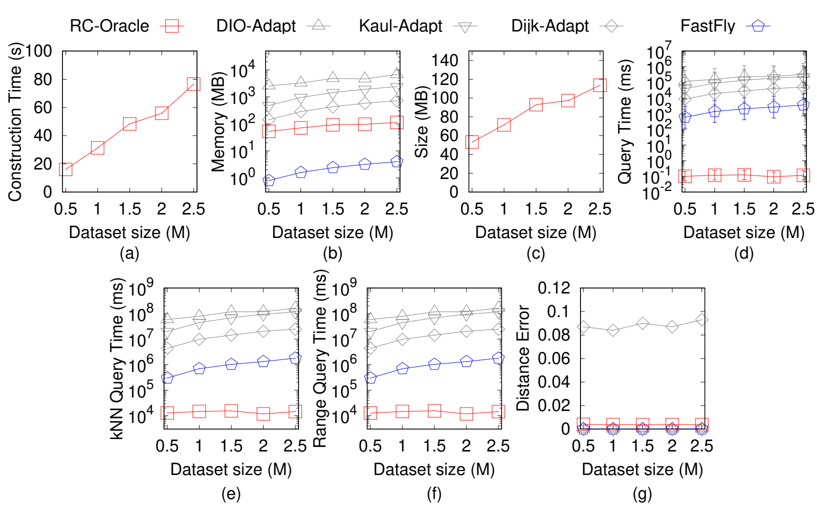

Fig. 89. Baseline comparisons (effect of N on EP_p point cloud dataset for the P2P query)

range queries algorithm are $O(N \log N + n')$ time. For the error rate, since *RC-Oracle-A2P-SmCon* is also a $(1+\epsilon)$-approximate shortest path oracle, its error rate of both the *kNN* and range queries algorithm is the same as that of *RC-Oracle*.

We give the proof for *RC-Oracle-A2P-SmQue* as follows. Since the shortest path query time of *RC-Oracle-A2P-SmQue* is the same as that of *RC-Oracle*, and *RC-Oracle-A2P-SmQue* is also a $(1+\epsilon)$-approximate shortest path oracle, its query time and error rate of both the *kNN* and range queries algorithm is the same as that of *RC-Oracle*.

We give the proof for *RC-Oracle-A2A* as follows. Since the shortest path query time of *RC-Oracle-A2A* is the same as that of *RC-Oracle*, and *RC-Oracle-A2A* is also a $(1+\epsilon)$-approximate shortest

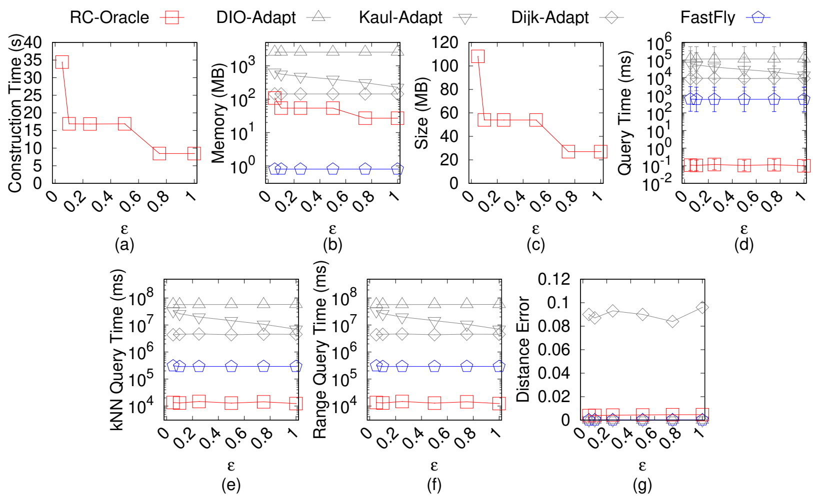

Fig. 90. Baseline comparisons (effect of ϵ on GF_p point cloud dataset for the P2P query)

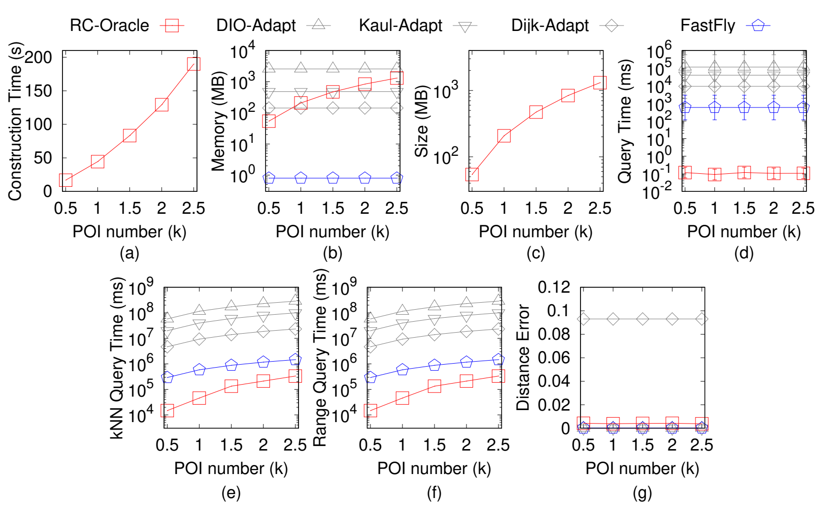

Fig. 91. Baseline comparisons (effect of n on GF_p point cloud dataset for the P2P query)

path oracle, its query time and error rate of both the *kNN* and range queries algorithm is the same as that of *RC-Oracle*. □

Proof of Lemma 5.2. We give the proof for *TI-Oracle* and *TI-Oracle-A2A* as follows. Since the shortest path query time of *TI-Oracle* and *TI-Oracle-A2A* are the same as that of *RC-Oracle*, and *TI-Oracle* and *TI-Oracle-A2A* are also $(1+\epsilon)$-approximate shortest path oracles, their query time and error rate of both the *kNN* and range queries algorithm are the same as that of *RC-Oracle*. □

Theorem E.1. *The shortest path query time and memory consumption of algorithm DIO-Adapt are $O(N^2 \log N)$ and $O(N^2)$. Compared with $\Pi^*(s,t|T)$, algorithm DIO-Adapt returns the exact shortest*

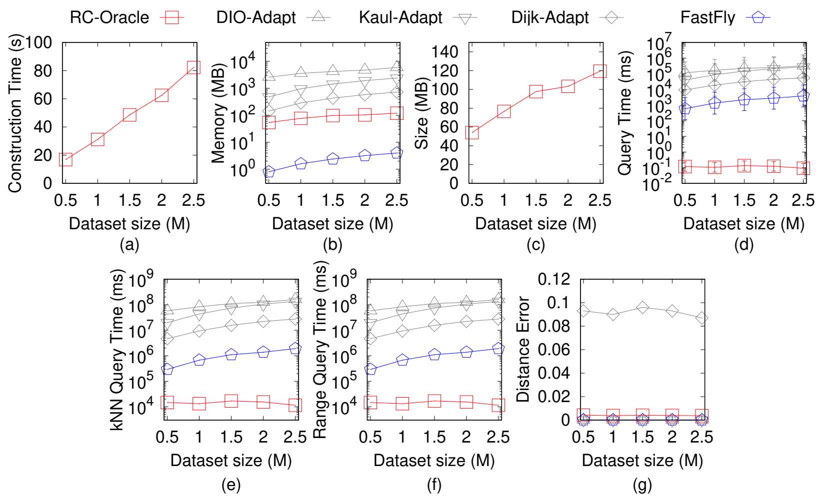

Fig. 92. Baseline comparisons (effect of N on GF_p point cloud dataset for the P2P query)

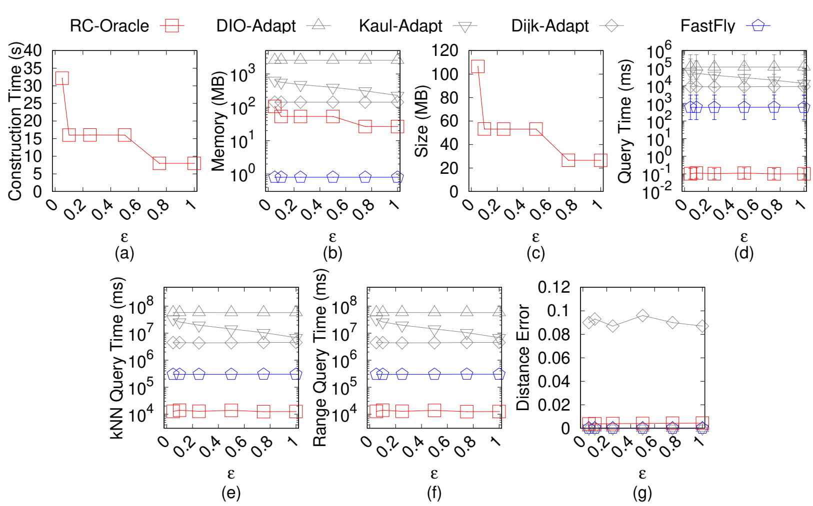

Fig. 93. Baseline comparisons (effect of ϵ on LM_p point cloud dataset for the P2P query)

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

Proof. Firstly, we show the *shortest path query time* of algorithm *DIO-Adapt*. The proof of the shortest path query time of algorithm *DIO-Adapt* is in [64]. But since algorithm *DIO-Adapt* first needs to construct the *TIN* using the point cloud, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *DIO-Adapt* is $O(N + N^2 \log N) = O(N^2 \log N)$.

Secondly, we show the *memory consumption* of algorithm *DIO-Adapt*. The proof of the memory consumption of algorithm *DIO-Adapt* is in [64]. Thus, the memory consumption of algorithm *DIO-Adapt* is $O(N^2)$.

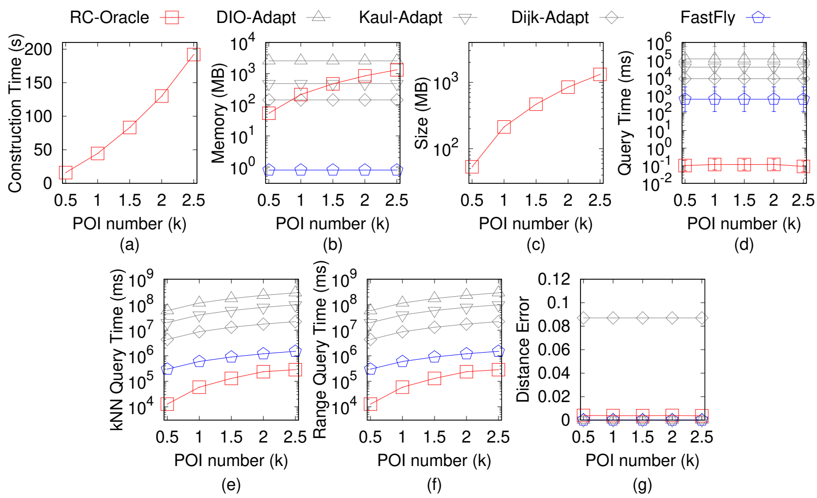

Fig. 94. Baseline comparisons (effect of n on LM_p point cloud dataset for the P2P query)

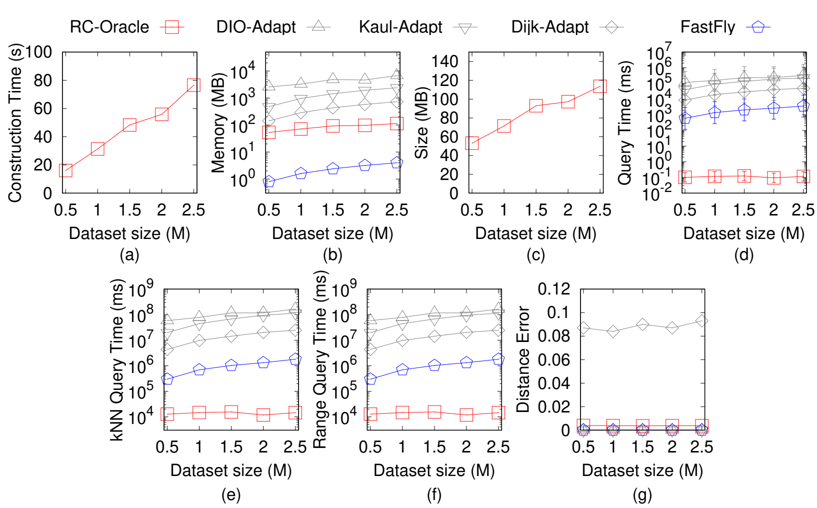

Fig. 95. Baseline comparisons (effect of N on LM_p point cloud dataset for the P2P query)

Thirdly, we show the *error bound* of algorithm *DIO-Adapt*. Compared with $\Pi^*(s,t|T)$, the proof that algorithm *DIO-Adapt* returns the exact shortest path passing on a *TIN* is in [64]. Since the *TIN* is constructed by the point cloud, so algorithm *DIO-Adapt* returns the exact shortest surface path passing on a *TIN* (that is constructed by the point cloud). Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *DIO-Adapt* returns the approximate shortest path passing on a point cloud. □

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

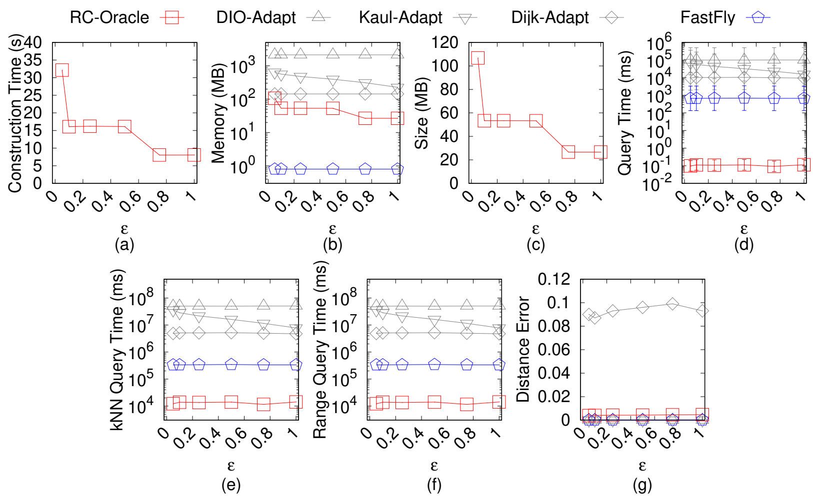

Fig. 96. Baseline comparisons (effect of ϵ on RM_p point cloud dataset for the P2P query)

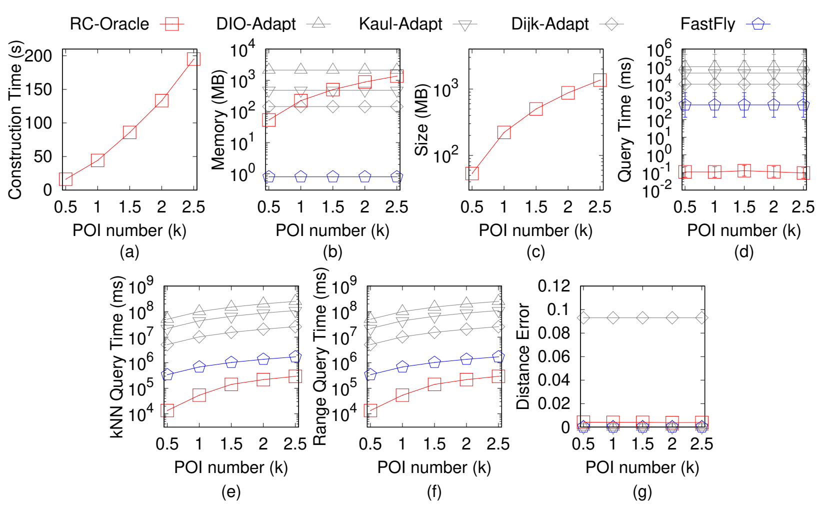

Fig. 97. Baseline comparisons (effect of n on RM_p point cloud dataset for the P2P query)

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

Proof. Firstly, we show the *shortest path query time* of algorithm *Kaul-Adapt*. The proof of the shortest path query time of algorithm *Kaul-Adapt* is in [35]. Note that in Section 4.2 of [35], the shortest path query time of algorithm *Kaul-Adapt* is $O((N+N')(\log(N+N')+(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where $N'=O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ 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 $\frac{1}{K-1}$. Thus, we can derive that the shortest path query time of algorithm *Kaul-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}})+\frac{l_{max}^2}{(\epsilon l_{min}\sqrt{1-\cos\theta})^2})$. Since for N, the first term is larger than the second term, so we obtain the shortest path query time

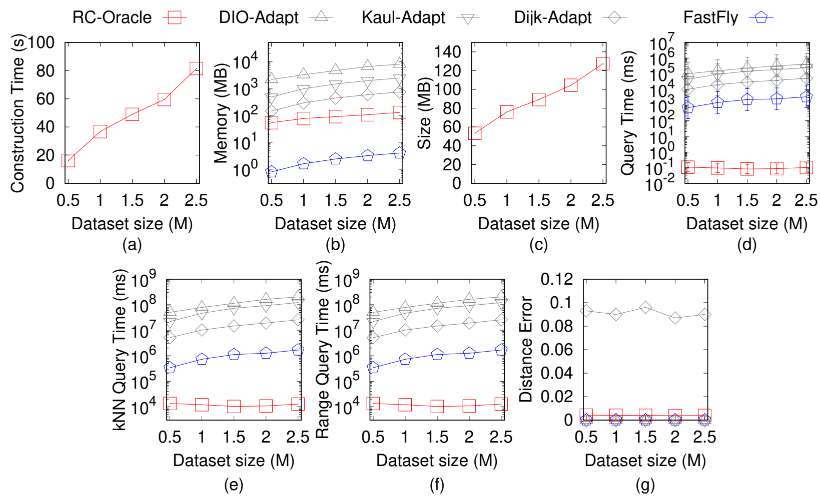

Fig. 98. Baseline comparisons (effect of N on RM_p point cloud dataset for the P2P query)

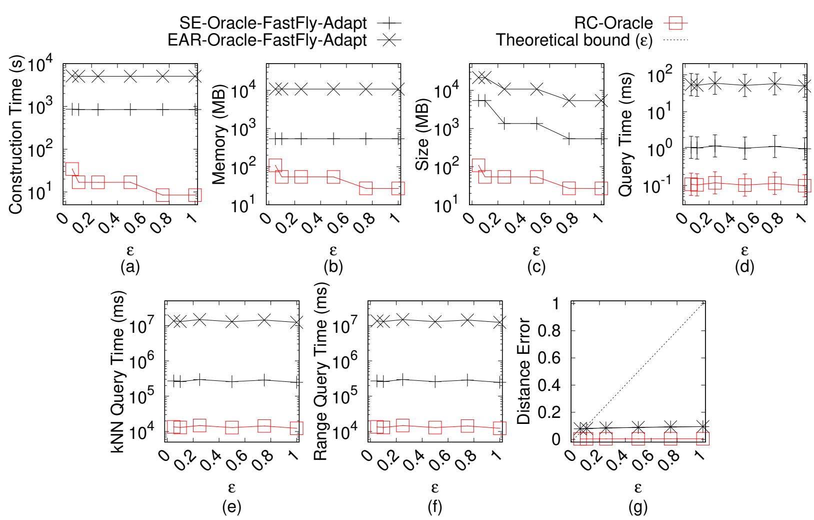

Fig. 99. Ablation study on BH_p point cloud dataset for the P2P query

of algorithm *Kaul-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. But since algorithm *Kaul-Adapt* first needs to construct a *TIN* using the point cloud, so it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Kaul-Adapt* is $O(N+\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}})) = O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$.

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

Thirdly, we show the *error bound* of algorithm *Kaul-Adapt*. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of algorithm *Kaul-Adapt* is in [35]. Since the *TIN* is constructed by the point cloud, so algorithm *Kaul-Adapt* always has $|\Pi_{Kaul\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pairs of

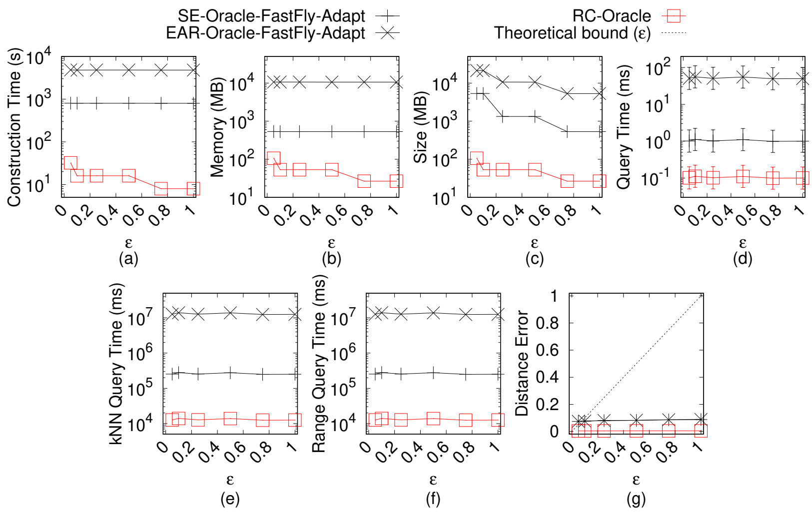

Fig. 100. Ablation study on EP_p point cloud dataset for the P2P query

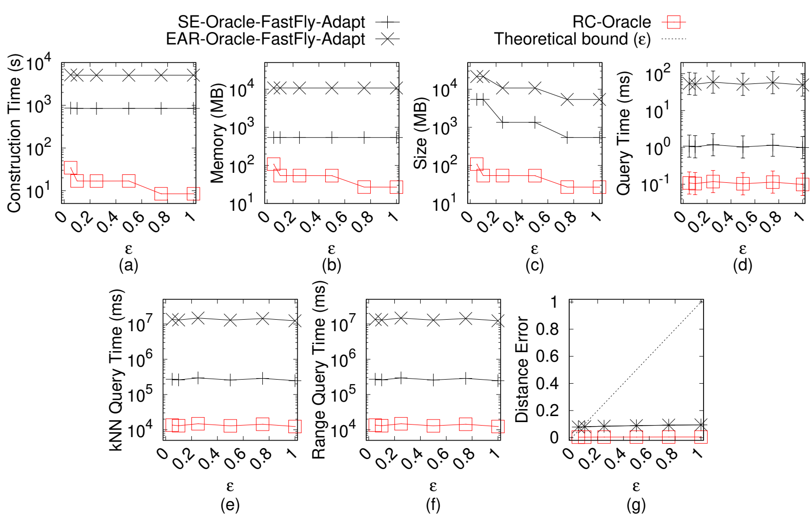

Fig. 101. Ablation study on GF_p point cloud dataset for the P2P query

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

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

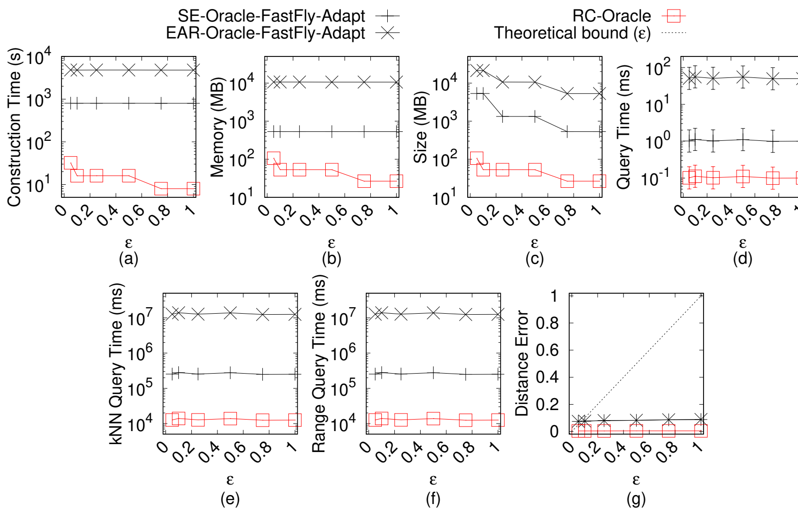

Fig. 102. Ablation study on LM_p point cloud dataset for the P2P query

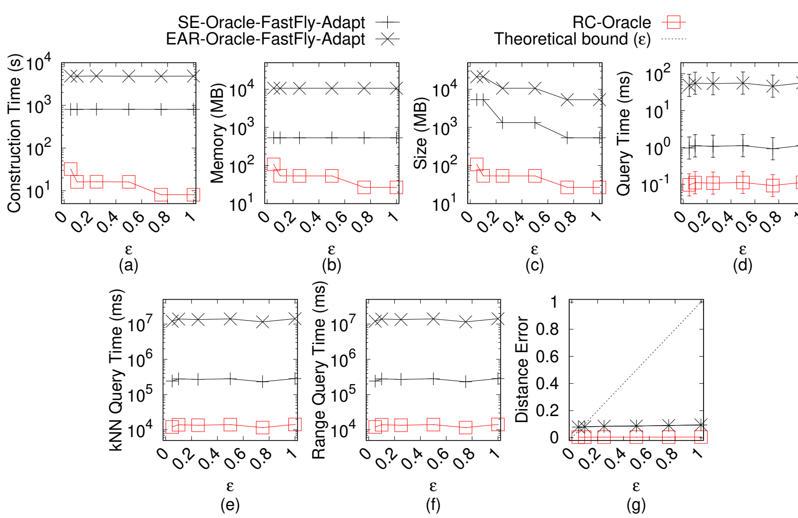

Fig. 103. Ablation study on RM_p point cloud dataset for the P2P query

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

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

Thirdly, we show the *error bound* of algorithm *Dijk-Adapt*. Recall that $\Pi_N(s,t|T)$ is the shortest network path passing on T (that is constructed by the point cloud) between s and t, so actually

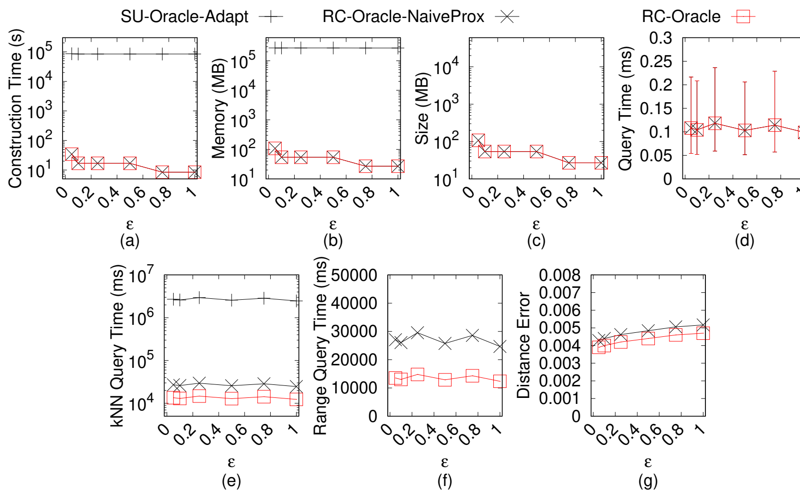

Fig. 104. Comparisons with other proximity queries oracles and variation oracles on BH_p point cloud dataset for the P2P query

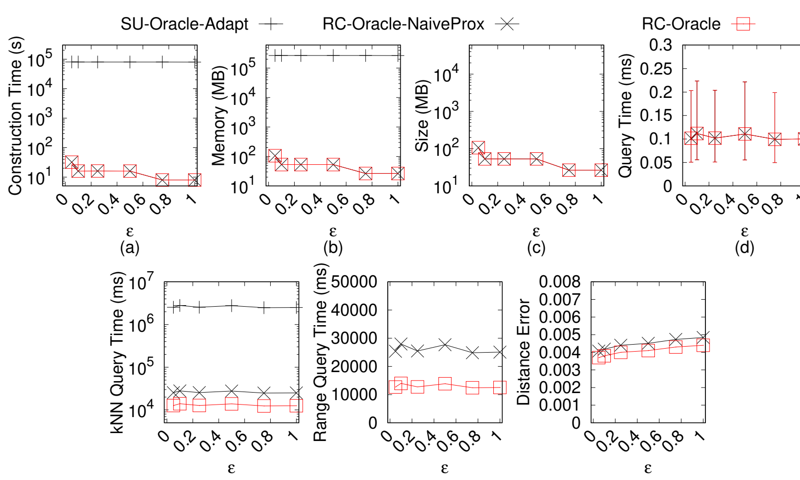

Fig. 105. Comparisons with other proximity queries oracles and variation oracles on EP_p point cloud dataset for the P2P query

$\Pi_N(s, t|T)$ is the same as $\Pi_{\mathit{Dijk\text{-}Adapt}}(s, t|T)$. We let $\Pi_E(s, t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s, t|T)$ passes) between s and t. Compared with $\Pi^*(s, t|T)$, we know $|\Pi_E(s, t|T)| \leq k \cdot |\Pi^*(s, t|T)|$ (according to left hand side equation in Lemma 2 of [36]) and $|\Pi_N(s, t|T)| \leq |\Pi_E(s, t|T)|$ (since $\Pi_N(s, t|T)$ considers all the edges on T), so we have algorithm *Dijk-Adapt* always has $|\Pi_{\mathit{Dijk\text{-}Adapt}}(s, t|T)| \leq k \cdot |\Pi^*(s, t|T)|$ for any pairs of vertices s and t on T. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on the point cloud, algorithm *Dijk-Adapt* returns the approximate shortest path passing on a point cloud. □

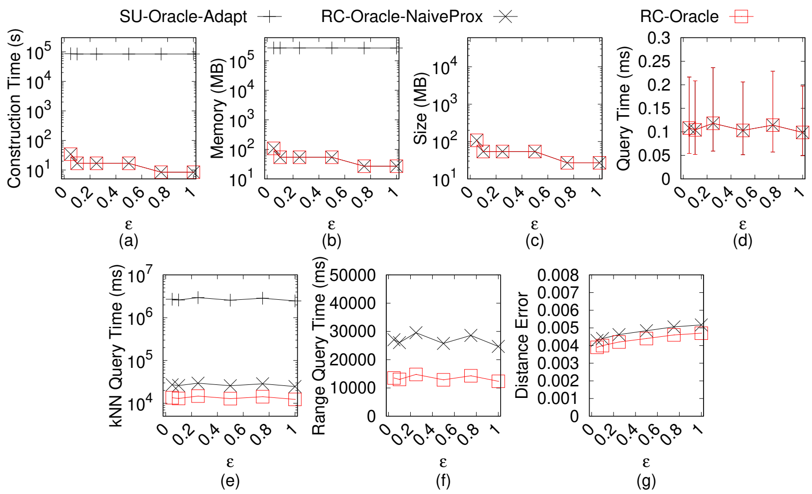

Fig. 106. Comparisons with other proximity queries oracles and variation oracles on GF_p point cloud dataset for the P2P query

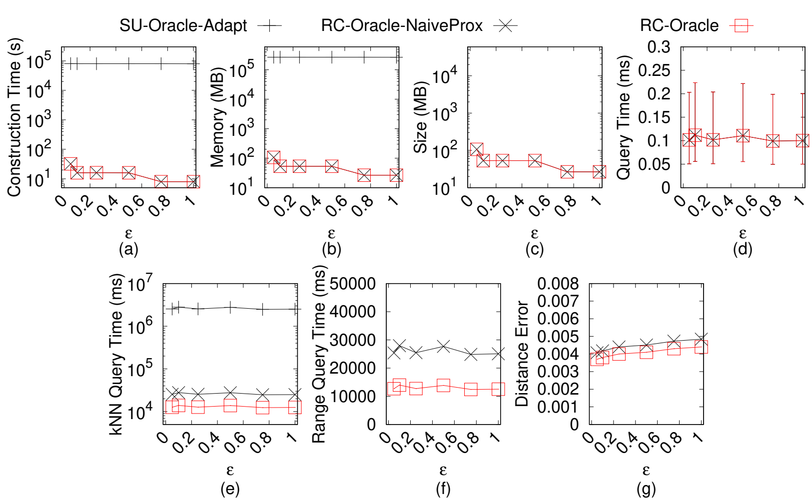

Fig. 107. Comparisons with other proximity queries oracles and variation oracles on LM_p point cloud dataset for the P2P query

Theorem E.4. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-Adapt are* $O(nN^2 + n\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ *and* $O(h^2)$. *Compared with* $\Pi^*(s,t|T)$, *SE-Oracle-Adapt always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{\textit{SE-Oracle-Adapt}}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for any pairs of POIs* s *and* t *in* P, *where* $\Pi_{\textit{SE-Oracle-Adapt}}(s,t|T)$ *is the shortest surface path of SE-Oracle-Adapt passing on a TIN* T *(that is constructed by the point cloud) between* s *and* t. *Compared with* $\Pi^*(s,t|C)$, *algorithm SE-Oracle-Adapt returns the approximate shortest path passing on a point cloud.*

Proof. Firstly, we show the *oracle construction time* of *SE-Oracle-Adapt*. The oracle construction time of the original oracle in [62, 63] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-Adapt*, we use algorithm *DIO* for the *TIN* shortest path query, which has shortest path query time $O(N^2\log N)$ according to Theorem E.1. But, we also need to

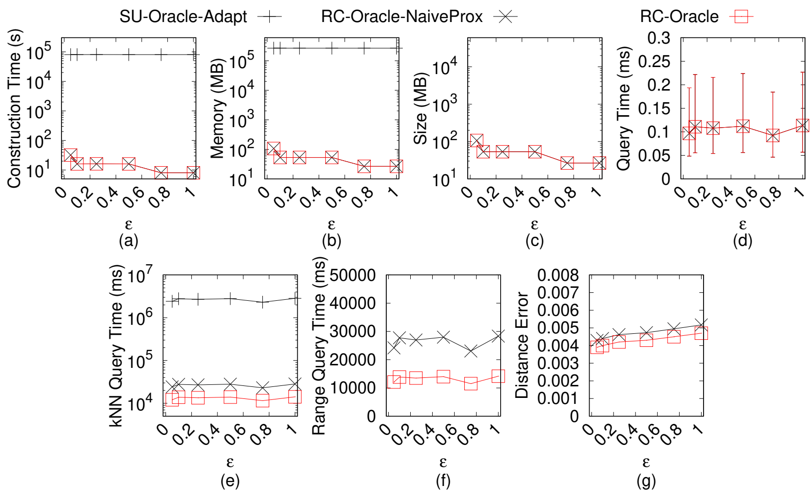

Fig. 108. Comparisons with other proximity queries oracles and variation oracles on RM_p point cloud dataset for the P2P query

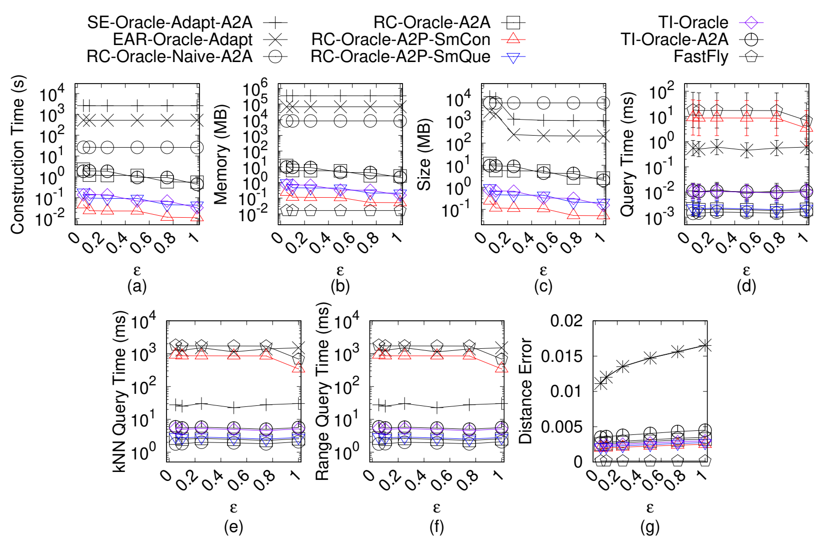

Fig. 109. Baseline comparisons (effect of ϵ on BH_p*-small* point cloud dataset for the A2P query)

construct the *TIN* using the point cloud at the beginning, so we substitute u with $N^2 \log N$, and *SE-Oracle-Adapt* needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt* is $O(N + nN^2 + n \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n) = O(nN^2 + n \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$.

Secondly, we show the *oracle size* of *SE-Oracle-Adapt*. The proof of the oracle size of *SE-Oracle-Adapt* is in [62, 63]. Thus, the oracle size of *SE-Oracle-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-Adapt*. The proof of the shortest path query time of *SE-Oracle-Adapt* is in [62, 63]. Thus, the shortest path query time of *SE-Oracle-Adapt* is $O(h^2)$.

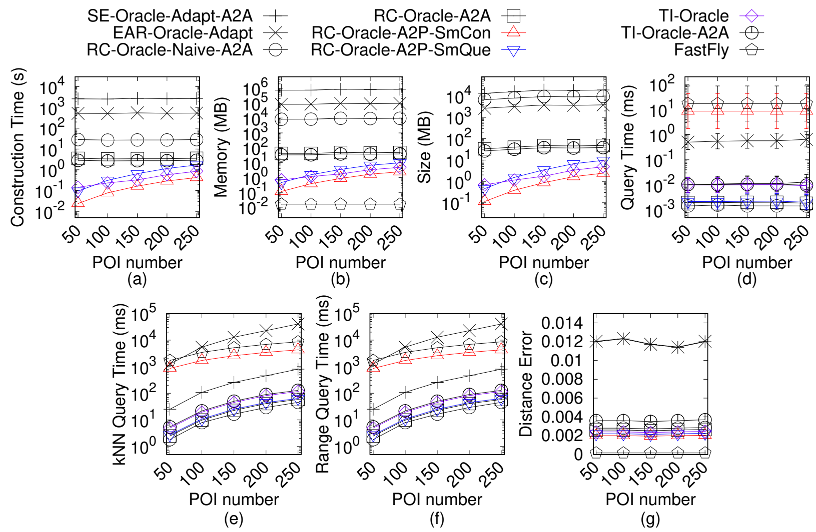

Fig. 110. Baseline comparisons (effect of n on BH_p-*small* point cloud dataset for the A2P query)

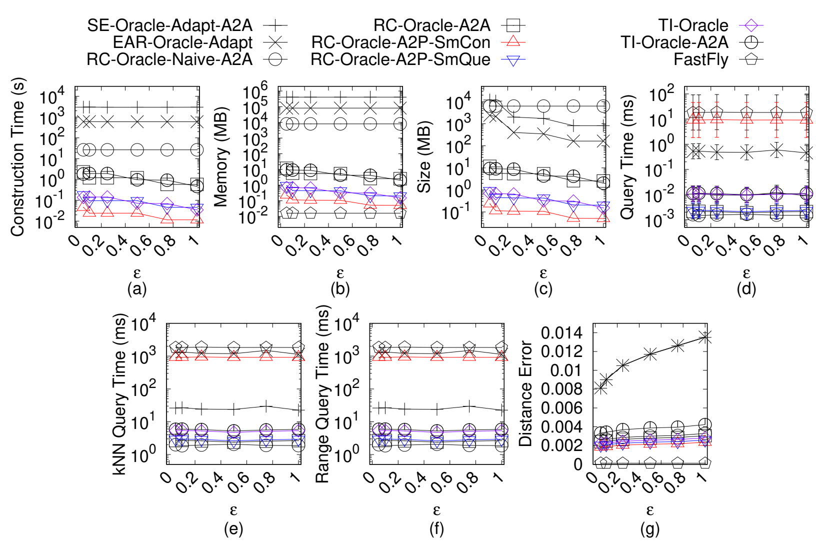

Fig. 111. Baseline comparisons (effect of ϵ on EP_p-*small* point cloud dataset for the A2P query)

Fourthly, we show the *error bound* of *SE-Oracle-Adapt*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt* is algorithm *DIO*, which returns the exact surface shortest path passing on T (that is constructed by the point cloud) according to Theorem E.1, so the error of *SE-Oracle-Adapt* is due to the oracle itself. Compared with $\Pi^*(s, t|T)$, the proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt* is in [62, 63]. 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 any pairs of POIs s and t in P. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on the point cloud, algorithm *SE-Oracle-Adapt* returns the approximate shortest path passing on a point cloud. $\square$

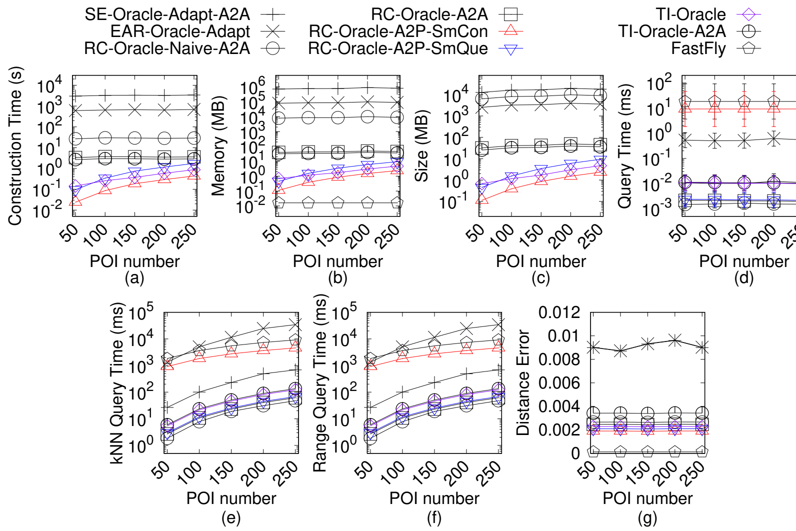

Fig. 112. Baseline comparisons (effect of n on *EP_p-small* point cloud dataset for the A2P query)

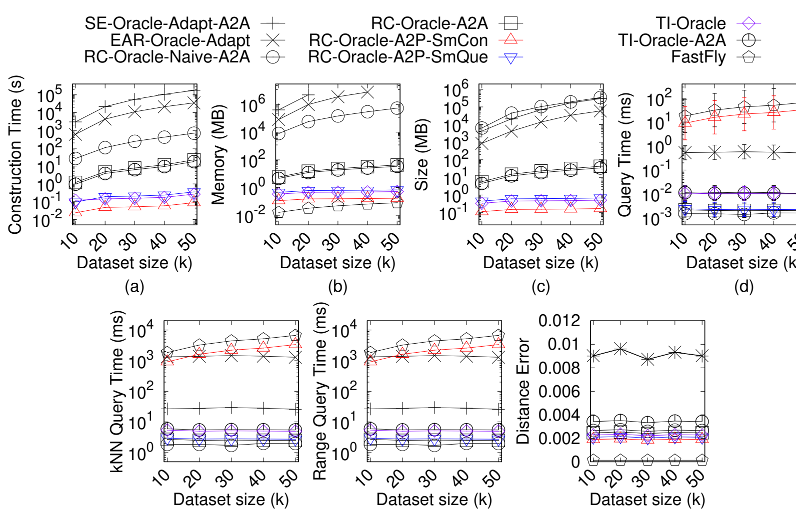

Fig. 113. Baseline comparisons (effect of N on *EP_p-small* point cloud dataset for the A2P query)

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

Proof. Firstly, we show the *oracle construction time* of *SE-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle in [62, 63] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N \log N)$ according to Theorem

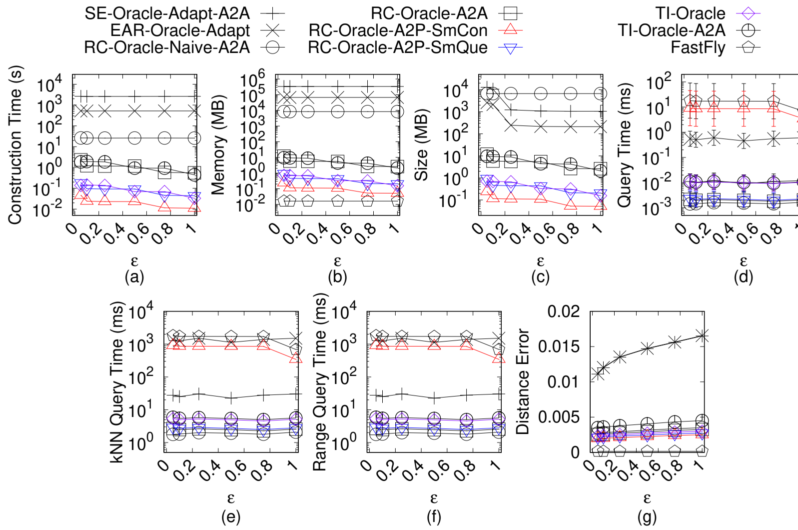

Fig. 114. Baseline comparisons (effect of ϵ on GF_p*-small* point cloud dataset for the A2P query)

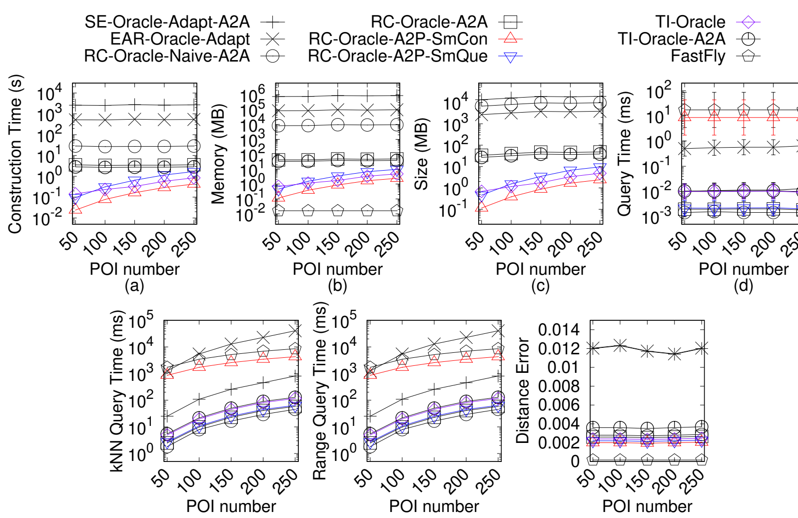

Fig. 115. Baseline comparisons (effect of n on GF_p*-small* point cloud dataset for the A2P query)

4.1. We substitute u with $N \log N$. Thus, the oracle construction time of *SE-Oracle-FastFly-Adapt* is $O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$.

Secondly, we show the *oracle size* of *SE-Oracle-FastFly-Adapt*. The proof of the oracle size of *SE-Oracle-FastFly-Adapt* is in [62, 63]. Thus, the oracle size of *SE-Oracle-FastFly-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-FastFly-Adapt*. The proof of the shortest path query time of *SE-Oracle-FastFly-Adapt* is in [62, 63]. Thus, the shortest path query time of *SE-Oracle-FastFly-Adapt* is $O(h^2)$.

Fourthly, we show the *error bound* of *SE-Oracle-FastFly-Adapt*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest

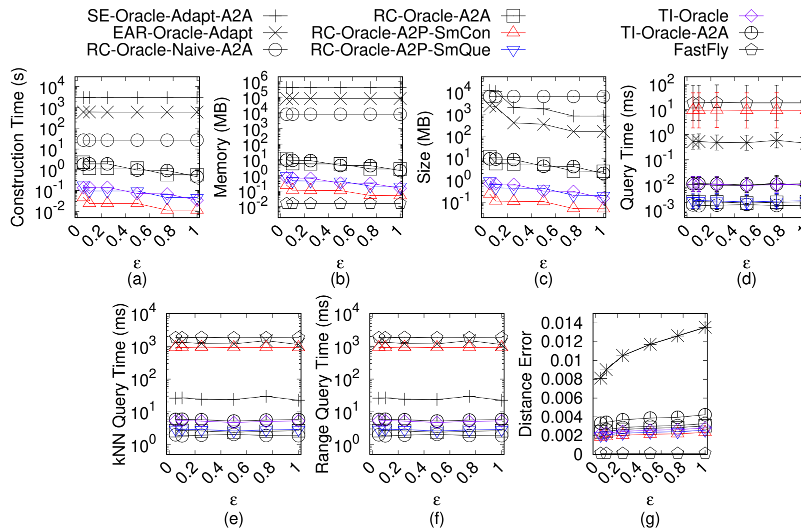

Fig. 116. Baseline comparisons (effect of ϵ on *LM_p-small* point cloud dataset for the A2P query)

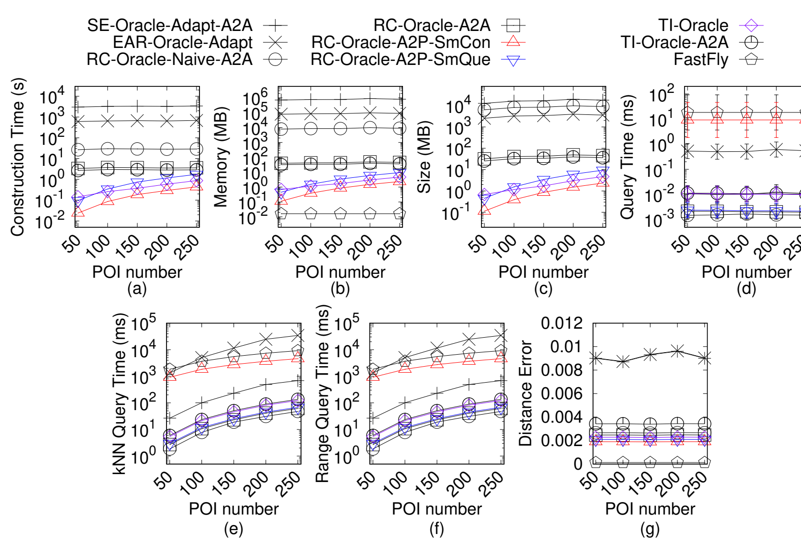

Fig. 117. Baseline comparisons (effect of n on *LM_p-small* point cloud dataset for the A2P query)

path passing on the point cloud according to Theorem 4.1, the error of *SE-Oracle-FastFly-Adapt* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-FastFly-Adapt* is in [62, 63]. So we obtain that *SE-Oracle-FastFly-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{\textit{SE-Oracle-FastFly-Adapt}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pairs of POIs s and t in P. $\square$

Theorem E.6. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-Adapt-A2A are* $O(\frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon})$ *and* $O(\frac{h^2}{\sin\theta\cdot\epsilon})$. *Compared with* $\Pi^*(s,t|T)$, *SE-Oracle-Adapt-A2A always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{\textit{SE-Oracle-Adapt-A2A}}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for any pairs of points s and t*

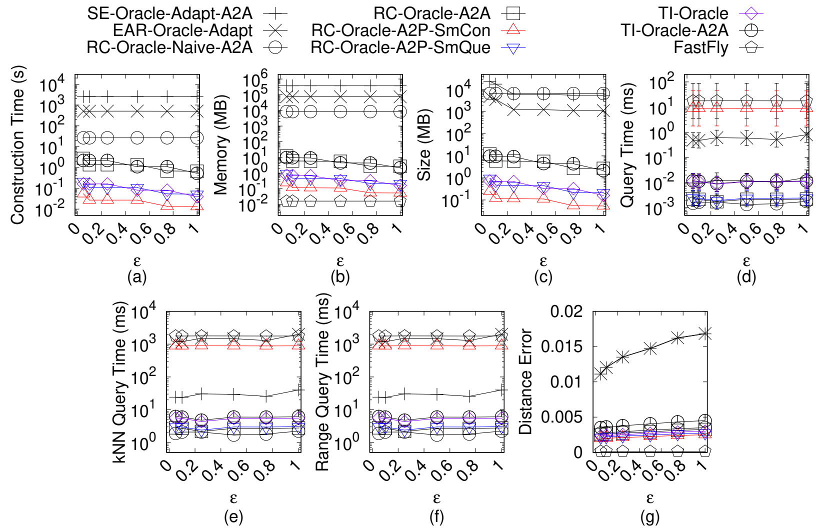

Fig. 118. Baseline comparisons (effect of ϵ on RM_p-*small* point cloud dataset for the A2P query)

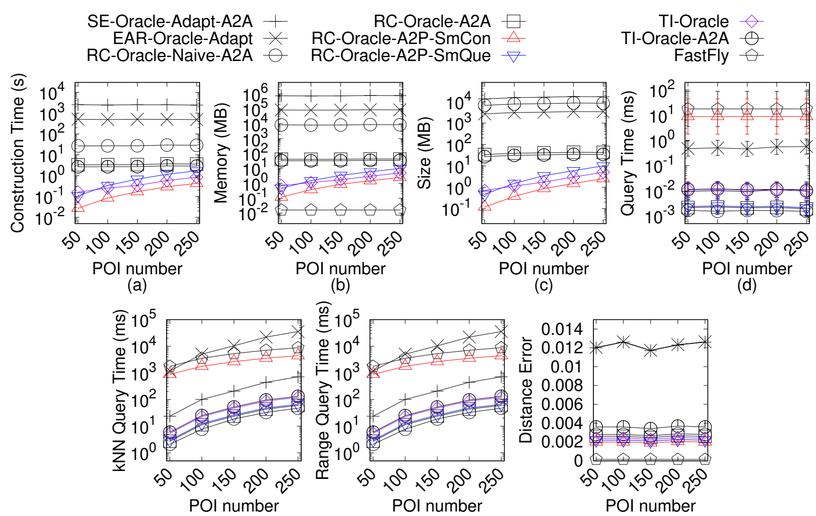

Fig. 119. Baseline comparisons (effect of n on RM_p-*small* point cloud dataset for the A2P query)

on T, where $\Pi_{SE\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)$ is the shortest surface path of SE-Oracle-Adapt-A2A passing on a TIN T (that is constructed by the point cloud) between s and t. Compared with $\Pi^*(s,t|C)$, algorithm SE-Oracle-Adapt-A2A returns the approximate shortest path passing on a point cloud.

Proof. Firstly, we show the *oracle construction time* of *SE-Oracle-Adapt-A2A*. We need to create POIs as Steiner points on faces of the constructed *TIN* by the given point cloud using the method in [62, 63] to adapt *SE-Oracle-Adapt* to *SE-Oracle-Adapt-A2A*, and there are $O(\frac{1}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ Steiner points per face. Since there are $O(N)$ faces, there are total $O(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ Steiner points. We substitute n of *SE-Oracle-Adapt* in Theorem E.4 with $\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$. But, we also need to construct

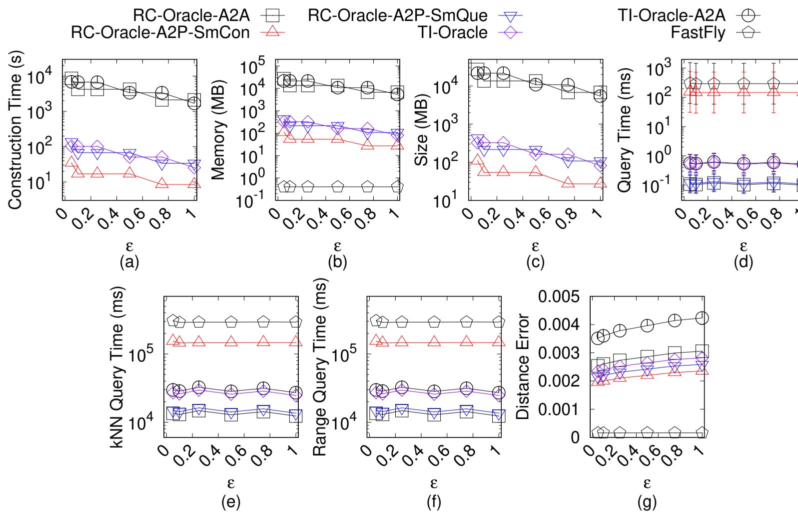

Fig. 120. Baseline comparisons (effect of ϵ on BH_p point cloud dataset for the A2P query)

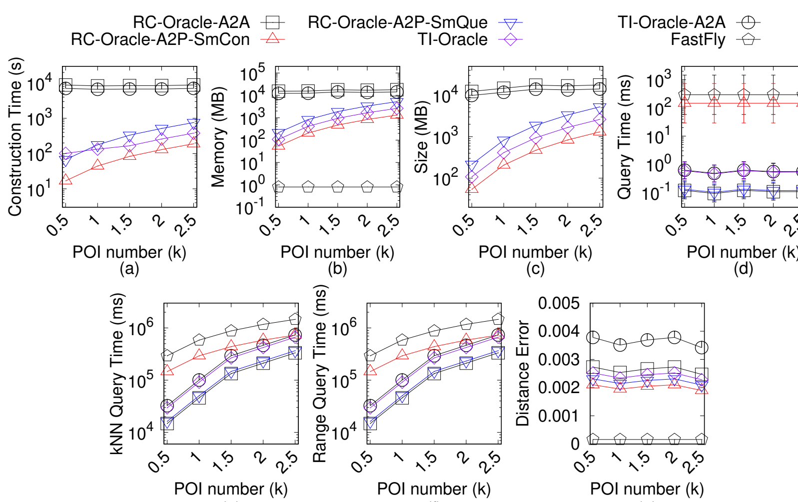

Fig. 121. Baseline comparisons (effect of n on BH_p point cloud dataset for the A2P query)

the *TIN* using the point cloud at the beginning, so *SE-Oracle-Adapt* needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt-A2A* is $O(N + \frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})) = O(\frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$.

Secondly, we show the *oracle size* of *SE-Oracle-Adapt-A2A*. We substitute n of *SE-Oracle-Adapt* in Theorem E.4 with $\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$. Thus, the oracle size of *SE-Oracle-Adapt-A2A* is $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-Adapt-A2A*. According to [62, 63], there are total $\frac{1}{\sin\theta\cdot\epsilon}$ possible pairs of sources and destinations, and the shortest path query time of each

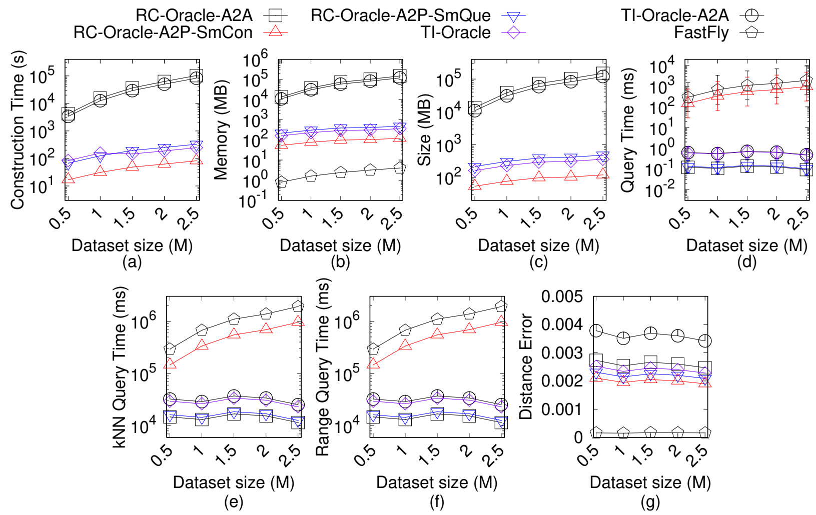

Fig. 122. Baseline comparisons (effect of N on BH_p point cloud dataset for the A2P query)

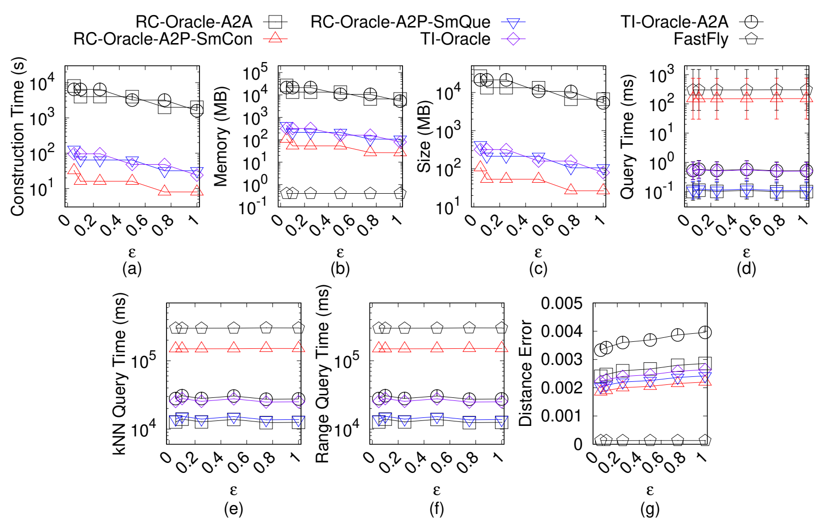

Fig. 123. Baseline comparisons (effect of ϵ on EP_p point cloud dataset for the A2P query)

pair of source and destination is $O(h^2)$. Thus, the shortest path query time of *SE-Oracle-Adapt-A2A* is $O(\frac{h^2}{\sin\theta\cdot\epsilon})$.

Fourthly, we show the *error bound* of *SE-Oracle-Adapt-A2A*. Since *SE-Oracle-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pairs of POIs s and t in P, according to the Steiner points placed in [62, 63], we obtain that *SE-Oracle-Adapt-A2A* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pairs of points s and t on C. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *SE-Oracle-Adapt-A2A* returns the approximate shortest path passing on a point cloud. □

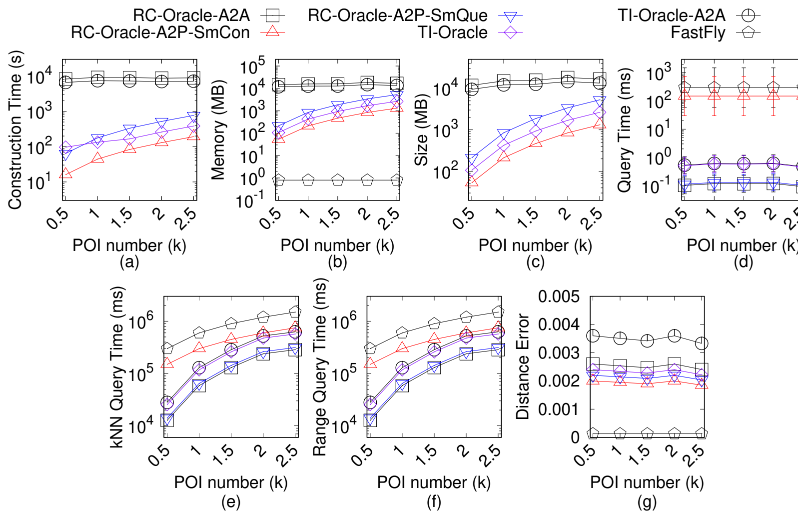

Fig. 124. Baseline comparisons (effect of n on EP_p point cloud dataset for the A2P query)

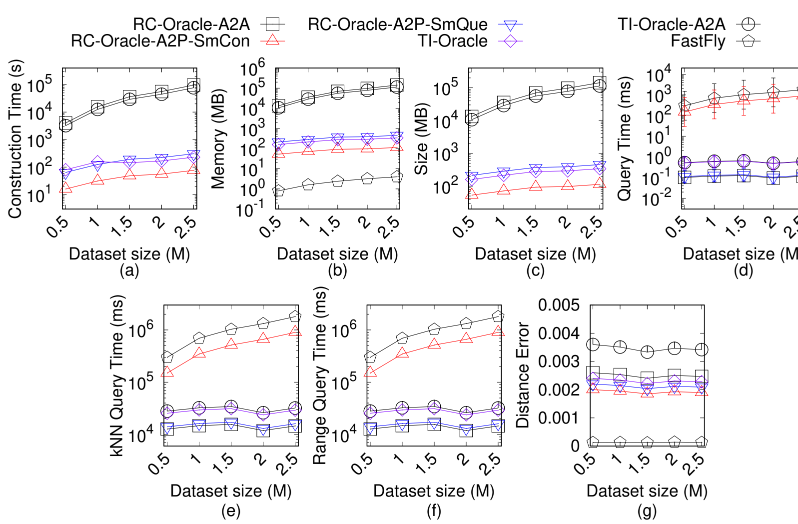

Fig. 125. Baseline comparisons (effect of N on EP_p point cloud dataset for the A2P query)

THEOREM E.7. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-FastFly-Adapt-A2A are* $O(\frac{N^2 \log N}{\sin\theta\cdot\sqrt{\epsilon}} \log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}} \log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}} \log\frac{1}{\epsilon} \log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}} \log\frac{1}{\epsilon}))$, $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}} \log\frac{1}{\epsilon})$ *and* $O(\frac{h^2}{\sin\theta\cdot\epsilon})$. *SE-Oracle-FastFly-Adapt-A2A always has* $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{\textit{SE-Oracle-FastFly-Adapt-A2A}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pairs of points s and t on C, where* $\Pi_{\textit{SE-Oracle-FastFly-Adapt-A2A}}(s,t|C)$ *is the shortest path of SE-Oracle-FastFly-Adapt-A2A passing on C between s and t.*

PROOF. Firstly, we show the *oracle construction time* of *SE-Oracle-FastFly-Adapt-A2A*. The oracle construction time of *SE-Oracle-Adapt-A2A* is $O(\frac{uN}{\sin\theta\cdot\sqrt{\epsilon}} \log\frac{1}{\epsilon} +$

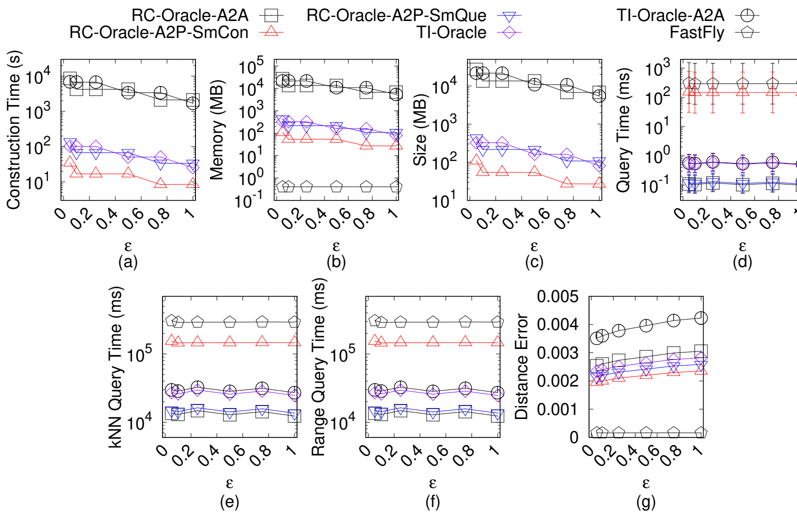

Fig. 126. Baseline comparisons (effect of ϵ on GF_p point cloud dataset for the A2P query)

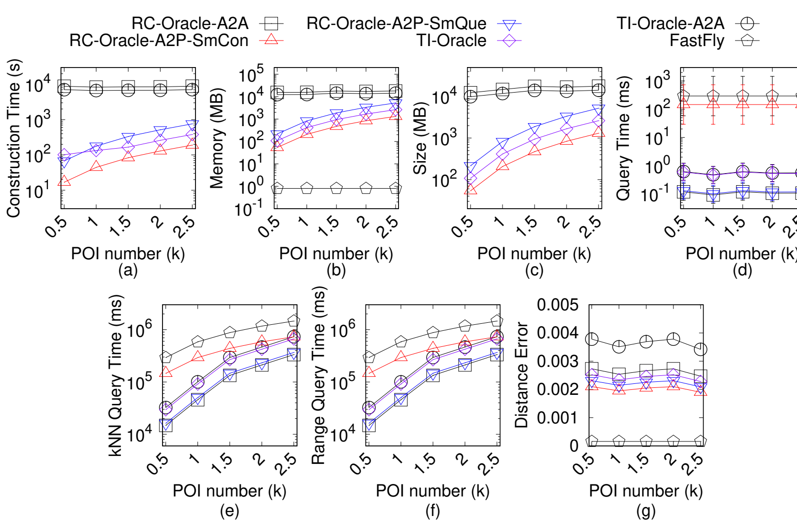

Fig. 127. Baseline comparisons (effect of n on GF_p point cloud dataset for the A2P query)

$\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, where u is the on-the-fly shortest path query time. In *SE-Oracle-FastFly-Adapt-A2A*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. Thus, the oracle construction time of *SE-Oracle-FastFly-Adapt-A2A* is $O(\frac{N^2\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$.

Secondly, we show the *oracle size* of *SE-Oracle-FastFly-Adapt-A2A*. The proof of the oracle size of *SE-Oracle-FastFly-Adapt-A2A* is the same as that of *SE-Oracle-Adapt-A2A*. Thus, the oracle size of *SE-Oracle-FastFly-Adapt-A2A* is $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon})$.

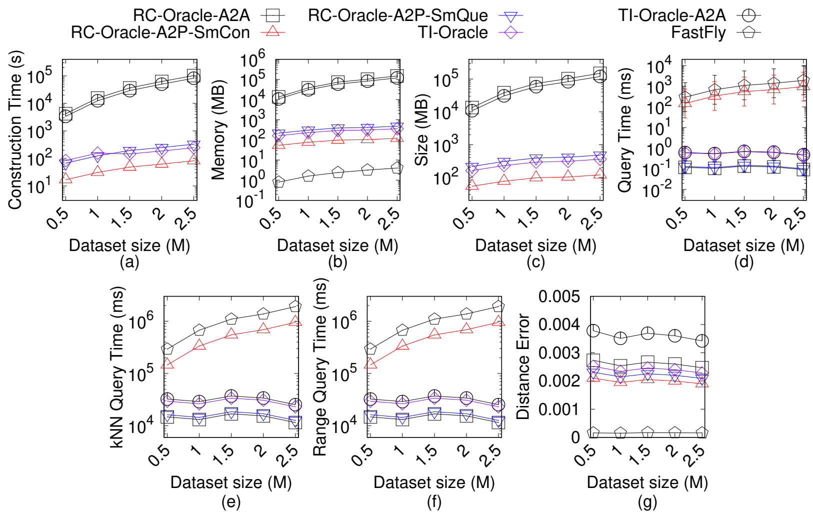

Fig. 128. Baseline comparisons (effect of N on GF_p point cloud dataset for the A2P query)

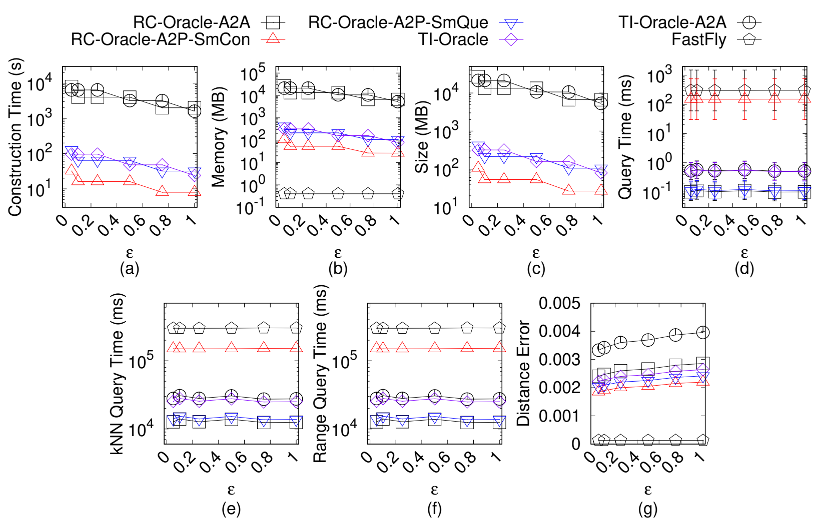

Fig. 129. Baseline comparisons (effect of ϵ on LM_p point cloud dataset for the A2P query)

Thirdly, we show the *shortest path query time* of *SE-Oracle-FastFly-Adapt-A2A*. The proof of the shortest path query time of *SE-Oracle-FastFly-Adapt-A2A* is the same as that of *SE-Oracle-Adapt-A2A*. Thus, the shortest path query time of *SE-Oracle-FastFly-Adapt-A2A* is $O(\frac{h^2}{\sin\theta\cdot\epsilon})$.

Fourthly, we show the *error bound* of *SE-Oracle-FastFly-Adapt-A2A*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-FastFly-Adapt-A2A* is algorithm *FastFly*, which returns the exact shortest path passing on the point cloud according to Theorem 4.1, the error of *SE-Oracle-FastFly-Adapt-A2A* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-FastFly-Adapt-A2A* is the same as that of *SE-Oracle-Adapt-A2A* on a *TIN*. So we obtain that *SE-Oracle-FastFly-Adapt-A2A* always has $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{\textit{SE-Oracle-FastFly-Adapt-A2A}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pairs of points s and t on C. □

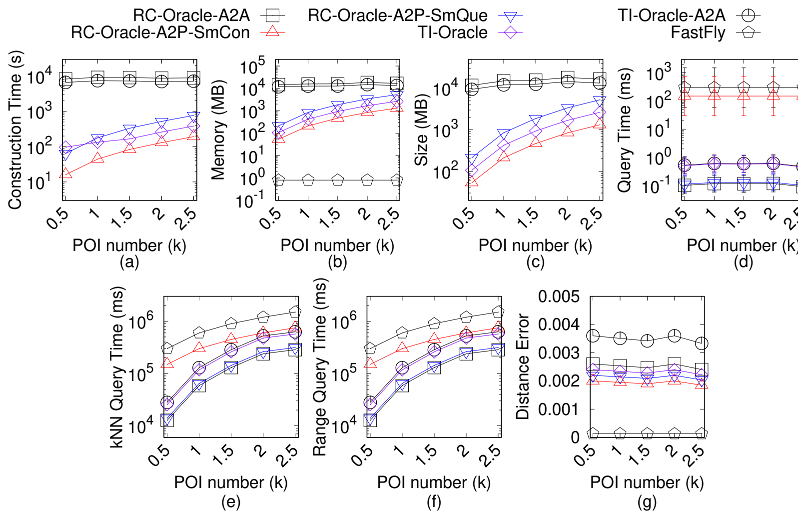

Fig. 130. Baseline comparisons (effect of n on LM_p point cloud dataset for the A2P query)

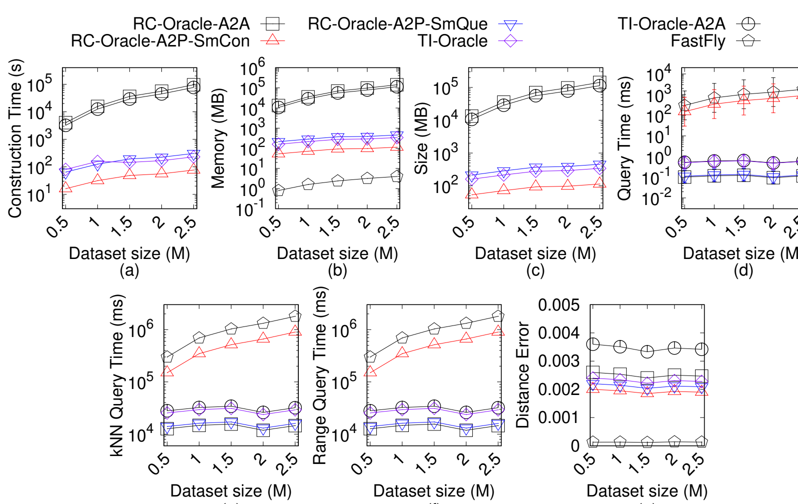

Fig. 131. Baseline comparisons (effect of N on LM_p point cloud dataset for the A2P query)

THEOREM E.8. *The oracle construction time, oracle size and shortest path query time of EAR-Oracle-Adapt are* $O(\lambda\xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m \log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ *and* $O(\lambda\xi\log(\lambda\xi))$. *Compared with* $\Pi^*(s,t|T)$, *EAR-Oracle-Adapt always has* $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)+2\delta|$ *for any pairs of points* s *and* t *on* T, *where* $\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)$ *is the shortest surface path of EAR-Oracle-Adapt passing on a TIN* T *(that is constructed by the point cloud) between* s *and* t *and* δ *is an error parameter [33]. Compared with* $\Pi^*(s,t|C)$, *algorithm EAR-Oracle-Adapt returns the approximate shortest path passing on a point cloud.*

PROOF. Firstly, we show the *oracle construction time* of *EAR-Oracle-Adapt*. The oracle construction time of the original oracle in [33] is $O(\lambda\xi m u + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, where u is

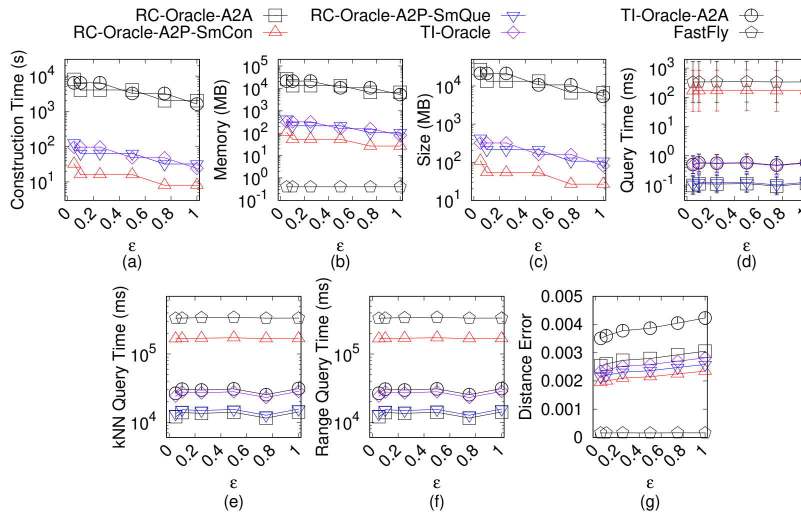

Fig. 132. Baseline comparisons (effect of ϵ on RM_p point cloud dataset for the A2P query)

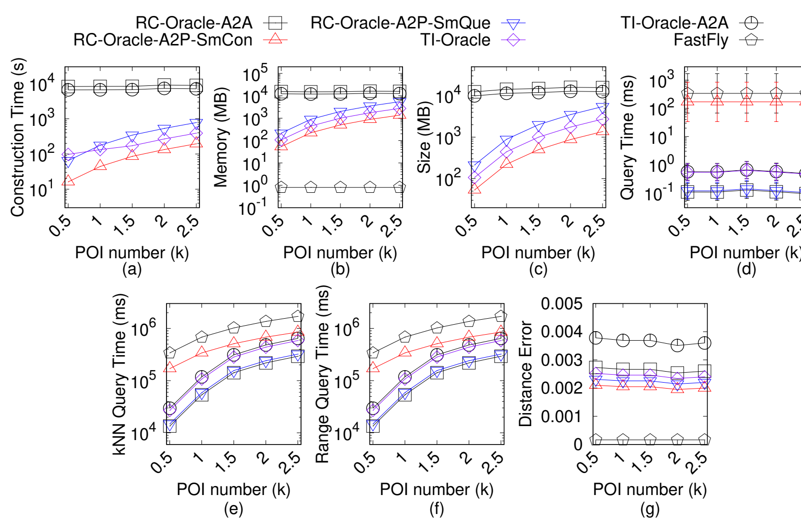

Fig. 133. Baseline comparisons (effect of n on RM_p point cloud dataset for the A2P query)

the on-the-fly shortest path query time. In *EAR-Oracle-Adapt*, we use algorithm *DIO* for the *TIN* shortest path query, which has the shortest path query time $O(N^2 + N \log N)$ according to Theorem E.1. But, we also need to construct the *TIN* using the point cloud at the beginning, so we substitute u with $N^2 + N \log N$, and *EAR-Oracle-Adapt* needs an additional $O(N)$ time for constructing the *TIN* using the point cloud. Thus, the oracle construction time of *EAR-Oracle-Adapt* is $O(N + \lambda\xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m \log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N) = O(\lambda\xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m \log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$.

Secondly, we show the *oracle size* of *EAR-Oracle-Adapt*. The proof of the oracle size of *EAR-Oracle-Adapt* is in [33]. Thus, the oracle size of *EAR-Oracle-Adapt* is $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

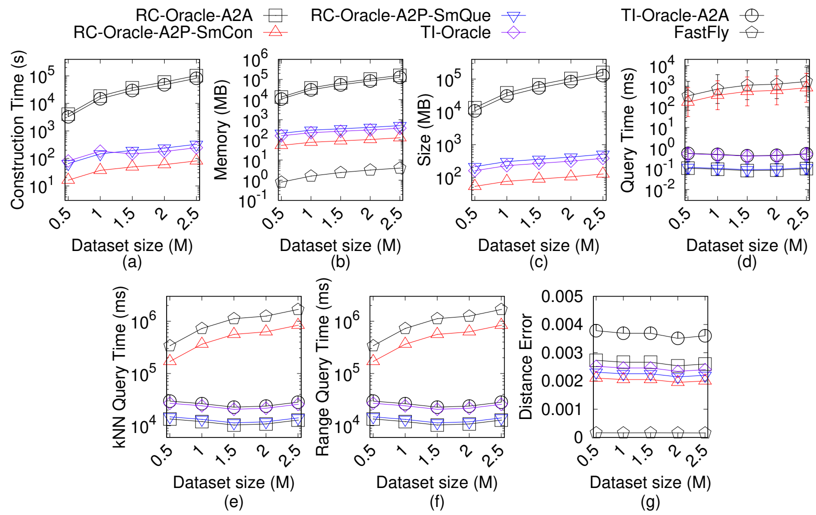

Fig. 134. Baseline comparisons (effect of N on RM_p point cloud dataset for the A2P query)

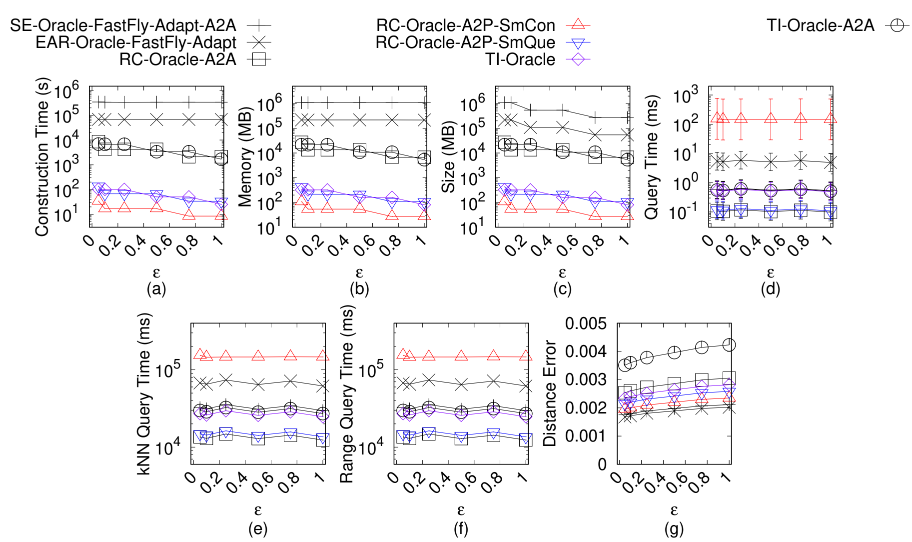

Fig. 135. Ablation study on BH_p point cloud dataset for the A2P query

Thirdly, we show the *shortest path query time* of *EAR-Oracle-Adapt*. The proof of the shortest path query time of *EAR-Oracle-Adapt* is in [33]. Thus, the shortest path query time of *EAR-Oracle-Adapt* is $O(\lambda\xi\log(\lambda\xi))$.

Fourthly, we show the *error bound* of *EAR-Oracle-Adapt*. Since the on-the-fly shortest path query algorithm in *EAR-Oracle-Adapt* is algorithm *DIO*, which returns the exact surface shortest path passing on T (that is constructed by the point cloud) according to Theorem E.1, so the error of *EAR-Oracle-Adapt* is due to the oracle itself. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of the oracle itself regarding *EAR-Oracle-Adapt* is in [33]. Since the *TIN* is constructed by the point cloud, we obtain that *EAR-Oracle-Adapt* always has $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)+2\delta|$ for

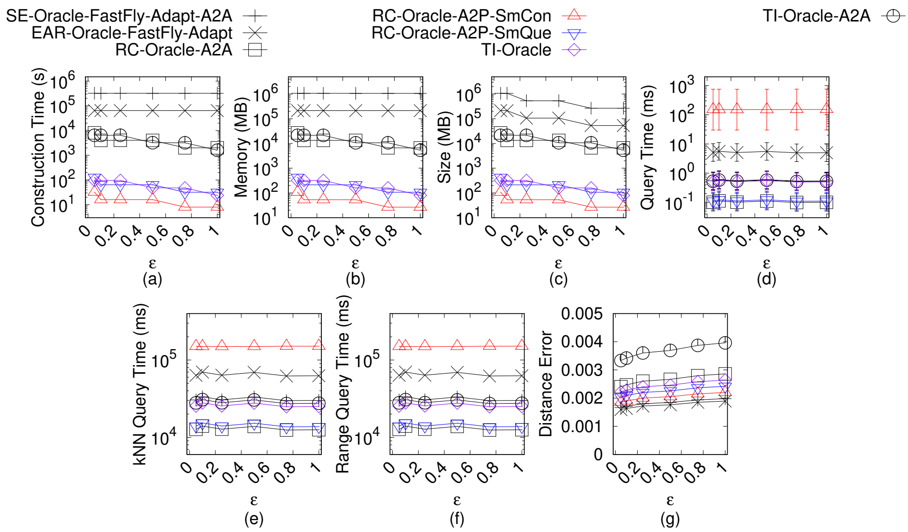

Fig. 136. Ablation study on EP_p point cloud dataset for the A2P query

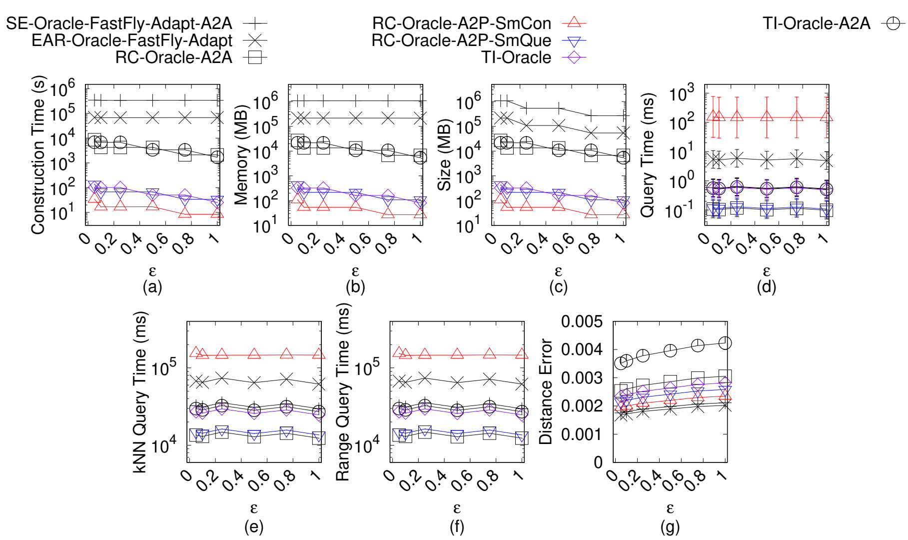

Fig. 137. Ablation study on GF_p point cloud dataset for the A2P query

any pairs of points s and t on T. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on the point cloud, algorithm *EAR-Oracle-Adapt* returns the approximate shortest path passing on a point cloud. □

Theorem E.9. *The oracle construction time, oracle size and shortest path query time of EAR-Oracle-FastFly-Adapt are* $O(\lambda\xi mN\log N + \frac{N\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ *and* $O(\lambda\xi\log(\lambda\xi))$. *EAR-Oracle-FastFly-Adapt always has* $|\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)| \le (1+\epsilon)|\Pi^*(s,t|C) + 2\delta|$ *for any pairs of points s and t on C, where* $\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)$ *is the shortest path of EAR-Oracle-FastFly-Adapt passing on C between s and t and δ is an error parameter [33].*

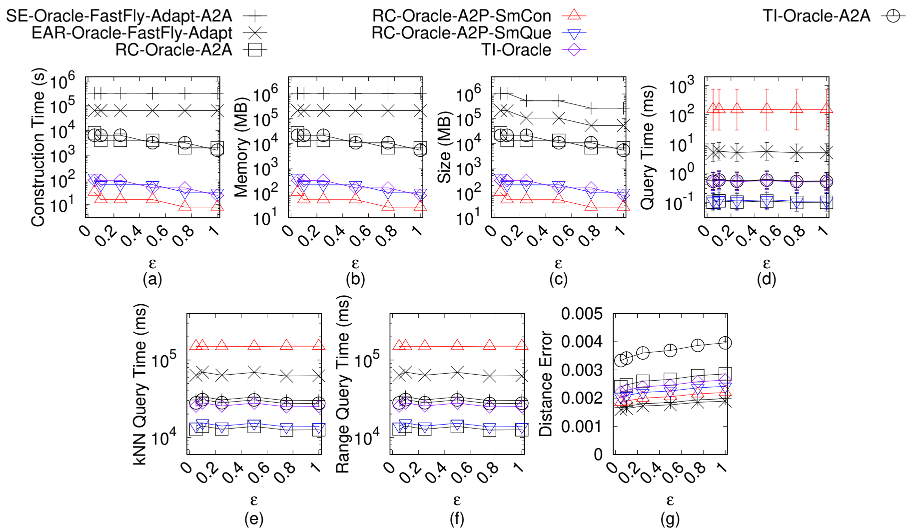

Fig. 138. Ablation study on LM_p point cloud dataset for the A2P query

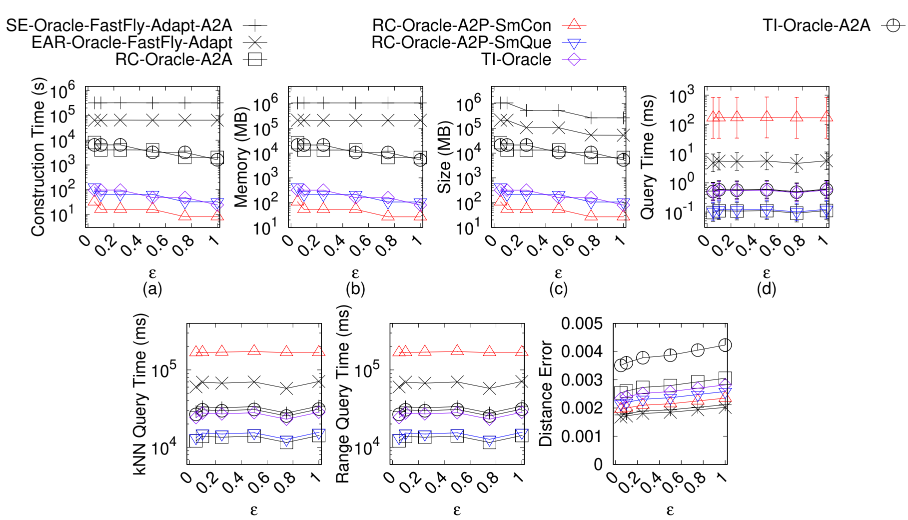

Fig. 139. Ablation study on RM_p point cloud dataset for the A2P query

Proof. Firstly, we show the *oracle construction time* of *EAR-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle in [33] is $O(\lambda\xi mu + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, where u is the on-the-fly shortest path query time. In *EAR-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. Thus, the oracle construction time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda\xi mN\log N + \frac{N\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$.

Secondly, we show the *oracle size* of *EAR-Oracle-FastFly-Adapt*. The proof of the oracle size of *EAR-Oracle-FastFly-Adapt* is in [33]. Thus, the oracle size of *EAR-Oracle-FastFly-Adapt* is $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

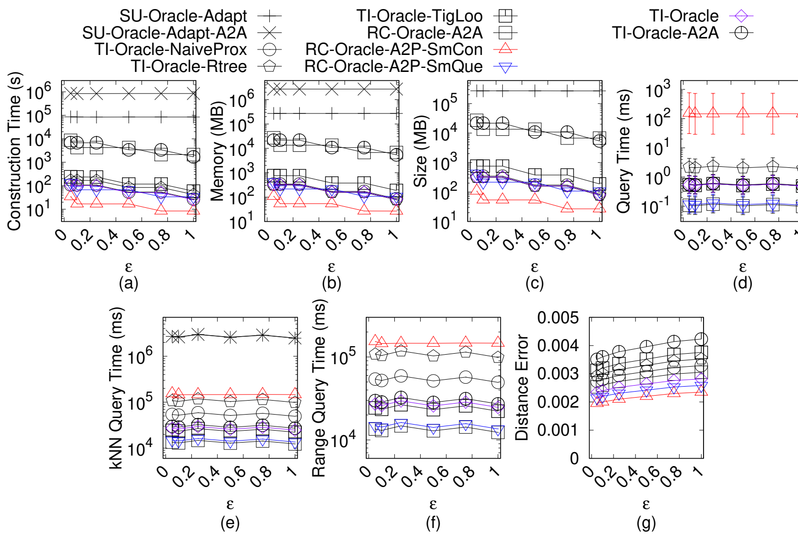

Fig. 140. Comparisons with other proximity queries oracles and variation oracles on BH_p point cloud dataset for the A2P query

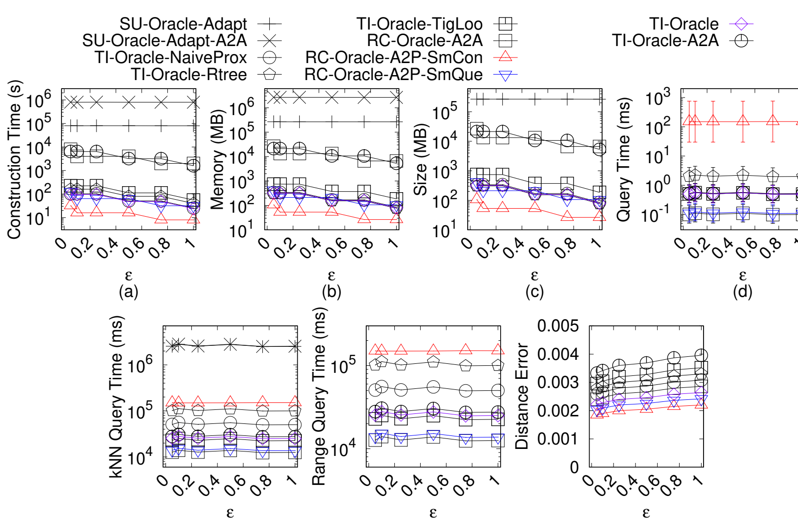

Fig. 141. Comparisons with other proximity queries oracles and variation oracles on EP_p point cloud dataset for the A2P query

Thirdly, we show the *shortest path query time* of *EAR-Oracle-FastFly-Adapt*. The proof of the shortest path query time of *EAR-Oracle-FastFly-Adapt* is in [33]. Thus, the shortest path query time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda\xi \log(\lambda\xi))$.

Fourthly, we show the *error bound* of *EAR-Oracle-FastFly-Adapt*. Since the on-the-fly shortest path query algorithm in *EAR-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the

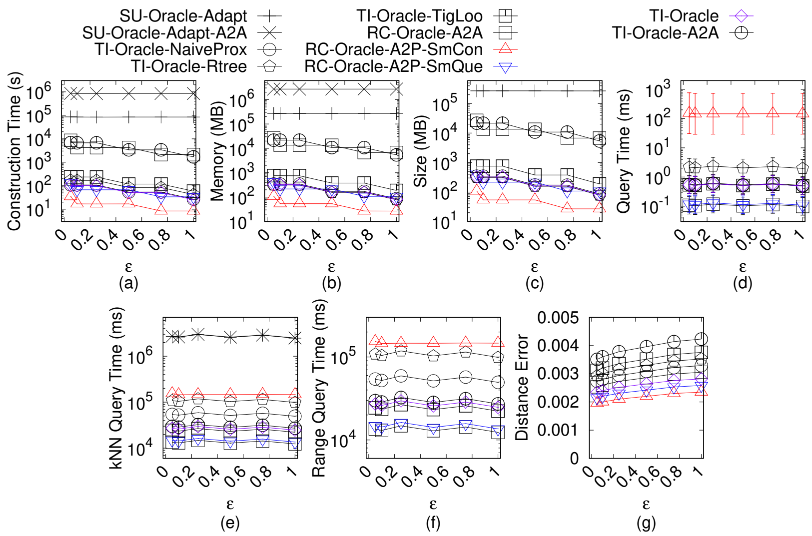

Fig. 142. Comparisons with other proximity queries oracles and variation oracles on GF_p point cloud dataset for the A2P query

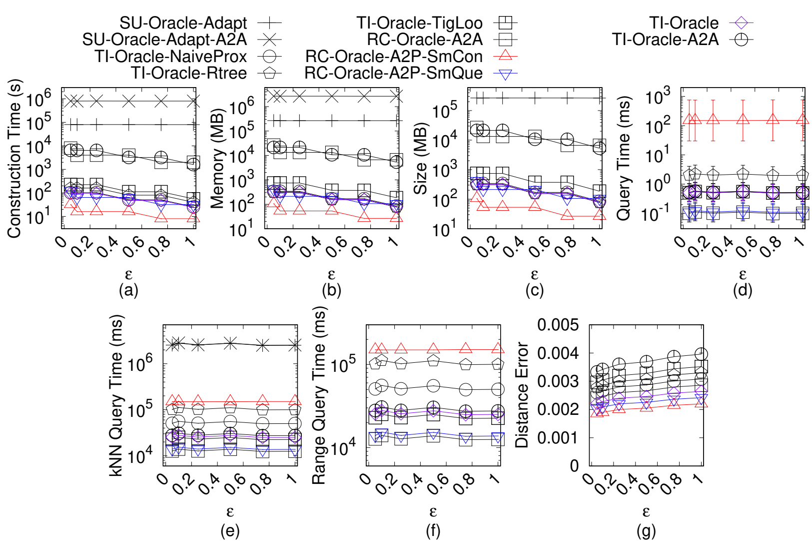

Fig. 143. Comparisons with other proximity queries oracles and variation oracles on LM_p point cloud dataset for the A2P query

exact shortest path passing on the point cloud according to Theorem 4.1, the error of *EAR-Oracle-FastFly-Adapt* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *EAR-Oracle-FastFly-Adapt* is in [33]. So we obtain that *EAR-Oracle-FastFly-Adapt* always has $|\Pi_{\textit{EAR-Oracle-FastFly-Adapt}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C) + 2\delta|$ for any pairs of points s and t in C. □

THEOREM E.10. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-Naive are $O(nN\log N + n^2)$, $O(n^2)$ and $O(1)$. RC-Oracle-Naive returns the exact shortest path passing on a point cloud.*

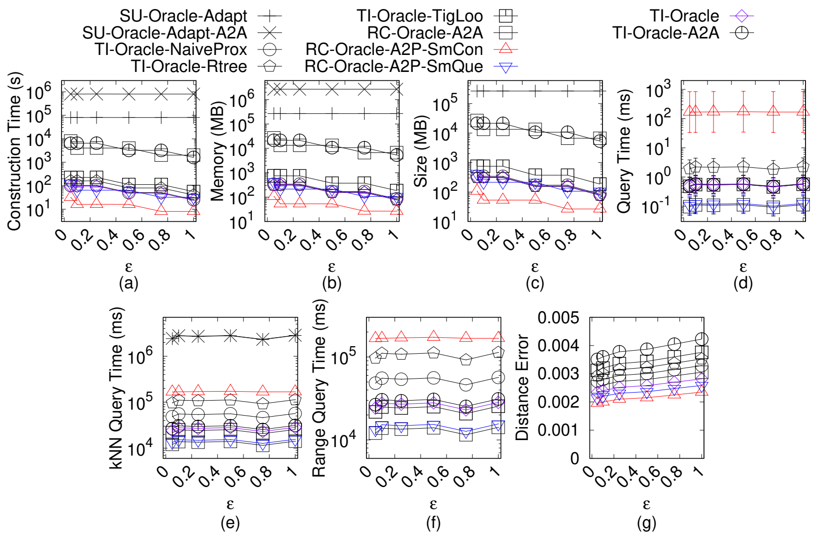

Fig. 144. Comparisons with other proximity queries oracles and variation oracles on RM_p point cloud dataset for the A2P query

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

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

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

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

Theorem E.11. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-Naive-A2A are $O(N^2 \log N)$, $O(N^2)$ and $O(1)$. RC-Oracle-Naive returns the exact shortest path passing on a point cloud.*

Proof. Firstly, we show the *oracle construction time* of *RC-Oracle-Naive-A2A*. Since we create POIs that have the same coordinate values as all points on the point cloud, we substitute n to N in the oracle construction time of *RC-Oracle-Naive*. Thus, the oracle construction time of *RC-Oracle-Naive-A2A* is $O(N^2 \log N)$.

Table 4. Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time	Oracle size	Shortest path query time	Error	Query type
Oracle-based algorithm					
SE-Oracle-Adapt [62, 63]	$O(nN^2 + n\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$ Large	$O(\frac{nh}{\epsilon^{2\beta}})$ Medium	$O(h^2)$ Small	Small	P2P
SE-Oracle-FastFly-Adapt [62, 63]	$O(nN\log N + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$ Medium	$O(\frac{nh}{\epsilon^{2\beta}})$ Medium	$O(h^2)$ Small	Small	P2P
SE-Oracle-Adapt-A2A [62, 63]	$O(\eta N^2 + \eta\log N + \frac{\eta h}{\epsilon^{2\beta}} + \eta h\log \eta)$ Large	$O(\frac{\eta h}{\epsilon^{2\beta}})$ Medium	$O(\frac{h^2}{\sin\theta\cdot\epsilon})$ Small	Small	P2P, A2P, A2A
SE-Oracle-FastFly-Adapt-A2A [62, 63]	$O(\eta N\log N + \frac{\eta h}{\epsilon^{2\beta}} + \eta h\log \eta)$ Medium	$O(\frac{\eta h}{\epsilon^{2\beta}})$ Medium	$O(\frac{h^2}{\sin\theta\cdot\epsilon})$ Small	Small	P2P, A2P, A2A
EAR-Oracle-Adapt [33]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \lambda\xi m\log N + \frac{\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$ Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ Large	$O(\lambda\xi\log(\lambda\xi))$ Medium	Small	P2P, A2P, A2A
EAR-Oracle-FastFly-Adapt [33]	$O(\lambda\xi mN\log N + \frac{N\log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$ Medium	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ Large	$O(\lambda\xi\log(\lambda\xi))$ Medium	Small	P2P, A2P, A2A
RC-Oracle-Naive	$O(nN\log N + n^2)$ Medium	$O(n^2)$ Large	$O(1)$ Small	Small	P2P
RC-Oracle-Naive-A2A	$O(N^2\log N)$ Medium	$O(N^2)$ Large	$O(1)$ Small	Small	P2P, A2P, A2A
RC-Oracle	$O(\frac{N\log N}{\epsilon} + n\log n)$ **Small**	$O(\frac{n}{\epsilon})$ **Small**	$O(1)$ **Small**	**Small**	**P2P**
RC-Oracle-A2P-SmCon	$O(\frac{N\log N}{\epsilon} + n\log n)$ **Small**	$O(\frac{n}{\epsilon})$ **Small**	$O(N\log N)$ **Medium**	**Small**	**P2P, A2P**
RC-Oracle-A2P-SmQue	$O(\frac{N\log N}{\epsilon} + n\log n)$ **Small**	$O(\frac{N}{\epsilon})$ **Medium**	$O(1)$ **Small**	**Small**	**P2P, A2P**
RC-Oracle-A2A	$O(\frac{N\log N}{\epsilon})$ **Small**	$O(\frac{N}{\epsilon})$ **Medium**	$O(1)$ **Small**	**Small**	**P2P, A2P, A2A**
TI-Oracle	$O(\frac{N\log N}{\epsilon} + Nn + n\log n)$ **Small**	$O(\frac{N}{\epsilon})$ **Medium**	$O(1)$ **Small**	**Small**	**P2P, A2P**
TI-Oracle-A2A	$O(\frac{N\log N}{\epsilon} + N\sqrt{N} + \sqrt{N}\log\sqrt{N})$ **Small**	$O(\frac{N}{\epsilon})$ **Medium**	$O(1)$ **Small**	**Small**	**P2P, A2P, A2A**
On-the-fly algorithm					
DIO-Adapt [64]	- N/A	- N/A	$O(N^2\log N)$ Large	Small	P2P, A2P, A2A
Kaul-Adapt [35]	- N/A	- N/A	$O(\gamma N\log(\gamma N))$ Large	Small	P2P, A2P, A2A
Dijk-Adapt [36]	- N/A	- N/A	$O(N\log N)$ Medium	Medium	P2P, A2P, A2A
FastFly	- **N/A**	- **N/A**	$O(N\log N)$ **Medium**	**No error**	**P2P, A2P, A2A**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [62, 63], $\eta = \frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$, θ is the minimum inner angle of any face in T, λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, $\gamma = \frac{l_{max}}{\epsilon l_{min}\sqrt{1-\cos\theta}}$, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T.

Secondly, we show the *oracle size* of *RC-Oracle-Naive-A2A*. Since we create POIs that have the same coordinate values as all points on the point cloud, we substitute n to N in the oracle size of *RC-Oracle-Naive*. Thus, the oracle construction time of *RC-Oracle-Naive-A2A* is $O(N^2)$.

Thirdly, we show the *shortest path query time* of *RC-Oracle-Naive-A2A*. The shortest path query time of *RC-Oracle-Naive-A2A* is the same as that of *RC-Oracle-Naive*. Thus, the shortest path query time of *RC-Oracle-Naive-A2A* is $O(1)$.

Fourthly, we show the *error bound* of *RC-Oracle-Naive-A2A*. The error bound of *RC-Oracle-Naive-A2A* is the same as that of *RC-Oracle-Naive*, so *RC-Oracle-Naive-A2A* returns the exact shortest path passing on the point cloud. □

Theorem E.12. *The oracle construction time, oracle size and kNN query time of SU-Oracle-Adapt are $O(n'N^2\log N)$, $O(n'N)$ and $O(n'N^2\log N)$. SU-Oracle-Adapt returns the exact kNN result.*

Proof. The proof of the oracle construction time, oracle size, *kNN* query time and error analysis of *SU-Oracle-Adapt* is in [58]. □

Table 5. Comparison of other proximity queries oracles and their variation oracles on a point cloud

Algorithm	Oracle construction time		Oracle size		*kNN* query time		Error	Query type
SU-Oracle-Adapt [58]	$O(n'N^2 \log N)$	Large	$O(n'N)$	Medium	$O(n'N^2 \log N)$	Large	No error	P2P, A2P
SU-Oracle-Adapt-A2A [58]	$O(N^2\sqrt{N} \log N)$	Large	$O(N\sqrt{N})$	Medium	$O(N^2\sqrt{N} \log N)$	Large	No error	A2A
RC-Oracle-NaiveProx	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(n')$	Small	Small	P2P
RC-Oracle	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(n')$	**Small**	**Small**	**P2P**
RC-Oracle-A2P-SmCon	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(N \log N + n')$	**Medium**	**Small**	**P2P, A2P**
RC-Oracle-A2P-SmQue	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P**
RC-Oracle-A2A	$O(\frac{N \log N}{\epsilon})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P, A2A**
TI-Oracle-NaiveProx	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle-Rtree	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle-TigLoo	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P**
TI-Oracle-A2A	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P, A2A**

Remark: n' is the number of query objects.

Theorem E.13. *The oracle construction time, oracle size and kNN query time of SU-Oracle-Adapt-A2A are* $O(N^2\sqrt{N} \log N)$, $O(N\sqrt{N})$ *and* $O(N^2\sqrt{N} \log N)$. *SU-Oracle-Adapt returns the exact kNN result.*

Proof. Since we select $\sqrt{N}$ points as POIs, we need to change n' to $\sqrt{N}$ in the oracle construction time, oracle size and *kNN* query time of *SU-Oracle-Adapt*, to obtain the theoretical analysis for *SU-Oracle-Adapt-A2A*. □

Theorem E.14. *The oracle construction time, oracle size and kNN query time of RC-Oracle-NaiveProx are* $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ *and* $O(n')$. *The error rate of the kNN query result by using RC-Oracle-NaiveProx is* $1 + \epsilon$.

Proof. The theoretical analysis proof of *RC-Oracle-NaiveProx* is the same as *RC-Oracle*. But, *RC-Oracle* performs better in terms of *kNN* query time in the experimental result. □

Theorem E.15. *The oracle construction time, oracle size and kNN query time of TI-Oracle-NaiveProx are* $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ *and* $O(n')$. *The error rate of the kNN query result by using TI-Oracle-NaiveProx is* $1 + \epsilon$.

Proof. The theoretical analysis proof of *TI-Oracle-NaiveProx* is the same as *TI-Oracle*. But, *TI-Oracle* performs better in terms of *kNN* query time in the experimental result. □

Theorem E.16. *The oracle construction time, oracle size and kNN query time of TI-Oracle-Rtree are* $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ *and* $O(n')$. *The error rate of the kNN query result by using TI-Oracle-Rtree is* $1 + \epsilon$.

Proof. The oracle construction time, oracle size and error rate of the *kNN* query result of *TI-Oracle-Rtree* is the same as *TI-Oracle*. For the *kNN* query time, since *TI-Oracle-Rtree* first uses R-tree to find the location of a query point in $O(\log n')$ time, and use the similar method as of *TI-Oracle* to find the final result in $O(n')$ time, so the *kNN* query time of *TI-Oracle-Rtree* is also $O(\log n' + n') = O(n')$. So, *TI-Oracle* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.17. *The oracle construction time, oracle size and kNN query time of TI-Oracle-TigLoo are* $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ *and* $O(n')$. *The error rate of the kNN query result by using TI-Oracle-TigLoo is* $1 + \epsilon$.

PROOF. The oracle size and error rate of the *kNN* query result of *TI-Oracle-Rtree* is the same as *TI-Oracle*. For the oracle construction time, *TI-Oracle-TigLoo* involves the construction of tight/loose surface indexes, but the result is the same as *TI-Oracle*. *TI-Oracle* performs better in terms of oracle construction time in the experimental result. For the *kNN* query time, although *TI-Oracle-TigLoo* can gradually expand in *kNN* queries to slightly reduce kNN query time, its result is still the same as *TI-Oracle*. □