ABSTRACT

Title of dissertation: High-Throughput Network Distance Computations for Spatial Analytics Inside Any Store

Shangfu Peng
Doctor of Philosophy, 2019

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

In the past decades, shortest distance computation methods for road networks have been developed that focus on how to speed up the latency of a single source-target pair distance query. Large analytic applications on road networks including simulations (e.g., evacuation planning), logistics, location-based advertisement, and transportation planning require methods that provide high throughput (i.e., distance computations per second) and the ability to "scale out" by using large distributed computing clusters. Although decreasing the latency time for one source-target query results in reducing the total response time for a spatial analytic query, it is far from enough since these methods don't take into account considerations such as cache results, query optimization, multi-threads, distributed systems, etc.

This thesis broadly expands on the use of the *distance oracle on road networks* to achieve above goals. In the first part, we present a new framework termed the *All-Store Distance Oracle (ASDO)* for large road networks and shows how to efficiently compute it for any large road network in a distributed cluster. The ASDO

representation is a well-separated pair decomposition (WSPD) of a road network using network distance instead of Euclidean distance. The ASDO representation benefits from the small size of the WSPD which enables the ASDO representation to answer ϵ-approximate network distance queries in a high-throughput rate and can be easily embedded within any database system including RDBMS, Column-oriented DBMS, and key-value stores. Experimental results show that the ASDO representation of the USA road network can be computed in a few hours using a modest size cluster. In comparison, previous database-centric approaches either do not scale to large road networks or are several orders of magnitude slower than the proposed ASDO for spatial queries.

In the second part , we show how useful the ASDO representation is in real applications evaluating two proposed architectures on a variety of spatial analytic queries in common use such as KNN, distance matrix, and trajectory queries. One architecture is our ASDO representation embedded in PostgreSQL, and the other one is a widely used hybrid architecture in industry. Embedding the ASDO representation inside PostgreSQL supports the performance of complex analytic queries on road networks using standard SQL. This makes the results of ASDO simple to use, yet considerably expressive, compared to traditional methods that require extensive development effort. Experimental results indicate that our ASDO architecture within PostgreSQL can compute more than $60K$ road distance operations per second on a large road network (e.g., USA), which achieves $20\times$ more throughput compared to the state-of-the-art shortest distance computation methods.

In the third part, as some applications require the ability to scale out on large

distributed computing clusters, a framework called SPDO is presented which implements an extremely fast distributed algorithm for computing spatial analytic queries on Apache Spark. The approach extends the ASDO representation which has now been adapted to use Spark's resilient distributed dataset (RDD). SPDO improves the throughput by at least two orders of magnitude, which makes the approach suitable for applications that need to compute millions of network distances per second.

Interviews with tens of related companies whom we deemed to be needy of performing some analytic queries on road networks led us to observe that they are usually concentrated in a local area spanning several cities, and need a high-throughput solution such as performing millions of shortest distance computations per second. In the forth part, we first demonstrate a solution, termed City Distance Oracles (CDO) to achieve as many as 7 million shortest distance computations per second per commodity machine on a city road network. Next, we extend CDO to yield a new distance oracle system (DOS) for general road networks. It can solve most spatial analytic queries, and its throughput achieves $5M$ distance computations per second even on the whole USA road network. In addition, a $10K \times 10K$ origin-distance (OD) matrix can be computed in 20 seconds.

High-Throughput Network Distance Computations for Spatial Analytics Inside Any Store

by

Shangfu Peng

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Hanan Samet, Chair/Advisor
Professor Shunlin Liang, Dean's rep
Professor David Mount
Professor Udaya Shankar
Professor Ramani Duraiswami

Dedication

This thesis is dedicated to Yi Huang, Jian Xu, and Shuguo Peng.

Acknowledgments

I would like to express my gratitude to all the people who helped me during my Ph.D. journey at UMD. These people have made this thesis possible and made my graduate experience a lifelong benefit.

First of all, I want to thank my advisor, Professor Hanan Samet for giving me the opportunity to work on a very diverse set of challenging projects over the past five years, which made me know the big picture and trend of this area. He has always been available for help and advice when I approached him in his office. I benefited a lot from insightful technical discussions with him such as how to pick a significant research topic, how to learn from related work, how to utilize the traditional methods to improve new methods, how to improve my ideas and papers, and so on. I also remember that he encouraged me to continue my thesis topic even when my first papers were rejected several times at the top database and GIS conferences. It has been a pleasure to work with and learn from such a distinguished individual.

I would also like to thank my co-advisor Dr. Jagan Sankaranarayanan, who was my initial senior labmate. His Ph.D. thesis work introduced the fundamental theory at the fundation of my research. Many ideas in this dissertation could not have been executed so well without his help. Counting from my second year in the Ph.D. program, we have more than one thousand email conversations. Without his extraordinary theoretical ideas, the completion of this thesis might have been delayed several years. Furthermore, the greatest experience of my Ph.D. life was in April 2016 when he, Prof. Hanan Samet, and I participated in the NSF I-Corps

program. We interviewed more than one hundred representatives of companies to better understand the needs for my research and identify the valuable product opportunities. I gained much industrial experience from it.

My thanks also go to my thesis committee members, Udaya Shankar, Ramani Duraiswami, David Mount, Shunlin Liang, and my preliminary oral exam committee member Larry Davis, for agreeing to serve on my committee and for sparing their valuable time reviewing the manuscript. They extremely supported my research work and gave me many helpful comments on my proposal and thesis.

I am fortunate to work with my labmates, Sarana Nutanong, Brendan C. Fruin, Marco D. Adelfio, Hao Li, and Hong Wei in no particular order. Although our research topics were very different, we usually discussed ideas for solving the problems in different areas. We also cooperated to maintain some lab systems and demos such as the NewsStand, TwitterStand, PhotoStand, etc. Sarana Nutanong, Brendan C. Fruin, and Marco D. Adelfio helped me know more about our lab when I arrived. They introduced much of their prior research results to me. Hao Li and Hong Wei are people who study with me the longest time during these years. The three of us participated in the 2016 ACM GIS Cup competition where we came in third place.

My colleagues at the computer science department have enriched my graduate life in many ways and deserve a special mention. Chang Liu showed me how to study and cooperate with different professors. Ang Li was an important teammate of mine for the ACM/ICPC 2013 World Finals. Hui Miao introduced a lot of industrial experiences to me. Ruofei Du helped me develop a map interface to

explore interesting images from Instagram.

I owe my deepest thanks to my family - my mother and father who have always stood by me. They have pulled me through against impossible odds at times. My wife, Yi Huang, who gives me her unlimited support and encouragement. She let me step away from the triviality and focus on research. She makes my life abroad more colorful and joyful. Words are not enough to express my gratitude to them. Thank you all for steering me to a better self.

Finally, I would like to acknowledge financial support from the UMD Computer Science Department and the National Science Foundation (NSF).

It is impossible to remember all, and I apologize to those whom I've inadvertently omitted.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

The past two decades have seen a steady increase in processing spatial queries. Such functionality has been implemented in early systems such as QUILT [59,68] and SAND [38,58] which had a browsing capability to full-fledged mapping applications such as MapQuest, Yahoo Maps, Google Maps, and Bing Maps. The most common interactions with a map invokes the shortest route, distance, travel time, or a simple query such as "finding my k nearest restaurants?" [60] or in a specific region (e.g., [23, 24]). They all require the computation of the distance between two locations x and y which in our work is more accurately represented as the shortest network distance $d_G(x, y)$ instead of the Euclidean distance $||x - y||$, or variants of it such as a minimum distance to a block boundary (e.g., [56, 62]) or the Hausdorff distance (e.g., [46]).

To make the discussion more general, we introduce some basic concepts about the *spatial network* and the *spatial analytic query*. A spatial network G is modeled as a weighted directed graph denoted by $G(V, E, w, p)$, where V is a set of nodes or vertices, $n = |V|$, $E \subset V \times V$ is the set of edges, $m = |E|$, and w is a weight function that maps each edge $e \in E$ to a positive real number $w(e)$, e.g., distance or time.

Without loss of generality, for each node v, $p(v)$ denotes the spatial position of v with respect to a spatial domain S, which is also referred to as an embedding space (e.g., a reference coordinate system in terms of latitude and longitude). In this thesis, all discussion is for a 2-dimensional space S, but note that it is straightforward to extend our results to d-dimensional space. Given nodes u and v, we define the network distance $d_G(u, v)$ to be the shortest distance from u to v in the spatial network, and $d_E(u, v)$ to be the Euclidean distance or geodesic distance from u to v. In addition, we introduce two values γ_L and γ_H termed the minimum and maximum distortions of G as follows.

$$\gamma_L = \min_{u,v \in V} \frac{d_G(u, v)}{d_E(u, v)} \qquad \gamma_H = \max_{u,v \in V} \frac{d_G(u, v)}{d_E(u, v)} \tag{1.1}$$

We assume that for some spatial networks (e.g., road networks), γ_L and γ_H are two constants, albeit γ_H may be large.

Beyond simple navigation queries, location-based web services like Google Maps repeatedly pose queries on a road network and utilize the results to serve a user base. For example, Google Distance Matrix offers an API that computes the distance matrix between a set of origins and a set of destinations. Other examples include analysts who use OLAP stores to perform complex simulations on road networks to help answer queries such as determining where to locate an additional Walmart among a number of potential locations, or the roads where bottlenecks exist for evacuation planning purposes. Moreover, mobile services frequently interact with write-optimized stores to store the current positions of mobile hosts as they move about in a road network. These services also frequently compute the

distances from their mobile hosts to other mobile hosts or landmarks in order to provide services such as locating the k nearest restaurants or gas stations. We use the term *spatial analytic queries* to collectively describe such queries. The challenge lies in taking note of the realization that each such instance of a spatial analytic query invariably involves being able to perform hundreds to as many as millions of computations of distance along a spatial network rather than as the crow flies.

In the face of a massive amount of spatial analytic queries from internet scale users, for example, Google Maps [8] drastically restricts the number of shortest distance results per query (e.g., a limit of 625 (25×25 origin-destination matrices) shortest distances per query using the Google Distance Matrix API even to their paying customers). Most other existing services such as Yelp just use Euclidean distance instead of network distance. Figure 1.3 illustrates the drawback of using Euclidean distance in Google Maps and Yelp, respectively. Although Figure 1.3(a) uses an old version of Google Maps, the ordering problem still exists in the current version. The geodesic distance ordering of results is very different from the network distance ordering of results. Figure 1.3(b) shows Yelp's response to the query: find the restaurants around River Road, Edgewater, NJ (blue icon) with the distance filter that they are within a 2 mile biking distance. Obviously, the 5^{th} and 9^{th} results that lie on the other side of the river are impossible to reach by biking less than 2 miles. However, they are in the result set (e.g., Flat Top, the 5^{th} result, is 1.3 miles away using Euclidean distance).

Clearly, using geodesic or Euclidean distance to approximate network distance can produce significant errors. To measure how big the difference is, we tabulated

ratios of network distance to geodesic distance in some regions in Figure 1.1, termed the *route directness spectrum* (RDS). It shows the distortion resulting from approximating network distance with geodesic distance. In particular, a measure that is of immense interest to transportation planners is the *route directness index* (RDI) [45]. The RDI of any two locations in the road network is the ratio between the shortest network distance to the geodesic distance. We proposed the route directness spectrum as a distribution of the RDI, which plots as a proportion of the total $O(n^2)$ shortest paths for a road network. Figure 1.1 was produced by bucketing and counting up the $O(n^2)$ ratio values of the route directness index located in $[x, x + 0.1)$, where x is a point on the x-axis. For each bucket, we compute the percentage of each group as a percentage of the total number of shortest paths. Note that the route directness index must be larger than 1.0 since the geodesic distance is always less than or equal to the network distance.

Figure 1.1 shows the route directness spectrum of New York City (NYC), the Bay Area (Bay), and Salt Lake City (SLC) road networks, respectively, from which it is easy to see that NYC has a higher road network connectivity than the Bay Area or SLC as its road directness spectrum is skewed more towards one (i.e., a larger proportion of the location pairs have a route directness index close to one). The result comes up to what we expect as we know that most streets in NYC are laid out on a grid, and the lengths of the side of the blocks are relatively small in contrast with Salt Lake City that long blocks are made.

However, even for NYC, a well-connected road network, 50% of the distance queries will have an error of 20% or more by approximating using geodesic distance.

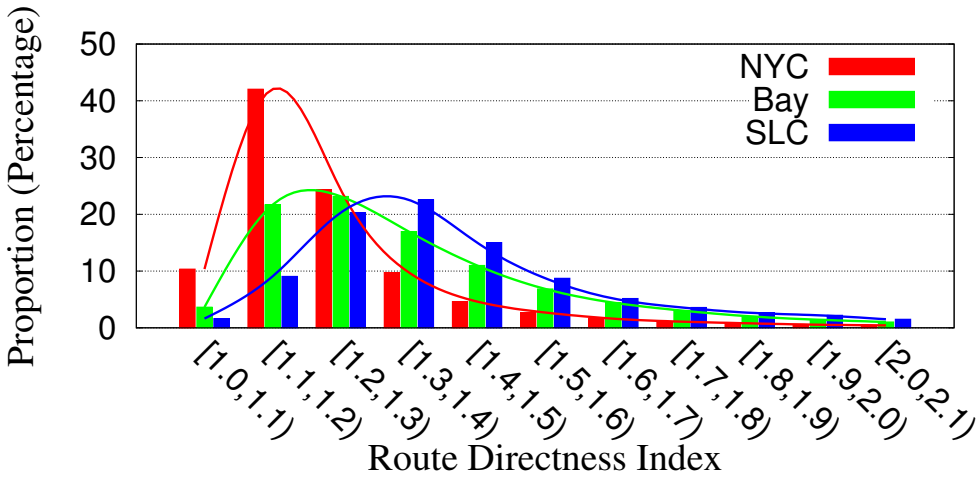


Figure 1.1: Route directness spectrum (RDS) of New York City (NYC), the Bay Area (Bay), and Salt Lake City (SLC). In contrast, the maximum RDI, corresponding to the maximum ratio of network distance to geodesic distance, is 10.6 in NYC, 30.4 in Bay, and 26.3 in SLC; the average RDI is 1.213 in NYC, 1.384 in Bay Area, and 1.475 in SLC.

Considering the ordering example in Figure 1.3 and the results of the route directness spectrum in Figure 1.1, we conclude that actual network distances are well worth computing in spatial analytic applications.

1.2 Distance Oracle Representation

Reviewing previous research work, we find none that are concerned with general spatial analytic queries. Instead, they focus on speeding up one specific type of query, e.g., KNN search queries [32, 36, 47, 50], CNN queries [31], and distance matrix [41]. However, these algorithms are not easy to extend to include general spatial analytic queries. On the other hand, most state-of-the-art methods such as

HL [21], TNR [25], CH [39], etc, focus on decreasing the latency time for a single source-target (s-t) query, which is the basic unit of a spatial analytic query. Although decreasing the latency time for one s-t query results in reducing the total response time for a spatial analytic query, it is far from enough since these methods don't take into account considerations such as cache results, multi-threads, and distributed systems that can be used to speed up a spatial analytic query.

An alternative approach to speeding up spatial analytic queries is to take advantage of the query optimizer associated with the database system which makes use of selectivity factors about the underlying data (i.e., stored in the relations being operated on). For example, suppose that we want to find the cities with population greater than $500,000$ within 200 miles of the Mississippi River. We have two options here. Armed with knowledge about the complexity of executing a within (or buffer) algorithm as well as the data distribution, the query optimizer can either call for performing the spatial selection first or the relational selection first where the choice will depend on the number of cities with such a population and and the size of the spatial area in question. As another example, suppose that we want to find all stores within 25 driving miles of a warehouse. Here, armed with knowledge of the complexity of finding entities within a given driving distance as well as the data distribution, the query optimizer can either call for a solution based on finding the nearby stores vis-a-vis the individual warehouses or on finding the nearby warehouses vis-a-vis the individual stores where the choice depends on the number of warehouses and the number of stores. The important thing to note about these examples is that the query optimizer requires knowledge about the complexity of an external module

or algorithm that is executed outside the database. Unfortunately, such knowledge is usually not present and thus users cannot rely on it.

The first attempts to answer spatial queries within a database system are the ϵ-distance oracle (ϵ-DO) [64] and PCPD [67] methods, the previous work of our research group, for approximately estimating the network distance and path between any two vertices of a spatial network. It proposed the *distance oracle representation* such as Figure 1.2 based on the notion of *spatial coherence*, which can be described intuitively as follows. Consider two cities A (e.g., Washington, DC) and B (e.g., Boston, MA) which are really the sets of vertices that are in the cities such that A and B are far away from each other but the diameters of A and B (i.e., the maximum distances between two locations in Washington, DC) are significantly smaller than the distance between the two cities A and B. If this property holds, then the network distance between any vertex in A and any vertex in B will be more or less similar, and hence can be approximated by a single value. Furthermore, all the shortest paths between a source in A and a destination in B will likely pass through a single common vertex.

Formally, both ϵ-DO and PCPD describe a well-separated pair decomposition (WSPD) [28] of a road network in order to produce well-separated pairs (e.g., (A, B) with particular network distance or shortest path properties). Two sets of vertices A and B are said to be well-separated if the minimum distance between any two vertices in A and B is at least $s \cdot r$, where $s > 0$ is a separation factor and r is the larger diameter of the two sets. The pair (A, B) is termed a *well-separated pair* (WSP).

Figure 1.2: A dumbbell pair of the distance oracle representation in Silver Spring, MD, which shows the notion of spatial coherence. It includes the shortest paths between all pairs of vertices in A and B are marked in a darker shade. The $30,000$ shortest paths pass through a single vertex.

The WSPD can be implemented as a PR quadtree where the decomposition rule is such that a block is split until all of the vertices within it are in the same element of a well-separated pair WSP. In particular, ϵ-DO specifies an approximate network distance oracle of size $O(\frac{n}{\epsilon^2})$ of WSPs $(A, B, d_\epsilon(A, B))$ such that A and B are the blocks in a quadtree with the property that for any pair of vertices (s, t), $s \in A$ and $t \in B$, $d_\epsilon(A, B)$ provides an approximate network distance that satisfies the condition

$$(1 - \epsilon) \cdot d_\epsilon(A, B) \leq d_G(s, t) \leq (1 + \epsilon) \cdot d_\epsilon(A, B) \tag{1.2}$$

Blocks A and B must be at the same level of the quadtree and represented by their Morton codes. To obtain the network distance between a pair (s, t) of vertices, the ϵ-DO finds a WSP $(A, B, d_\epsilon(A, B))$ where $s \in A$ and $t \in B$ and returns the approximate distance $d_\epsilon(A, B)$. An important property, called the *uniqueness property*,

is that every pair (s,t), $s,t \in V$ is contained in one *unique* WSP, which can be determined in $O(\log n)$ time.

PCPD produces a path oracle of WSPs (A, B, Ψ) such that A and B are blocks in a quadtree with the property that for any pair of vertices (s,t), $s \in A$ and $t \in B$, the shortest path between them passes through Ψ. In other words, Ψ is a common vertex or edge to all the shortest paths between any vertex in A to any vertex in B. In PCPD, the exact shortest path between any two vertices $s,t \in V$ can be computed as follows. First, retrieve the unique path oracle (A_1, B_1, Ψ_1) that covers s and t. Without loss of generality, Ψ_1 is a vertex in V, and by the above property of PCPD, Ψ_1 should lie on the shortest path from s to t. Therefore, Ψ_1 decomposes the shortest path between s and t into two components: the shortest path from s to Ψ_1 and the shortest path from Ψ_1 to t. This enables PCPD to further decompose each component into two smaller parts. Applying the above procedure recursively enables computing the shortest path from s to t with $O(k)$ lookups in PCPD, where k is number of vertices in the shortest path.

1.3 Contribution and Organization

However, there are two essential weaknesses with these two methods, ϵ-DO and PCPD. First, many subsequent papers (e.g., Wu et al. [74]) claim that ϵ-DO and PCPD are not scalable since computing the distance oracle representation is too slow, thereby making them feasible only for cities of moderate size (e.g., $80,000$ vertices). It means that ϵ-DO is not available even for a city road network such

as New York City with $264,346$ vertices. Second, both of these methods require modifications to the comparison function of the B-tree in a database, which significantly reduces the efficiency, and is not available for all types of databases. Besides, an additional shortcoming of PCPD is that it requires posing k SQL queries for retrieving a single exact shortest path, where k is the length of the shortest path. For the full USA dataset, k could be as high as $15,000$.

Thus, we first propose a new framework called ASDO that overcomes the two drawbacks of ϵ-DO and PCPD by making the following improvements.

1. We developed an infrastructure to resolve the scalability issue of ϵ-DO by making it possible to compute the ASDO representation for much larger road networks. Our experimental results show that the pre-processing needed to form the oracle for the entire USA road network can be achieved in 7.1 hours when using a modest size cluster of 20 Amazon EC2 machines incurring less than $50 in AWS charges.

2. We designed the ASDO representation to be *store-independent* which means that it does not require any modification to the database or any special indices. The ASDO representation is now available for popular RDBMS systems such as PostgreSQL, MySQL, Oracle, and SQL Server; For column-oriented DBMS like MonetDB [26]; For key-value stores such as Berkeley DB [48], HBase [29], and Redis [16]; and even for the in-memory distributed framework Apache Spark [75].

Secondly, we show how to use our ASDO representation in real applications on

popular RDBMS [52]. Two architectures, one is the *integrated architecture* (using our ASDO representation) within PostgreSQL and the other one is a widely used *hybrid architecture*, are evaluated for solving spatial analytic queries. In order to make our ASDO more powerful, in this part, we develop more efficient solutions. In particular, here we present SQL solutions for KNN and trajectory queries. These two types of queries serve as building blocks to enable people to easily write SQL solutions for more complex queries. In our examples, each spatial analytic query is expressed by just a few lines of SQL that utilize pre-defined functions. In contrast, the situation is far more complicated if for each of the queries users would have to devise efficient programs in Java (or other high level programming languages) to obtain the necessary query results. Experimental results indicate that our integrated architecture within PostgreSQL can compute more than $60K$ road distance operations per second on a large road network (e.g., USA), which achieves $20\times$ more throughput compared to the state-of-the-art shortest distance computation methods.

Thirdly, to achieve an extreme high-throughput performance, we develop a distributed framework called SPDO (pronounced *speedo* denoting *Spark and Distance Oracles*) using Apache Spark [75]. We extend our ASDO work [52] to map the distance oracle representation to a distributed key-value store (i.e., hash abstraction) which we choose to be Spark. Combining Spark and the ASDO representation is a good match. In essence, Spark provides a highly scalable fault-tolerant distributed framework with the ability to cache large datasets in memory using RDD [75], while the ASDO representation provides a compact representation of network distances

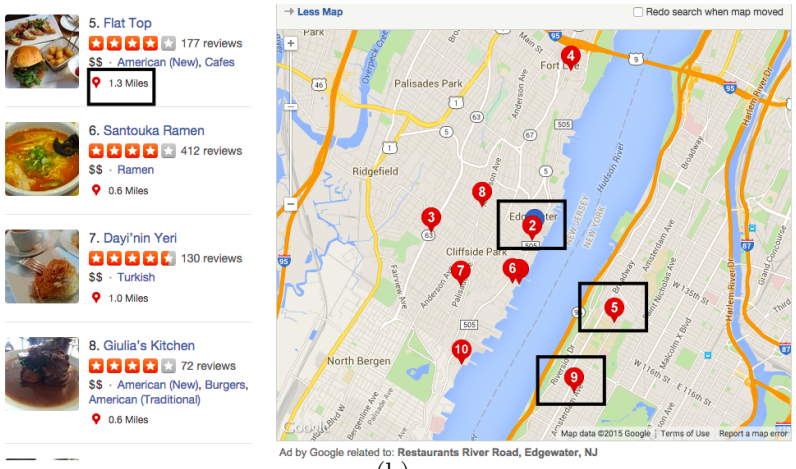
that requires very little computation at real-time. Furthermore, Spark is a popular open-source distributed framework for general purposes, which can be used as a key-value store. We can easily develop functions in Spark combining distance oracles and other techniques that are not efficient in a key-value store. In particular, we use the *IndexedRDD* library on Spark which is a memory resident, key-value store. The high-throughput of our proposed framework is achieved due to the ability to spread query processing across multiple machines in a Spark cluster as well as the in-memory representation of distance oracles.

Fourthly, after discussions with representatives of tens of related location-based service companies, we observed that some analytic queries on road networks usually focus in a local area spanning several cities. We first demonstrate a solution, termed City Distance Oracles (CDO) [53] to achieve as many as 7 million shortest distance computations per second per commodity machine on a city road network. Next, we extend the CDO solution to a new distance oracle system (DOS) for general road networks using some industrial techniques such as FlatBuffers [6]. It can solve most spatial analytic queries, and its throughput achieves $5M$ distance computations per second even on the whole USA road network. In addition, a $10K \times 10K$ origin-distance (OD) matrix can be computed in 20 seconds.

The rest of the thesis is organized as follows. Chapter 2 presents the ASDO framework to compute the ASDO representation. Chapter 3 describes efficient SQL solutions using our ASDO representation and also presents a detailed comparison between the performance of our ASDO method and the state-of-the-art methods for a variety of spatial analytic queries. Chapter 4 proposes the distributed framework

SPDO and the extreme high-throughput performance. Chapter 5 demonstrates the CDO solution with spatial queries from related companies. Chapter 6 introduces the implementation of DOS with a detailed evaluation. Chapter 7 concludes the thesis.

(a)



(b)

Figure 1.3: Using a geodesic distance ordering of results instead of network distance ordering: (a) Google Maps results for the query: find the Moroccan restaurants near to Broadway St & W Grant St, Bayonne, NJ. The ordering $A - H$ is the geodesic distance ordering provided by Google Maps and the ordering $1 - 8$ marked by green is the network distance (the values in blue) ordering we computed. (b) Yelp results for the query: find the restaurants around River Road, Edgewater, NJ that are within a 2 mile biking distance. The ordering $1 - 9$ is the biking distance ordering provided by Yelp. Obviously, restaurant #5 must be more than 1.3 miles by bike as it needs to cross the river.

Chapter 2: ASDO: An All-Store Oracle for Fast, Approximate Shortest Distances on Road Networks

2.1 Overview

During the analyst's exploration, each query can potentially involve thousands to millions of network distance computations being issued on a road network. Spatial analytic queries on road networks are typically performed by analysts who prefer to pose such queries using SQL. Since these queries combine existing database relations and perform network distance computations, one requirement is that these road network queries be executed entirely in a database system. This is attractive as it allows the analyst to leverage the power of a database language to create new types of online services resulting in easy programming, customization, and maintenance [52]. In this thesis, our goal is to develop a method with superior throughput while not sacrificing latency and to be able to work inside any database system with no additional software or hardware.

As an example of a use-case, consider an analyst working for a real-estate company (e.g., Zillow) who is looking for the top 100 houses in an area that are both affordable (i.e., list price of less than $\$500,000$) and that have several highly ranked

schools within a 1 mile network distance from them. To respond, the analyst would issue the following query that performs hundreds of thousands of network distance computations on a road network. The two tables of *House* and *School* are existing tables in a relational database system that are joined on the fly and filtered based on the network distance, house price, and school ranking.

```
SELECT House.id, COUNT(*) as count
  FROM House as H, School as S
  WHERE dist(H.lat, H.lon, S.lat, S.lon) ≤ 1 mile
        AND House.price ≤ $500,000
        AND School.ranking ≥ 9
  GROUP BY House.id
  ORDER BY count DESC
  LIMIT 100;
```

The clause $dist(H.lat,\ H.lon,\ S.lat,\ S.lon) \leq 1\ mile$ checks if the network distance between the house and school is within 1 mile, where *lon* and *lat* refer to the latitude and longitude, respectively, of the houses and schools. In our case, the *dist* function queries a lookup table to obtain the network distance between pairs of houses and schools rather than computing the network distances on the fly. Note that the database automatically optimizes the query by pushing the application of the predicate on price and ranking before the join operator in order to reduce the cardinality of the join.

There are only two methods that provide such kind of functionality, both

of which resort to precomputing and storing, as a database relation, the network distances between all pairs of vertices in the road network. Queries use this relation as a *lookup* table during query processing to estimate the network distance either accurately or to a great amount of accuracy during query processing. The first method is our prior work ϵ-DO [64], which computes ϵ-approximate network distance and compresses it to be stored as an *oracle* relation in the database requiring $O(\frac{n}{\epsilon^2})$ space for n vertices and an ϵ error bound. As noted in [74], this method is not scalable to road networks with more than $80,000$ vertices. It means that ϵ-DO is not available even for a city road network such as New York City with $264,346$ vertices. Furthermore, ϵ-DO requires modification to the database in terms of a Morton Index [64] on the *oracle* relation. The second method is HLDB [20] from Microsoft. It attempts to overcome both of these drawbacks of ϵ-DO, although HLDB requires an additional join operator, which can be expensive as shown by our experimental evaluation in Section 2.3.4.

In this chapter, we propose a new framework called ASDO whose main work flow is shown in Figure 2.1. It overcomes the main drawbacks of ϵ-DO. We also present a detailed evaluation of the execution time to compute the ASDO representation and of the accuracy of ASDO in our experiments. In addition, we provide the basic SQL solutions for some example queries. They show the ease with which analysts can devise SQL solutions for more complex queries using ASDO without having to worry about query performance.

In addition, we set up an ASDO demo [1] and provide some use cases in our

[1] `http://sametnginx.umiacs.umd.edu/`

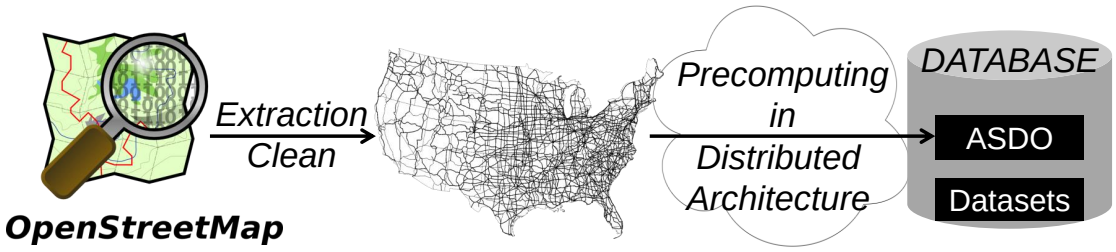


Figure 2.1: The work flow for obtaining the ASDO representation: First, extract any road network from OpenStreetMap [11]. Next, precompute the ASDO representation using our distributed framework illustrated in Figure 2.5. Finally, embed it into a database.

blog site [2]. They link to a free precomputed ASDO which we put in AWS S3. It needs just a few lines of code to be incorporated in an existing relational database.

When compared with HLDB [20] in a hard disk drive (HDD) storage, ASDO is about 100 times faster for network distance queries as we show later in the experimental results. Compared to methods that are optimized to compute a single network distance (e.g., the fastest one, DisLand [43], has an average latency of 0.28 ms on the road network of the USA), ASDO has a similar latency (about 0.25 ms on the average). However, when comparing throughput, we find that ASDO can answer $65,000$ queries per second, while the throughput of DisLand [43] is only $3,571$ queries per second. This enables us to achieve our goal of developing a method with superior throughput and capable of dealing with very large road networks. Of course, the price is that the results of ASDO are approximate. However, the errors

[2] `http://roadsindb.com/`

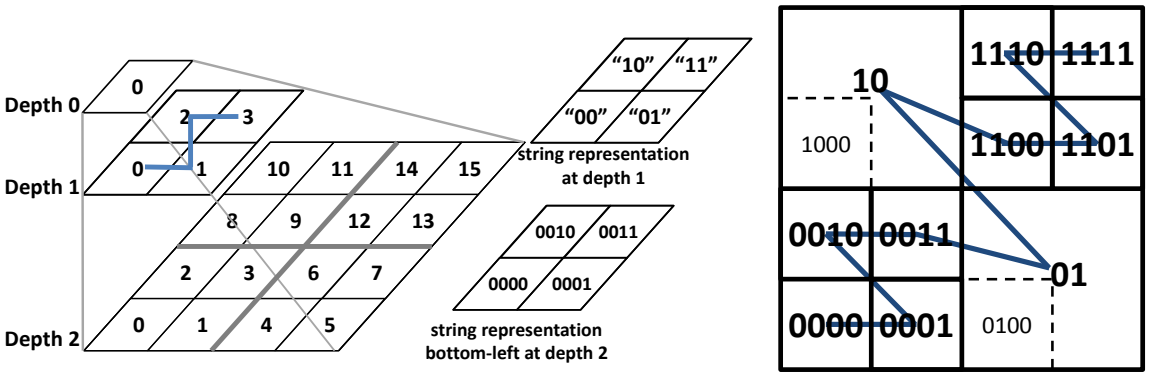
are bounded and significantly low.

The rest of this chapter is organized as follows. Section 2.2 presents our distributed framework and the ASDO representation. Section 2.3 contains a detailed experimental evaluation of our framework including disk usage, latency, throughput, and accuracy. Section 2.4 summarizes related work. Section 2.5 contains the basic SQL queries using ASDO in a database. Section 2.6 draws concluding remarks and describes directions for future work.

2.2 ASDO Framework

At the beginning, we introduce the Morton (Z) order space-filling curve [57] that provides a mapping, $\mathbb{Z}^2 \rightarrow \mathbb{Z}$, of a multidimensional object (e.g., a vertex or a quadtree block) in a 2-dimensional embedding space to a positive number. Given an object o, let $mc(o)$ be the mapping function that produces the Morton representation of o by interleaving the binary representations of its coordinate values.

Given a spatial domain S, the Morton order of blocks in S can be obtained by subdividing the space into $2^L \times 2^L$ equal sized blocks named *unit blocks*, where L is a positive integer named the maximal decomposition depth. Each unit block i is referenced by a unique Morton code $mc(i)$. Figure 2.2 shows how a Morton order of quadtree blocks in a two-dimensional space with $L = 2$. A spatial network $G(V, E, w, p)$ on the domain S can also be divided into $2^L \times 2^L$ unit blocks. Given vertex v in the unit block i, the Morton code $mc(v)$ is $mc(i)$. All vertices located in the same block have the same Morton code. Besides the unit blocks, every



(a) (b)

Figure 2.2: Examples of Morton codes in a 4×4 space: (a) Example of the number representation and the string representation of Morton codes in a domain space when $Depth = 0$, 1, and 2, respectively; (b) Example of the key representation of our ASDO.

larger block b has a unique Morton code, which is the longest common prefix of all unit blocks contained in b, e.g., the Morton code of the upper left quadrant $(1000, 1001, 1010, 1011)$ is 10. In this thesis, given blocks A and B, we define the relation $A \prec B$ if and only if block A is contained in block B, and thus $mc(B)$ is a prefix of $mc(A)$ denoted as PREFIX($mc(B)$, $mc(A)$). Once the data is sorted using this order, the resulting blocks can be stored using any one-dimensional data structure such as, but not limited to, a B-tree.

In the rest of this section, we describe the details of a new framework ASDO which overcomes the deficiencies of ϵ-DO. Instead of precomputing all the shortest distance pairs in ϵ-DO [64], ASDO computes only the distances between a few carefully selected representative vertices. We provide a deterministic way of choosing the representative vertices that will enable us to dramatically reduce the number

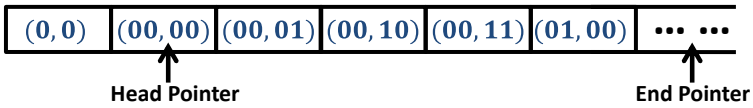


Figure 2.3: Example queue Q: it shows the processing order of the ASDO precomputation.

of shortest distance information that needs to be calculated. Furthermore, we can decompose the ASDO precomputation into a number of smaller parallel tasks, thus enabling the oracle to be computed using a cluster of machines. We explain the construction of our ASDO in Section 2.2.1, our distributed framework in 2.2.2, and our ASDO representation that obviates the need to modify the database in Section 2.2.3.

2.2.1 Encoding Network Distances

To compute ASDO, we first build a PR quadtree [57] on V based on the spatial position of the vertices. For quadtree block A, its Morton representation is given by $mc(A)$. The Morton code 0 represents the root block which spans the entire spatial domain S. The ASDO construction algorithm is a top-down approach that starts with the WSP decomposition [28] of the block pair (S, S), which is the largest *potential oracle*. A potential oracle is a pair of blocks that have not yet been examined, denoted as (A, B), where blocks A and B must be at the same depth of the PR quadtree and represented by their Morton codes. For example, a potential distance oracle denoted by $(01, 10)$ in Figure 2.2(b), where "01" denotes

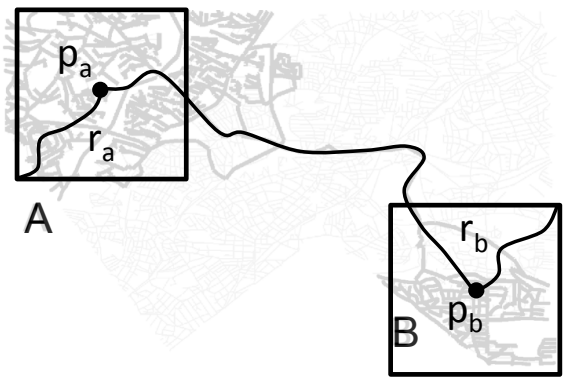


Figure 2.4: A potential oracle containing blocks A and B in Silver Spring, MD showing representative vertices p_a, p_b and the blocks' radius r_a and r_b.

the bottom-right 2×2 unit blocks, "10" denotes the upper-left 2×2 unit blocks.

Starting with (S, S) as the first entry, a queue Q holds all current potential oracles such as Figure 2.3. The algorithm pops a potential oracle (A, B) from the head of the queue. Figure 2.4 shows a potential oracle containing blocks A and B overlaid on the road network of Silver Spring, MD. The potential oracle (A, B) is first given to CheckOracle*(A,B)* described in Algorithm 1, which returns *true* if the network distances between all pairs of vertices in (A, B) can indeed be approximated by a single approximate value, in which case the potential oracle becomes an *accepted oracle* and we add it to the result set of ASDO. In addition, we also terminate the process if the quadtree blocks A and B are too small, i.e., their depth exceeds a given *threshold*. If CheckOracle() returns *false*, then we subdivide the potential oracle (A, B) into 4×4 new potential oracles by subdividing A and B once into their children quadtree blocks. The resulting potential oracles are inserted into Q and the algorithm continues.

We now describe the process by which we check to see if a potential oracle

Algorithm 1: CheckOracle(A, B)

Data: A, B: quadtree block; *Nodes*: an array of road vertices ordered by the Morton codes; h: a hash map storing the information of computed quadtree blocks; *Depth*: the depth of the quadtree blocks A and B

Result: *True* or *False*; d: network distance of (A, B)

if h *contains* $mc(A)$ **then**
    $p_A \leftarrow h[mc(A)].p$;
    $r_A \leftarrow h[mc(A)].r$;
else
    $p_A \leftarrow$ ChooseRep(A, *Nodes*);
    $r_A \leftarrow$ MaxDistance(A, *Nodes*, A_{start}, A_{end});
    $h[mc(A)].insert(\ \text{pair}(p_A, r_A)\ )$;
if h *contains* $mc(B)$ **then**
    $p_B \leftarrow h[mc(B)].p$;
    $r_B \leftarrow h[mc(B)].r$;
else
    $p_B \leftarrow$ ChooseRep(B, *Nodes*);
    $r_B \leftarrow$ MaxDistance(B, *Nodes*, B_{start}, B_{end});
    $h[mc(B)].insert(\ \text{pair}(p_B, r_B)\ )$;
$d \leftarrow$ GetDistance(p_A, p_B);
if *(Depth >threshold) or* ($d! = 0$ *and* $\frac{r_A + r_B}{d} \leq \epsilon$) **then**
    return True, d;
else
    return False, d;

can be an accepted oracle in Algorithm 1. In order to run CheckOracle(A, B), we first introduce some auxiliary information. Since every vertex v has a unique latitude and longitude, $mc(v)$ is also unique. We first sort all vertices by their Morton codes and place them in an array *Nodes*. For any block b in the PR quadtree, we calculate the minimum and maximum Morton code values b_{min} and b_{max} in O(1) time by computing the Morton codes for the bottom left and upper right corners. We then perform a binary search in *Nodes* to get the index range $[b_{start}, b_{end}]$ such that *Nodes*$[i]$, $i \in [b_{start}, b_{end}]$, corresponds to a sub-array that contains all vertices whose Morton codes contained in the range denoted by $[b_{min}, b_{max}]$. An equivalent way of describing *Nodes*$[i]$ is that it contains all vertices in *Nodes* whose prefix is $mc(b)$.

For any block b, $h[mc(b)]$ returns the pair (p_b, r_b), where p_b is the representative vertex of b and r_b is the network distance of the farthest vertex in block b from p_b (i.e., radius). The representative vertex p_b is chosen using a method discussed later. By reusing the representative vertex p_b for all blocks b, we significantly reduce the computational complexity of the computation of ASDO with respect to a method such as ϵ-DO [64] that chooses a representative point each time at random. If a block does not already have a representative vertex, then we choose a new representative vertex by invoking ChooseRep(), and then compute the radius of the block using MaxDistance().

A simple and efficient way to choose a representative vertex (i.e., ChooseRep() algorithm) is to pick the closest vertex to the geographic center of block b, termed p_b^{geo}. Since we know the Morton code of block b, we can compute p_b^{geo} by scanning

every vertex in b. This takes $O(|b|)$ time where $|b|$ is the number of vertices in b, i.e., $|b| = b_{end} - b_{start} + 1$.

MaxDistance() is implemented by using Dijkstra's algorithm starting at p_b and terminating once all vertices in b have been visited. Note that if we model the road network as a directed graph, for a potential oracle (A, B) under consideration, r_A is the maximum distance from all the other vertices in A to p_A, while r_B is the maximum distance from p_B to all the other vertices in B.

GetDistance() obtains the network distance d between p_A and p_B (i.e., $d = d_G(p_A, p_B)$). The CH method is a good choice to obtain these values and we use it here. Of course any other network distance method such as [21, 25, 43] could have also been used.

After we obtain the radius of the two blocks, r_A for A and r_B for B, and the distance d between the representative vertices, we test if $\frac{r_A+r_B}{d} \leq \epsilon$. If so, then we know that for any of vertices pair $(s, t), s \in A, t \in B$, recalling that $d = d_G(p_A, p_B)$, we have that

$$d_G(s,t) \leq d_G(p_A, p_B) + r_A + r_B \leq (1+\epsilon) \cdot d \tag{2.1}$$

$$d_G(s,t) \geq d_G(p_A, p_B) - r_A - r_B \geq (1-\epsilon) \cdot d \tag{2.2}$$

which it is consistent with Equation 1.2.

Reusing the representative vertices has a significant impact on the total cost of the precomputation. Theorem 2.2 below shows that the total time complexity for ChooseRep() and MaxDistance() as a result of reusing representative vertices is $O(n \log^2 n)$. In contrast, choosing the representative vertices at random results

in an algorithm with a time complexity of $O(n^2)$. This is one of the reasons for the scalability of the method of ASDO over ϵ-DO [64].

Lemma 2.1. *Assuming $\frac{\gamma_H}{\gamma_L}$ is a constant for the given spatial network, the number of accepted oracles and the potential oracles examined by Algorithm 1 are both $O(\frac{n}{\epsilon^2})$.*

Proof. The developers of ϵ-DO proved that given a spatial network with constant ratio $\frac{\gamma_H}{\gamma_L}$, they can construct an oracle of size $O(\frac{n}{\epsilon^2})$ [64]. Similar to their proof, the number of accepted oracles for ASDO is also $O(\frac{n}{\epsilon^2})$. In our construction process, one failed oracle produces 16 potential oracles. Since the number of accepted oracles is $N_{ac} = O(\frac{n}{\epsilon^2})$, the number of potential oracles examined by Algorithm 1 is: $N_{tot} = N_{ac} + \frac{1}{16}N_{ac} + \frac{1}{16^2}N_{ac} + ... = \frac{16}{15}N_{ac}$ □

Theorem 2.2. *Assuming that given a spatial network, the maximum depth of the PR quadtree is $O(\log n)$, and the time complexity of* ChooseRep*() is less than or equal to $O(|A| \log |A|)$ for every quadtree block A, then the time complexity of precomputing ASDO is bounded by $O(n \log^2 n + \frac{n}{\epsilon^2} \cdot Time(CH))$, where $Time(CH)$ is the average response time for a single exact network distance query.*

Proof. As we see, Algorithm 1 calls three external functions: ChooseRep(), MaxDistance(), and GetDistance(). For each quadtree block A, ChooseRep() and MaxDistance() are only calculated once. So, we analyze these two functions first. Since MaxDistance() is Dijkstra's algorithm, its time complexity is $O(|A| \log |A|)$. So, if the time complexity of ChooseRep() is less than or equal to the time complexity of MaxDistance(), then MaxDistance() would dominate the time consumption among these two functions.

Without loss of generality, here we assume that the maximum depth of the PR quadtree is bounded by $O(\log n)$, since if in a road network input there are two vertices that are infinitesimally close to each other thus causing the depth of the PR quadtree to be arbitrarily large, then they are typically due to errors in the dataset, which can be cleaned up by a preprocessing step that merges such vertices. We first obtain the time complexity of MaxDistance() for each depth. Note that we only invoke MaxDistance() for a quadtree block A if the number of network vertices in A is larger than 0, i.e., $|A| > 0$. At depth l, suppose that there are K_l quadtree blocks that $|A_i| > 0, i \in [1, K_l]$. Since each vertex of the spatial network must be located in one of the K_l quadtree blocks, we have $\sum_{i=1}^{K_l} |A_i| = n$. Then, the upper bound of the total time complexity of MaxDistance() at depth l is

$$T_l(n) = \sum_{i=1}^{K_l} |A_i| \log |A_i| \leq \sum_{i=1}^{K_l} |A_i| \log n = n \log n \tag{2.3}$$

Thus, the upper bound of the time complexity of MaxDistance() in the whole process is

$$T(n) = \sum_{l=1}^{O(\log n)} T_l \leq O(n \log^2 n) \tag{2.4}$$

On the other hand, the lower bound can be obtained by considering the total time complexity of MaxDistance() in the best case scenario that the number of vertices in one quadtree block is always $\frac{1}{4}$ that of its parent block. The total time complexity $T^*(n)$ of MaxDistance() in this case is given by the Master theorem:

$$T^*(n) = 4 \cdot T^*(\frac{n}{4}) + O(n \log n) = O(n \log^2 n) \tag{2.5}$$

Thus, the lower bound of MaxDistance() in this process is also

$$T(n) \geq T^*(n) = O(n \log^2 n) \tag{2.6}$$

From the inequalities (2.4) and (2.6), we know that $T(n) = O(n \log^2 n)$ when the maximum depth of the PR quadtree is $O(\log n)$.

In the second part of this process, GetDistance() is invoked for each potential oracle. Based on Lemma 2.1, it would result in N_{tot} invocations of a network distance algorithm (e.g., the CH algorithm), where N_{tot} is $O(\frac{n}{\epsilon^2})$ as well. Since this invocation is to a module that is essentially a black box to our algorithm and furthermore we have multiple instances of the algorithm running on different machines to hide latency, the time complexity of GetDistance() is essentially $O(\frac{n}{\epsilon^2}) \cdot Time(CH)$. Note that since most latency methods such as CH have a pre-processing stage, their latency time, e.g., $Time(CH)$, is far less than $O(n)$.

In summary, the total time complexity of computing ASDO is $O(n \log^2 n)$ for ChooseRep() and MaxDistance(), plus $O(\frac{n}{\epsilon^2} \cdot Time(CH))$ for GetDistance().

□

Next, considering function ChooseRep(), there are many ways of implementing the ChooseRep() function to compute the representative vertex of each block. An efficient way is to choose the p_A^{geo} for a block A. It can be calculated in a single scan, which takes $O(|A|)$ time. In fact, if we build a spatial index for the vertices, then we can compute p_A^{geo} in $O(\log n)$ time. Later, refer to this method as the *Geo* method.

From our experiments, we found that p^{geo} is a time-efficient choice but not a space-efficient choice in the sense that it is possible to choose the representative vertex in a different way to further reduce the size of the resulting oracle.

The rationale for choosing the geographic center is because p_A^{geo} tends to make r_A smaller, hence rendering $\frac{r_A+r_B}{d} \leq \epsilon$ easier to achieve. However, the geographic method is just a heuristic approximation of a method that identifies a vertex p_A^c that minimizes the distance to the farthest vertex from it, i.e., $r_A(p_A^c)$. This vertex is known as the *graph center* (or Jordan center) [73]. Using p_A^c and p_B^c means that we minimize r_A and r_B as well as their sum, making $\frac{r_A+r_B}{d} \leq \epsilon$ more likely. We found that using the graph center reduces the number of oracles by 35–55% compared to the geographic method. Unfortunately, there is no efficient way to compute the graph center other than the $O(|A|^2 \log |A|)$ method, which essentially results in running Dijkstra's algorithm $|A|$ times. As a way of keeping this cost under check, we use the graph center algorithm sparingly. In particular, only use it if $|A|$ is small (i.e., less than $2,000$ vertices in our case); otherwise we use the geographic method for lager blocks. We found this approach to be also much more space-effective than the geographic method in our experiments since most accepted oracles consist of medium or small size blocks. We refer to this method as the *Hybrid* method in the experiments section.

2.2.2 Task Partition and Parallelism

As the size of the computation becomes larger, we need a distributed architecture to compute ASDO since it takes a long time with a single machine. The quadtree structure that we use to represent the oracles lends itself to partition the workload. From Algorithm 1 and Figure 2.3, we observe that the task of examining

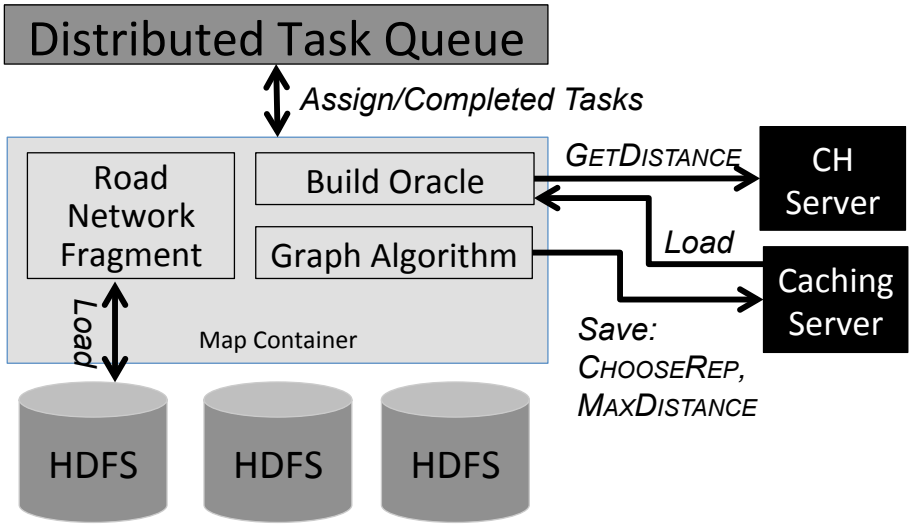


Figure 2.5: Distributed architecture for computing the ASDO representation

each potential oracle is essentially a data-independent task. Based on this observation, we design and implement a distributed architecture for ASDO precomputation showed in Figure 2.5.

Since precomputation takes a bit of time, we employ the Hadoop framework to benefit from its in-built fault recovery feature. There is a bank of machines that handle network distance queries needed during precomputation. As mentioned before, we run the CH Algorithm [39] (referred to as "CH servers") in these machines that are accessed through a load balancer. We also run a caching service on the CH servers for saving and retrieval of information about (p_A, r_A). These are essentially key-value stores where the Morton codes of the blocks form the keys. Finally, we use a distributed queue (e.g., ActiveMQ) for task assignment.

We decompose the precomputation step into two steps. In the first step, the CH servers load the graph in their main memory and perform extensive graph operations. The goal here is to load the graph once, use it many times and store

auxiliary information for later use. In the second step, the map tasks simply query the CH servers without requiring any graph information. Since in this framework, the state information is stored in the queue, unless the queue fails, the map process can terminate and restart.

In the first stage of computing, we compute ChooseRep() and MaxDistance() for each quadtree block A and save the result to the caching server. As shown in Figure 2.6, we decompose the road network into 16 quadtree blocks in the Morton order such that each quadtree block fits in the main memory of the machine. This is loaded from the HDFS into the main memory where it resides until the first stage is complete. We then select the representative vertex either by choosing a vertex near the geographic center of a quadtree block for the *Geo* method or apply the graph center algorithm to compute the center for the *Hybrid* method. Once we have obtained the representative, we compute the radius of the quadtree block by applying Dijkstra's algorithm. We save the representative vertex for each quadtree block and its radius in the caching server. We then subdivide the quadtree block and continue processing until we reach the leaf blocks. These algorithms are implemented in the graph algorithm module given in Figure 2.5. At this point, for every block in the quadtree, we have stored a representative vertex and its radius.

In the second stage of processing, we start by populating the distributed queue with the potential oracles corresponding to an initially chosen depth. For the US, we choose a depth of 4 as shown in Figure 2.6, so we initialize the quadtree with 16 blocks and the queue with 16^2 potential oracles. We do this since starting with the root potential oracle (S, S) results in a "slow" start for our algorithm as the quadtree

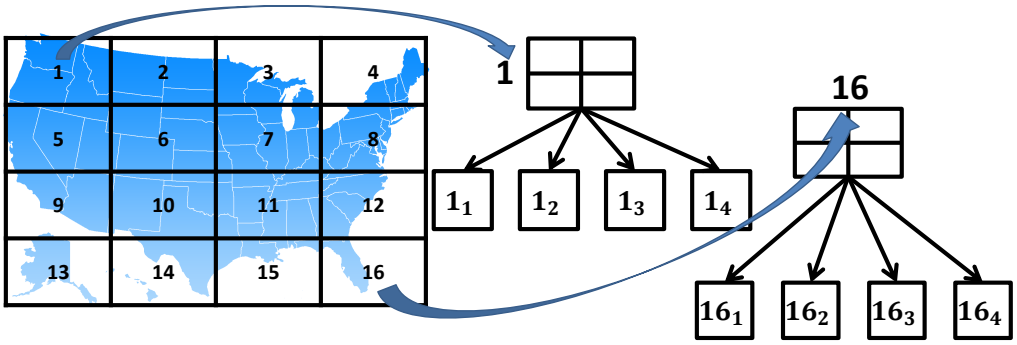


Figure 2.6: Initial decomposition of the US dataset where each quadtree block fits in main memory

blocks are really large and these may eventually not participate in any accepted oracles. Moreover, the initial depth is chosen as 4 since the graph representation corresponding to the quadtree block fits in the main memory of the machine. For larger quadtree blocks than these, we don't need to compute the representative vertices and the radius. Computing ASDO starts by requesting a potential oracle from a queue. Then the check process would invoke ChooseRep(), MaxDistance(), and GetDistance() by making requests to the CH servers. Finally, check if the potential oracle satisfies the WSP property. If it does not, then decompose the potential oracle into its 4×4 children potential oracles and insert them into the queue; otherwise, the potential oracle is saved to the HDFS as an accepted oracle. When the process finishes, ASDO has been computed and can be loaded into a database.

2.2.3 ASDO Representation

Given a source location $p_1 = (lat_1, lng_1)$ and a destination location $p_2 = (lat_2, lng_2)$, traditionally computing the shortest distance between them requires

two steps: (1) Find the *nearest road vertices* s, t to p_1 and p_2, respectively; (2) Calculate the network distance between s and t. The first step requires a query to a spatial index (e.g., k-d tree, quadtree, R-tree, etc.) to obtain the nearest vertex, after which the network distance can be obtained by traversing the graph information. Our ASDO method can directly take the source and destination locations to obtain the network distance, which means that we get step (1) for free.

Once ASDO has been computed, we load it into a table in a relational database system. The schema of ASDO is given by (code, d), where code is a concise representation of the accepted oracle and d is the approximate network distance. Although such a schema is similar to the one proposed in ϵ-DO [64], our method just uses the default integer comparator operator instead of redefining the string comparator operators (i.e., $<$ and $=$) while searching for a code using the B-tree. This is important because the default integer comparator saves much time in contrast to the redefined string comparator.

To illustrate our newer method for packing the code, we first start with a simpler two-dimensional example (i.e., Z_2) and we then describe how to encode an accepted oracle as a four-dimensional Morton block. Suppose that we have a number of varying length Morton codes in two-dimensions, which means that the corresponding quadtree blocks are at different depths. The simpler problem we want to solve is that we are given a point p, and we need to efficiently find a unique quadtree block A containing p. Here we assume that the uniqueness property from the property of WSP [28] is also true in this simpler example. The uniqueness property here means that there is exactly one quadtree block containing p such as

in Figure 2.2(b). This search problem is equivalent to finding the unique $mc(A)$ such that PREFIX($mc(A)$, $mc(p)$). This is done in ϵ-DO [64] by truncating one of the two comparand Morton codes to be the same length as the other Morton code and then checking if they are the same value. Truncating the Morton codes to make them the same length is the reason why ϵ-DO [64] requires overloading the comparison operator.

Instead of truncating one of the Morton codes, our approach is to make all the Morton codes have the same length by padding them with enough zeros, so that all Morton codes are always the same length, i.e., $2 \cdot L$ bits long in two-dimensions. For any Morton code $mc(A)$, padding with enough zeros is equivalent to choosing a unit-sized block that is a descendant of A in the quadtree that has the smallest Morton code. This needs to be done carefully as we illustrate with the following example. Suppose that our two-dimensional oracles has ten quadtree blocks as in Figure 2.2(b) whose Morton codes are 0000, 0001, 0010, 0011, 01, 10, 1100, 1101, 1110, and 1111. Only two Morton codes 01 and 10 are not 4 bits long. Thus, consider the quadtree blocks 01 and 10 in Figure 2.2(b), which we convert to 0100 and 1000 respectively by padding zeros to the right hand side. The codes of our oracle become: 0000, 0001, 0010, 0011, 0100, 1000, 1100, 1101, 1110, and 1111 in order. Given a query point $p = 0111$ that is contained by a unique quadtree block A. To find A, we need to find a quadtree block in the B-tree such that it is the largest value that is less than or equal to p, which in this case is 0100 (i.e., quadtree block 01, which is the correct answer).

Now going back to ASDO, we obtain a four-dimensional Morton code by inter-

leaving $mc(A)$ and $mc(B)$ two bits at a time. This packing is given by the function $Z_4(A, B)$. Next, we define function $Z_4^0(A, B)$ by padding $Z_4(A, B)$ with zeros to the right side. For example for the blocks in Figure 2.2(b), Z_4 and Z_4^0 should be

$$Z_4(01, 10) = 0110 \qquad Z_4^0(01, 10) = 01100000 \tag{2.7}$$

$$Z_4(0000, 1111) = Z_4^0(0000, 1111) = 00110011 \tag{2.8}$$

This packing Z_4^0 produces a Morton code of $4 \cdot L$ bits length. This forms the *code* attribute of the relation table which is indexed by a B-tree. At this point, given a source location p_1 and a destination location p_2, the approximate network distance query first calculates $key = Z_4^0(mc(p_1), mc(p_2))$ in $O(1)$ time which will be introduced in Section 2.5.1 and then issues the following query that is answered extremely efficiently by the B-tree index on code.

```
SELECT code, d
  FROM ASDO
  WHERE code = (
    SELECT max(code)
      FROM ASDO
      WHERE code<=key)
```

```
OR
```

```
SELECT code, d
  FROM ASDO
  WHERE code <=key
  ORDER BY code DESC
  LIMIT 1
```

This scheme works because of the uniqueness property of WSP. For any two points in the domain S, there is exactly one WSP containing them.

To choose a suitable value of depth L, one consideration is that ideally L should be less than or equal to 16, so that the Morton code Z_4 can fit in a 64 bits integer. Thus, having a longer Morton code does not affect correctness but increases

the size of the oracle. For all practical purposes, we can safely truncate the quadtree at a depth of 16 or less for most road networks. For instance, for the US dataset, we use a depth of 15 which provides a resolution of 100 meters.

2.3 Experiments

In this section, we first describe our experimental environment in Section 2.3.1. We give an overview for the previous latency and throughput methods versus ASDO in Section 2.3.2. Next, we present a detailed evaluation of our approach, which includes the following two components evaluated in Section 2.3.3 and 2.3.4, respectively, 1) The cost of the pre-computation stage which computes all oracles and inserts them into a database, 2) The latency time, the throughput performance, and the accuracy of network distance queries.

2.3.1 Computing Environment

We evaluated two clusters for precomputing ASDO, one in-house and one on AWS. Our in-house cluster consists of Intel Xeon(R) E3-1225 v3 CPUs @ 3.2GHz (4 cores) with 16 GB RAM. A single machine precomputes ASDO except for the USA dataset, for which we employed a cluster containing 4 machines. To show scalability, we use a modest cluster size of 20 Amazon EC2 *m3.xlarge* instances. Once ASDO has been precomputed, it is loaded into a PostgreSQL 8.4.18 database which is used to answer queries. PostgreSQL was installed on a larger server machine, which has Xeon(R) L5520 CPUs (2.26 GHz and 8 cores), 24GB RAM with a commodity 2TB

Table 2.1: Dataset Characteristics in ASDO

Name	NY	FL	US	*EU
Region	NY City	Florida	USA	Western Europe
# of Nodes	264,346	1,070,376	23,947,347	18,029,721
# of Arcs	733,846	2,712,798	58,333,344	42,199,587
Max depth ($0.1m$)	20	22	25	-
Practical depth ($100m$)	10	12	15	-

hard disk and a 7200 RPM RPM disk with 64MB Cache.

Table 2.1 provides the characteristics of the road network datasets used in our evaluation. The first three datasets are from the 9^{th} DIMACS Implementation Challenge [3]. The last dataset is the Westen Europe (EU) road network which is used in [20]. We do not use it in our evaluation since, instead, we use the larger US road network dataset. The latitude and longitude values in the road network have a resolution of at most 6 decimal place values. The maximum resolution in terms of the number of bits needed to represent the coordinate values is 20, 22, and 25 for the NY, FL, and US datasets, respectively. Since we employ a quadtree to discretize the positions of the vertices on the road network, we need to choose the maximum depth of the quadtree at which to truncate it. For the experiments in this section, we terminate the decomposition process in the ASDO computation step at the 100 meter depth. The maximum depth L of the quadtree headed to provide a 100 meter resolution for the NY, FL, and US datasets is 10, 12, and 15, respectively. This

means that if the source and the destination are closer than 100 meters, then we simply return the Euclidean distance between them. If not, ASDO is guaranteed to provide the ϵ-approximate network distance. In addition, the length of Z_2 for a unit-size block is $(2 \cdot L)$. Limiting the oracles to this resolution also enabled us to pack the Morton block of each accepted oracle inside a 64 bits integer for the US dataset. We need just 40, 48, and 60 bits to represent an accepted oracle for the NY, FL, and US datasets, respectively.

2.3.2 Network Distance Retrieval

We first compare ASDO and the related methods based on the latency needed to compute network distances on a road network. Table 2.2 summarizes the results reported in previous work for the datasets that we use in this chapter. Note that since ϵ-DO is not scalable even for the NY dataset, we cannot show the latency and throughput results of ϵ-DO here. The first three methods CH [39], TNR [25], and DisLand [43] are memory-based latency methods. Wu et al. [74] is an experimental survey that separated queries into various groups and tested the average latency time for each query group. From the experimental survey [74], the response times of both CH and TNR are different from their response time in [39] and [25]. HLDB [20] is the first practical method inside a database. Comparing these methods is not straightforward in the sense that the methods differ in terms of their storage and precomputation requirements. The in-memory latency methods compute network distance values at run-time, while the throughput methods simply retrieve

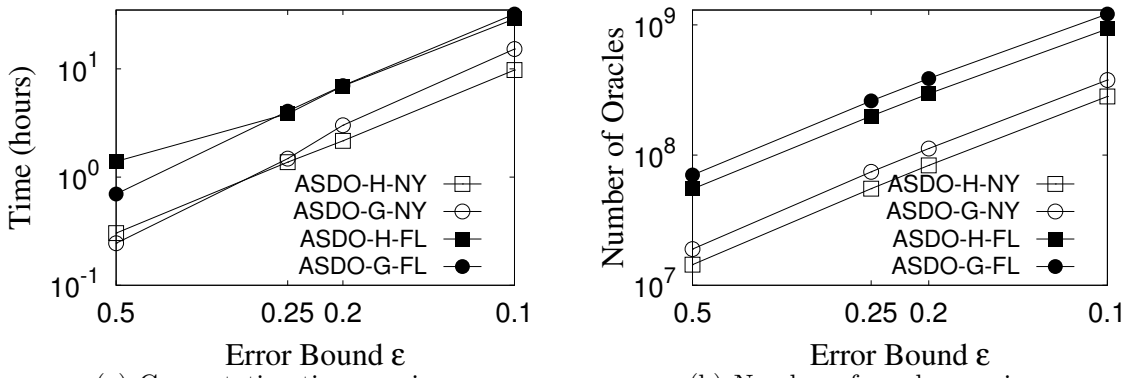
the network distance from a precomputed representation. Even among the latency methods, they differ in the amount of preprocessing requirements before they can be used. Since we use standard road network datasets, we first take results from related work to draw broad conclusions from the comparison results in this section. Then, we only implement CH and HLDB in our later experiments.

Table 2.2: Latency and Throughput Results of ASDO

Name	ref.	Avg latency time (ms)	Avg throughput/sec
CH	[74]	5–20, FL 5–50, US	350, US
	[39]	0.2–1.2, EU	
TNR	[74]	1–10, FL 1–20, US	2000, US
	[25]	0.01–0.3, US	
DisLand	[43]	0.01–0.1, FL 0.1–0.3, US	3571, US
HLDB	[20]	10–30, FL 30–70, EU	-
ASDO	-	0.20, NY (HDD) 0.22, FL (HDD) 0.24, US (HDD)	65000, US

Table 2.2 gives the average latency for a distance query in the various datasets, and its average throughput per second for the largest dataset, US. Among the latency methods, the DisLand method is significantly faster than the CH and TNR methods. As expected, HLDB under a hard disk drive (HDD) environment is slower than memory-based algorithms. For all methods except ASDO, the latency for a distance query is given as a time range since the time to compute the network distance depends on the distance between the source and the destination. For instance, the latency is higher if the distance between the sources and destinations is large. In contrast, the latency time of ASDO for a distance query is relatively constant (i.e., it does not vary for different shortest distances). ASDO is also significantly faster than all other latency methods, with the exception of the DisLand method. ASDO is almost 100× faster than HLDB for the same dataset. The reason for this is that HLDB performs an expensive join operator in order to retrieve a network distance, while ASDO performs a B-tree lookup which is the reason that its latency is around 0.20 ms for all three datasets.

The real benefit of ASDO is that it can provide a high throughput rate compared to the latency methods. For instance, by just using a single machine, ASDO can answer about $65,000$ network distance queries within one second, which is an order of magnitude higher than any of the latency methods. We discuss the throughput results in more details in Section 2.3.4. To summarize, ASDO has comparable latency, while providing significantly higher throughput compared to the state-of-the-art latency approaches.



(a) Computation time varying ϵ

(b) Number of oracles varying ϵ

Figure 2.7: Pre-computation performance varying ϵ

2.3.3 Precomputing ASDO

In this section, we first show how the different strategies for choosing the representative vertices significantly affect the number of accepted oracles. ASDO-G-ϵ denotes the *Geo* method, where the representative vertex is the one that is closest to the geographic center, and ASDO-H-ϵ indicates that the representative vertex is chosen using the *Hybrid* method that we discussed earlier.

Figure 2.7(a) shows the time to compute the oracles using a single machine. ASDO for the NY and FL datasets can be done in less than an hour for a fairly large ϵ value of 0.5 and in a little over 10 hours for $\epsilon = 0.1$. Note that these are large datasets comprising road networks of states in the US and being able to compute them within a few hours on a single machine means that computing the oracles is a practical proposition. Furthermore, we later show for the US dataset that by adding more machines to the computing infrastructure, we can significantly speed up this process. Next, for each method, we provide the number of accepted oracles for the

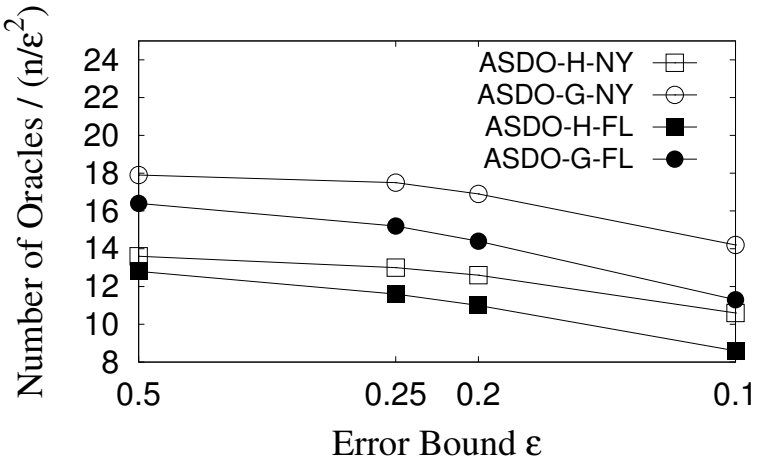


Figure 2.8: Number of oracles varying ϵ, C =normalized by n/ϵ^2

NY and FL datasets in Figure 2.7(b). Here we let ϵ vary taking on the values 0.5, 0.25, 0.2, and 0.1. We observe that as ϵ decreases, the number of accepted oracles increases. Figure 2.8 plots the ratio of the number of oracles and $\frac{n}{\epsilon^2}$ versus ϵ. The relatively horizontal lines in Figure 2.8 confirm that the number of oracles conforms to $C \cdot (\frac{n}{\epsilon^2})$ for the NY and FL datasets, where the value of C ranges between 8 and 14 for the Hybrid method, and between 11 and 18 for the Geo method.

Figures 2.7(a) and 2.7(b) show that the computation time is linear with the number of accepted oracles since the slope rates of the lines in the two figures are similar. From Figure 2.7(a), we can see that it is faster to compute ASDO-G than ASDO-H when ϵ is large, but is slower as ϵ decreases. This is because computing the center of the subgraph dominates the execution time for larger values of ϵ (e.g., 0.5), while the time to compute the network distance between representative points starts dominating as the value of ϵ decreases.

Next we show scalability results in Table 2.3 for ASDO using the US dataset

Table 2.3: Precomputation for ASDO on US Dataset

Performance	ASDO-H-0.25	ASDO-G-0.25
Cluster Size	Time	Time
4	1.2 days	1.4 days
20	7.1 hours	7.9 hours
Speed-up	4.05	4.25
Number of oracles	4.6 billion	10.4 billion
C = Number of oracles / (n/ϵ^2)	11.9	27.0

with ϵ of 0.25. For this experiment, we used two clusters: an in-house cluster of 4 machines and an Amazon EC2 cluster with 20 machines. We show the results for both the Geo and the Hybrid methods. From the table, it can be seen that precomputing ASDO can reduce the time needed from 1.4 days when using 4 machines to a little over 7 hours when using 20 machines. This constitutes a speed of 4.05 and 4.25 for the Hybrid and Geo methods respectively. Note that by going from 4 machines to 20 machines, we have roughly scaled the computing cluster by a factor of 5. The speedups we obtained for both methods are very close to 5 which indicates that our algorithm provides a linear speedup in the number of machines, which is a desirable property that one expects from parallel algorithms. Moreover, the cluster compute and storage cost for precomputing the oracle is less than $30 based on AWS December 2015 prices. These results provide powerful support for our claim vis-a-vis the feasibility of our method since it is cheap to precompute and

can be further sped up by simply adding more machines. In fact, if need be, the oracle can simply be recomputed rather quickly if there are large scale changes in the road network such as major road closures etc.

Table 2.3 shows that the Hybrid method significantly reduces the number of oracles compared with the Geo method. We see that the value of C is around 12 for the Hybrid method but much higher at 27 for the Geo method. Note that the value of the constant C depends on the nature of the dataset, but the range is still narrow thereby enabling us to estimate the expected disk usage. Finally, Figure 2.8 shows that C decreases as ϵ decreases, which is a good news for applications that require higher accuracy.

Database Choices and Disk Usage. Once ASDO has been computed, we next load it into a database system. The structure of ASDO is simple enough so that it can be loaded into a number of database families – row stores (e.g., PostgreSQL), column stores (e.g., MonetDB [26] and C-Store [70]), and key-value stores (e.g., Berkeley DB [48], HBase [29], and Redis [16]). PostgreSQL takes 285 GB to store ASDO for the US dataset containing 4.6 billion accepted oracles. This is much more than what we had expected since we are essentially just storing a bigint for the Z_4^0 value and a float for the network distance value, which should amount to 12 bytes for each tuple, with a total expected storage of 55.2 GB for the US dataset. The overhead in storage is due to the extra 27 bytes of the fixed-size header for each tuple as well as the additional space for the B-tree index. Column-oriented databases are much more space efficient than either row or key-value stores. For example, MonetDB [26] does not require extra disk storage since it stores by at-

tributes without requiring additional header information per tuple. In addition, some column-oriented databases also apply a compression on each column to further reduce the disk usage. Berkeley DB [48] with a B-tree index also requires extra disk usage since it needs header information to record the type of the data and its value. To quantify the additional space overhead, if the theoretical size of a tuple is 12 bytes, then the actual disk consumption for including an index is roughly 5.1× on PostgreSQL, 1.0× on MonetDB, and 3.5× on Berkeley DB.

2.3.4 ASDO Query Time and Accuracy

In this section, we evaluate the performance of the query response component from the perspectives of latency, throughput and accuracy. We have two query types: *basic query* and *analytic query*. *Basic query* performs a single network distance query and its corresponding SQL query is given in Section 2.5.2. *Analytic query* is expensive including thousands to millions of network distance computations. For example, suppose a database has a relation of universities and restaurants, and we want to find the average network distance from each university to its 10 closest restaurants (see Section 2.5.3 for the SQL query).

Our experiments found very little difference between the the Hybrid and Geo methods when it comes to query processing. Hence, here we only report the results of the Hybrid method. Our SQL queries are submitted via JDBC using a Java program to the PostgreSQL server. To avoid the influence of the network delay on time-consuming testing, we used the "UNION ALL" operator in PostgreSQL to

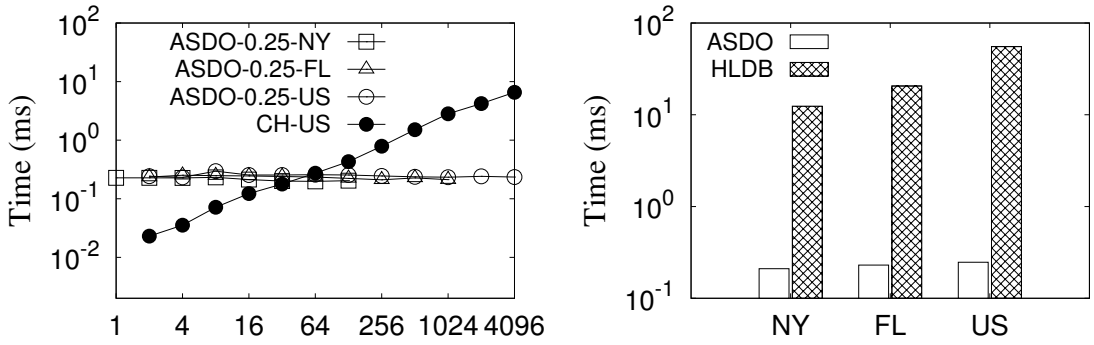
combine multiple SQL queries into one server roundtrip.

Table 2.4: Average latency time and error for the basic query in ASDO

Datasets	ϵ	Avg Time (ms)	Avg Error
NY	0.1	0.23	1.50%
	0.25	0.20	3.55%
	0.5	0.19	5.35%
FL	0.1	0.24	1.30%
	0.25	0.22	2.89%
	0.5	0.21	4.77%
US	0.25	0.24	2.74%

For every dataset used in the basic query experiments, we randomly generated 100,000 source target pairs to test the performance. Table 2.4 provides the average latency and the error percentage. The average latency time is stable ranging between 0.2 to 0.3 ms for a single distance query, which is similar to the amount of time needed by the fastest memory-based methods. Moreover, since our method is incorporated into a database, each SQL query just accesses the B-tree index. The database can batch multiples of these lookups in the B-tree at the same time in order to obtain a high throughput, e.g., $65,000$ shortest distance pairs within one second for multiple users or an analytic query.

On the other hand, the average error of ASDO is much smaller than the



(a) Comparison to CH

(b) Comparison to HLDB

Figure 2.9: Latency time comparisons between ASDO, CH, and HLDB: (a) We compare ASDO to CH by grouping results based on the network distance between the sources and destinations. (b) We compare ASDO to HLDB on three road networks under the same database and hardware environment.

corresponding error bound ϵ, which is roughly $(\frac{1}{10} \cdot \epsilon)$. We also observe that ASDO is much faster than the memory-based methods and HLDB, with a small sacrifice in accuracy. Below we provide more details on the effect of varying ϵ and the distance on the response time and accuracy.

2.3.4.1 Latency Time and Throughput for Basic Query

Figure 2.9 compares ASDO, CH, and HLDB. The HLDB implementation in our experiments is based on [21] and [20]. We computes the hub labels for each vertex using the forward CH search and backward CH search with the *stall-on-demand* heuristic modification [21]. Figure 2.9(a) tabulates the latency time for a single distance query using ASDO on a number of different datasets and different distance values. We use the CH method as the ground truth to compute the resulting

error in the answer provided by the distance oracle. We report by grouping results based on the network distance between the sources and destinations. In particular, we bucket results into ranges given by $[2^x, 2^{x+1})$, where x is an integer and 2^x is in kilometers. Unlike other previous approaches given in Table 2.2, the latency time for ASDO is relatively constant, taking about 0.2 ms. Furthermore, the retrieval time is the same across all the three datasets, even for the US dataset which is much larger than the rest. This is on account of using a B-tree which provides a scalable behaviour across oracles of different sizes. This is in contrast with the CH method where the latency increases with the increase of the network distance between the sources and the destination as well as increases as the road network dataset becomes larger. Figure 2.9(b) illustrates the performances of ASDO and HLDB on three road networks in the same PostgreSQL database and HDD environment. As the retrieval time of HLDB is also independent of the network distance [20], we immediately compare the average latency time between ASDO and HLDB. ASDO is almost two orders of magnitude faster than HLDB since HLDB needs a join operator between forward hubs and backward hubs for each *basic* query.

As seen in Figure 2.9(a), ASDO is slower than CH for short distance queries. However, this denotes a case where a single user is posing queries on both CH and ASDO. The situation becomes quite different when multiple users are posing queries at the same time. In this case, a method that can deliver a high throughput without significant degradation of latency is desirable. The throughput of ASDO and CH and the latency degradation of ASDO are shown in Figure 2.10, while varying the number of concurrent users. In Figure 2.10, ASDO-US-throughput and CH-US-throughput

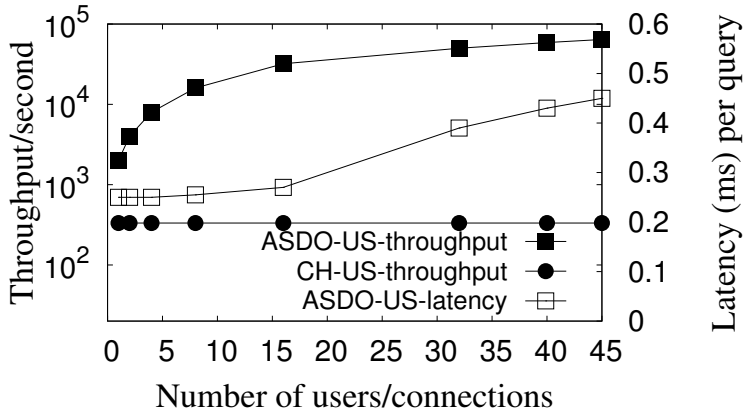


Figure 2.10: Throughput (queries/second) and latency for varying numbers of users

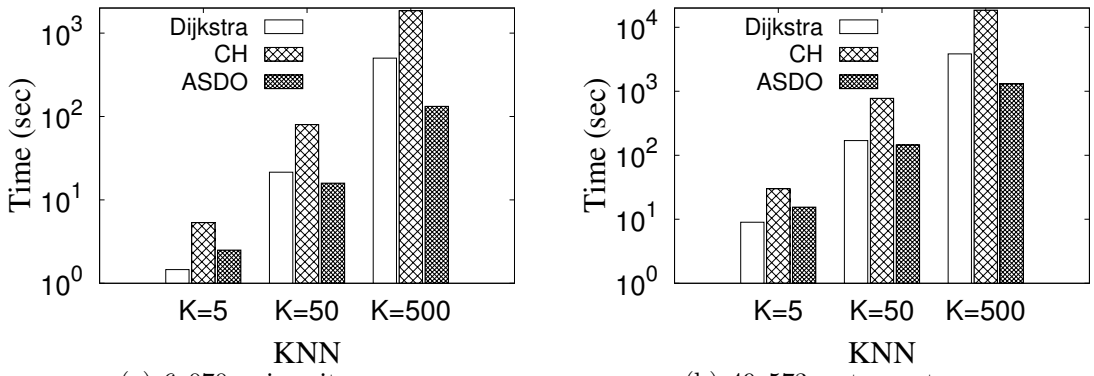
use the left y-axis for the average throughput, while ASDO-US-latency uses the right y-axis for the average latency. In order to model a workload of concurrent users submitting concurrent queries, we implemented a JAVA program that repeatedly generates pairs of vertices at random and submits a query to compute the network distance between them to a database. We keep varying the number of users between 1 and 45 and measure the latency as well as the number of total queries that the system can answer in one second (i.e., throughput). The maximum throughput we obtained was 64,285 queries per second with 45 concurrent users. Increasing the number of users beyond 45 users results in a small increase to the throughput, indicating that the system has reached saturation. While the throughput increases because of more users, the latency time also increases. When the number of users is less than 16, the increase of the latency is negligible. After 16 users, the latency increases in a linear rate with the number of users. For reference, we also plot throughput for CH on the US dataset. As this is a memory-based algorithm that stores the graph and auxiliary information in memory, we couldn't run more than

one instance of CH on a machine, and each instance could only process one query at a time.

2.3.4.2 Analytic Query Performance

ASDO can efficiently process *analytic* queries as it can respond to a large number of network distance queries. Figure 2.11 provides the time performance of Dijkstra's algorithm, CH algorithm, and our ASDO algorithm for an analytic query. Section 2.5.3 provides the KNN SQL query in detail. We have two relations in the PostgreSQL database. One relation contains 6,070 locations of universities from [7], and another relation contains 49,573 locations of fast food restaurants from [5]. The schema of both tables is *(id, Z_2 code, latitude, longitude)*, where we precomputed the Z_2 code.

The *KNN* query with which we experimented obtains the K nearest restaurants for each university in Figure 2.11(a), and the K nearest restaurants for each restaurant in Figure 2.11(b). ASDO first used PostGIS index to retrieve a candidate set of restaurants that have the potential to be the K nearest neighbors for each university (or restaurant), then computed the network distances for each university-restaurant (or restaurant-restaurant) pair. In particular, we restrict the search space around each university (or restaurant) to a small window. This window ensures that the K nearest neighbors must be located in the window. Note that the window size is different for each source location. Dijkstra's algorithm is implemented using a heap to speed it up. It starts at each university to search, and stops if the search

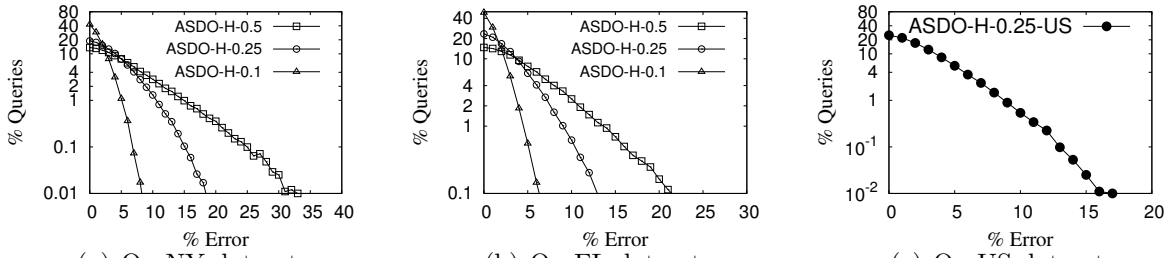


(a) 6,070 university sources (b) 49,573 restaurant sources

Figure 2.11: Response time for the KNN query, where the destinations of both (a) and (b) are 49,573 restaurants.

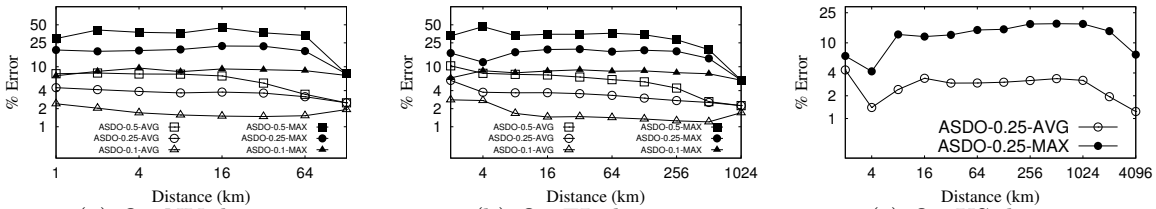
for this university has visited K restaurants. The CH algorithm finds all pairs of universities and restaurants, where the restaurant is in the window of the university, then computes the distances between the pairs, and finally sorts the result to get the top K restaurants for each university. ASDO computes the query in the same way as the CH algorithm.

From Figure 2.11, we can see that ASDO is much faster than CH. Although Dijkstra's algorithm is considered efficient for the KNN query, it is only faster than ASDO when K is very small, e.g, 5. For lager values of K, ASDO is at least $5X$ faster than Dijkstra's algorithm. ASDO has the additional advantage that a user can express complex queries using SQL in a few minutes, while doing the same with the other methods would require significant programming effort. For example, the SQL code of ASDO for the KNN query is $30X$ shorter than the C++ code of the Dijkstra or CH algorithms.



(a) On NY dataset (b) On FL dataset (c) On US dataset

Figure 2.12: Percentages of queries along with associated errors: it shows that very few of queries achieve the error bound ϵ. Thus, Although $\epsilon = 0.25$, which is not very small, it is sufficient for most queries.



(a) On NY dataset (b) On FL dataset (c) On US dataset

Figure 2.13: The maximum and average errors varying with their exact network distances

2.3.4.3 Accuracy with varying ϵ

Figure 2.12 describes the percentage of queries with particular errors for oracles for different datasets and built with ϵ values of 0.1 (10%), 0.25 (25%), and 0.5 (50%). All queries whose error percentage lie between $[x, x+1), x \in Z$ are grouped together. For example, the ASDO-H-0.25 lines in Figure 2.12 indicate that more than 20% of the queries contain less than 1% errors even though $\epsilon = 0.25$ (25%). From the above experiments, we believe that $\epsilon = 0.25$ is sufficiently accurate for most real applications as it provides a reasonable balance between decreasing disk usage and increasing accuracy.

Figure 2.13 provides another view on errors, which groups together the queries whose exact network distances lie in $[2^x, 2^{x+1})$ where x is an integer and 2^x is measured in kilometers. For each dataset and ϵ setting, we plot the maximum and average errors for every group. The maximum errors are close to the given error bound ϵ, but just few of queries attain the error bound. Note that the average error is relatively independent of the distance. This means that ASDO is good for both short and long distance queries.

2.4 Related Work

Latency approaches are designed to answer a single or a small number of shortest path or network distance queries on road networks. The original road network or a processed representation of it is stored in memory and queries perform operations on this in-memory representation. The most common latency approach is Dijkstra's algorithm [37]. Other methods [21, 22, 25, 30, 34, 39, 40, 43, 55, 61, 76] operate on the observation that some vertices in a spatial network are more important than others in answering shortest path queries. These methods offer different trade-offs between pre-processing time, storage, and query time. RE [40] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. HL [21] and m-hop [30] find hub nodes or distance labels such that the network distance between any two vertices can be computed by just checking their hub nodes or distance labels. DisLand [43] and LLS [55] find landmarks among network vertices to speed up network distance queries. [22, 25, 34, 39, 61, 76] build an explicit hierarchy graph to overcome

the drawbacks of Dijkstra's algorithm.

Precomputing ASDO requires knowledge of the network distances between some of the vertices in the road network. To do this, we use the CH method [39], which is one of the fastest available in-memory methods. CH also has a pre-processing stage where it computes an importance score for each node and then replaces some original edges by shortcuts. For a spatial network with n vertices, [39,74] show that CH takes $O(n)$ additional space to store this auxiliary information. For the full USA road network data, we found that CH's pre-processing stage takes about one hour and generates 24.5 million shortcuts. The response time for a single path and distance query is in the 0.1–10 ms range.

Other latency methods such as [69, 72] take advantage of the spatial information associated with the vertices and edges of a road network and use geometric techniques. *Road Network Embedding* (RNE) [69] applies a Lipschitz embedding [42] to a road network, such that vertices of the spatial network become points in a high-dimensional vector space. In this method, all operations on the road network happen in the high-dimensional space. [72] takes advantage of the fact that the shortest paths from vertex u to all other vertices can be decomposed into subsets based on the first edges on the shortest paths from u to them. This property is referred to as *spatial coherence* in ϵ-DO [64] and is used by [72] to speed up Dijkstra's algorithm. They store a few geometric objects for each vertex in the road network that can prune searches during run-time.

A characteristic of *throughput* methods is that the shortest paths and distances are precomputed so that the query process only requires a lookup as opposed to any

real computation on the fly. The resulting precomputed representation is large so that it is stored on disk thereby affecting latency. On the other hand, these methods are good for obtaining a high throughput since multiple lookups can be batched up at the same time thus increasing the number of queries that can be answered at the same time, although each query may take a bit more time.

Among the throughput methods, [64, 65, 67] exploit the spatial coherence of both sources and destinations in the sense that if a set of vertices are sufficiently far away, then distances between pairs of points in different clusters are similar. The *Path-Coherent Pairs Decomposition* (PCPD) [67] gives one exact shortest path algorithm, while the ϵ-DO [64,65] provides an approximate network distance with ϵ-error guarantees and $O(\frac{n}{\epsilon^2})$ space. [52] looks at the task of computing spatial analytic queries and experimentally compares their performance using the ϵ-DO architecture, where query processing is completely handled by an RDBMS, and using a hybrid architecture, where there are separate modules for the database, the road network, and a query analysis tool. SPDO [54] is another method of using ϵ-DO inside a distributed key value store such as Apache Spark. Another database-centric method is HLDB [20] which can answer exact network distance queries and full shortest-path even for an area as large as Europe containing 18 million vertices with complex SQL queries.

In HLDB [20], the authors mention that most of the memory-based latency approaches surveyed in [35] are difficult to embed into a database system and to query using SQL queries. This is because most methods rely on complicated data structures such as graphs and priority queues, which cannot be incorporated into a

database system in which the fundamental building blocks are relational operators. The main contribution of HLDB is the embedding of the memory-based hub labels (HL) [21] method into a database. The HL method precomputes the hub nodes for each vertex such that the distance between any two vertices s and t can be obtained given only their hub nodes. However, compared with the best previous database-centric oracle methods ϵ-DO [64, 65] and PCPD [67], HLDB has two drawbacks: 1) HLDB is not efficient if the average number of hub nodes per vertex is large; 2) Each spatial query in HLDB is a complex SQL statement that must perform a join operator on "forward" and "backward" tables, so that HLDB cannot guarantee query responses within a time bound. ϵ-DO uses much simpler SQL statements for the query but in its original formulation takes quite a lot of time to build. In this chapter we showed how to dramatically speed up its construction.

Wu et al. [74] evaluate several state-of-the-art methods (i.e., [25, 39, 64, 67]) for computing road network distance in the same environment. Even though they do not make the distinction between latency and throughput methods, and only compare all the methods from a latency perspective, there are some valuable lessons to be learned from this work. It shows that TNR [25] and CH [39] have fast preprocessing, low space overhead, support for real time queries, and the ability to easily handle continental road networks with tens of millions of vertices. This inspired our decision to use CH [39] for precomputing ϵ-DO. They also point out that although ϵ-DO and PCPD are better for answering queries, they are not practical because they they are too expensive to precompute. This chapter remedies this perceived deficiency of ϵ-DO and enables it to scale to handle continental road networks such as the entire

USA.

2.5 Querying ASDO in a Database

2.5.1 Function Creation

The function Z_2 computes the two-dimensional Morton codes from the latitude and longitude coordinate values by first normalizing them to real values that lie between $[0, 1)$. Next, it takes the binary representation of the decimal values, and converts it to an L digit bits representation. This is following by a bit-interleaving operation which takes a pair of integers and interleaves the bits to generate a 32-bit number. In a similar manner, here we provide the shuffle function that takes two Z_2 integer numbers and generates a bit-interleaved 64-bit number Z_4^0. The shuffle function performs a series of bit masking and shifting operations to interleave the bits and is very efficient.

```
CREATE FUNCTION shuffle (x bigint, y bigint)
       RETURNS IMMUTABLE bigint AS $$
BEGIN
  x := (x | (x<<32)) & X'00000000ffffffff'::bigint;
  x := (x | (x<<16)) & X'0000ffff0000ffff'::bigint;
  x := (x | (x<<8))  & X'00ff00ff00ff00ff'::bigint;
  x := (x | (x<<4))  & X'0f0f0f0f0f0f0f0f'::bigint;
  x := (x | (x<<2))  & X'3333333333333333'::bigint;
  y := (y | (y<<32)) & X'00000000ffffffff'::bigint;
```

```
  y := (y | (y<<16)) & X'0000ffff0000ffff'::bigint;
  y := (y | (y<<8))  & X'00ff00ff00ff00ff'::bigint;
  y := (y | (y<<4))  & X'0f0f0f0f0f0f0f0f'::bigint;
  y := (y | (y<<2))  & X'3333333333333333'::bigint;
  IF x > y THEN
    RETURN (y << 2) | x;
  ELSE
    RETURN (X << 2) | y;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

2.5.2 Basic Query Example

We provide two variants of the *basic* query that retrieves the network distance for a pair of vertices. ASDO of the US dataset is stored as the *oracleusa* relation, which has a schema of $(code, d)$. In the simplest case, the input is a key corresponding to a four-dimensional Morton code.

```
CREATE FUNCTION dist (bigint)
       RETURNS real LANGUAGE sql IMMUTABLE AS
  'SELECT d FROM oracleusa where code <= $1
   ORDER BY code DESC LIMIT 1';
```

A variant of *dist* is in the case that the input corresponds to the latitude/-longitude of the source and destination, respectively. A possible use case here is

a mobile host on a road network that is computing a network distance to another mobile host. The inputs to this function are four real values corresponding to two latitude and two longitude values.

```
CREATE FUNCTION dist(real, real, real, real)
       RETURNS real LANGUAGE sql IMMUTABLE AS
   'SELECT dist(shuffle(Z2($1,$2), Z2($3, $4)))';
```

2.5.3 Analytic Query Example

Next we illustrate two examples of the analytic query. One is the KNN query, and the other is the "walk score" query. More examples are provided online in `http://roadsindb.com/`.

In the KNN example, we have two location tables, named *University* and *Restaurant* with schema $(id, code, lat, lon)$. The KNN task is to find the top K nearest restaurants for each university in terms of network distance. In order to avoid computing all pairs of *University* and *Restaurant*, we need to compute a candidate set of restaurants that have the potential to be the K nearest neighbors for each university. Thus, we decomposed the task into two steps.

Step 1: Create a view named *kdn*, which computes the Euclidean distance to the K neighbors using the GiST index and then compute the maximum network distance among K neighbors for each university. Note that here we translate distance d to deg degrees assuming that 1 degree of latitude/longitude roughly equates to a geodesic distance of 111 km.

```
CREATE VIEW kdn AS
SELECT y.id as id, y.lat as lat, y.lon as lon,(
  SELECT max(dist)
  FROM (
     SELECT dist(x.lat, x.lon, y.lat, y.lon)
     FROM Restaurant x
     WHERE x.gid != y.gid
     ORDER BY x.geom<->st_setsrid(y.geom, 4326)
     LIMIT K
  ) as foo
) / 111000 as deg
FROM University y
```

Step2: Compute all nearest restaurants for each university whose Euclidean distance is less than or equal to d and then use *dist* to compute their corresponding network distances and retain the K closest ones.

```
SELECT * FROM (
  SELECT src, dst, dist, ROW_NUMBER() OVER
         (PARTITION BY src ORDER BY dist) AS KNN
  FROM (
    SELECT kdn.id AS src, R.id AS dst,
           dist(kdn.lat, kdn.lon, R.lat, R.lon)
    FROM kdn, Restaurant R
    WHERE R.lat between kdn.lat - kdn.deg
                    and kdn.lat + kdn.deg AND
          R.lon between kdn.lon - kdn.deg
```

```
                    and kdn.lon + kdn.deg;
  ) AS foo1
) AS foo2
WHERE KNN <= K
```

This method is correct because the Euclidean distance is a lower bound on the network distance. The lower bound property guarantees that we find the KNN within the candidate set.

Consider an analytic query that computes a "walking score" for houses. This is a popular feature that is offered by many real estate websites (e.g., Zillow). Suppose we have relations *House*, *Mall* of shopping malls, and *University* all with our usual schema of $(id, code, lat, lon)$. We define the walking score as the length of the shortest trip that first includes a visit to a mall and then a visit to a university. The task is to display the walking score for each house. Note that this query is complex since choosing the closest mall to a house may not produce the smallest trip distance. We can express such complex queries by writing just a few lines of SQL as we show below, and which took us less than 10 minutes to write.

```
SELECT foo1.src, MIN(dist1 + dist2) AS score
FROM (
     SELECT x.id AS src, y.id AS dst,
            dist(shuffle(x.code, y.code)) AS dist1
     FROM House x, Mall y
     WHERE y.lat<=x.lat+C AND y.lat>=x.lat-C AND
           y.lon<=x.lon+C AND y.lon>=x.lon-C
```

```
) AS foo1, (
     SELECT x.id AS src, y.id AS dst,
            dist(shuffle(x.code, y.code)) AS dist2
     FROM Mall x, University y
     WHERE y.lat<=x.lat+C AND y.lat>=x.lat-C AND
           y.lon<=x.lon+C AND y.lon>=x.lon-C
) AS foo2
WHERE foo1.dst = foo2.src
GROUP BY foo1.src
ORDER BY score DESC
```

where C is a rational value to restrict the search space around each house and each mall to a window with dimensions $2C \times 2C$. The above SQL query first generates *foo1* where for each house it computes the network distances to a few nearby malls. Then it generates *foo2* where for each mall it computes the network distances to a few nearby universities. Finally, it joins *foo1* and *foo2* based on the destination of *foo1* being the same as the source of *foo2* (i.e., same mall). Then group the results based on the house *id* to get the minimum "walk score" for each house.

2.6 Summary

We presented a new distributed framework, named ASDO, that computes the ASDO representation efficiently so that it can be done in just a few hours with a modest size cluster. The ASDO representation resides inside a database system and can be used. Although our distance results are approximate bounded

by ϵ, different applications can choose to balance the accuracy requirement and disk usage according their own budget. So we offer a more flexible choice. At the present, we achieve the basic functionality of systems such as Google Maps at a cheaper cost, and surpass the previous systems since we allow users to create new types of spatial queries in any declarative language. Our method in contrast to ϵ-DO can easily handle a road network of the size of the USA containing more than 24 million vertices. It produces stable results and does not need any modification to the database in which it is embedded. Compared to HLDB, it is two orders of magnitude faster since it does not need to perform an expensive join operation that HLDB performs. The *latency* of our method is relatively independent of the size of the underlying road network and even smaller than state-of-the-art memory-based methods (e.g. the CH method) when the network distance is reasonably long (e.g., greater than 30 kilometers). Furthermore, the *throughput* of our method greatly outpaces memory-based methods reaching about $65,000$ queries/second with just a single commodity machine. Thus our method can significantly speed up complex road network analytic queries in any database system, so that users can create new types of spatial queries using any declarative language.

Some directions for future work include the following. The first is to measure the SQL query performance for more complicated spatial queries such as points of interest (POI) queries which are frequently used in location-based services. The second is to compare the performance of ASDO under different database systems as the architecture of each database system provides different but ample opportunities for optimization. So, a guide for the use of ASDO on different types of databases is

needed. The third is to allow efficient updates of vertices and edges on ASDO for incorporating the changes in the road networks due to factors such as road closures. This requires devising ways of computing the oracle in piecemeal-fashion so as to avoid having to compute it from scratch as we currently do. A similar problem has been proposed recently [51], but it only considers the Euclidean distance metric. The fourth is to develop other related oracles such as ones that deal with travel times or traffic obstruction times. Addressing such problems would make the underlying spatial database systems more powerful.

Chapter 3: An Experimental Evaluation of Two System Architectures for Analytic Queries on Road Networks

3.1 Overview

In this chapter, we present a general efficient solution for spatial analytic queries. Our contribution is two-fold. The first is the proposal of two architectures, the hybrid architecture (HY) and the integrated architecture (DO) using our ASDO representation from the previous chapter, for solving spatial analytic queries. The second is the formulation of efficient solutions for spatial analytic queries using ASDO.

We propose the architectures and their modules by reviewing and summarizing the existing solutions and use cases. Most existing spatial analysis tools use the HY architecture illustrated in Figure 3.1, which separates the modules into two parts. The first part deals with point-of-interest (POI) locations, relations, and attributes in a database system, and while the second part retrieves shortest distance results on a road network, which are usually processed in memory. Embedding our ASDO representation into a database system makes the DO architecture in Figure 3.2 possible. Embedding map-based services within a database system is attractive as

it allows developers to leverage the power of a database language to create new types of online services resulting in easy programming, customization, and maintenance. In addition, there are ample opportunities to use optimization to speed up a spatial analytic query like finding the nearest restaurant for each coffee shop which ends up making millions of shortest path queries.

In our examples, each spatial analytic query is expressed by just a few lines of SQL that utilize pre-defined functions. In contrast, the situation is far more complicated if for each of the queries users would have to devise efficient programs in Java (or other high level programming languages) to obtain the necessary query results.

In this chapter, we experimentally evaluate our ASDO solution in a database in conjunction with a high-performance implementation of Dijkstra's algorithm with multi-threads on the entire USA road network. Dijkstra's algorithm is the most widely used method for spatial analytic queries (e.g., in Esri [4]) as it is adaptable for most spatial analytic queries and is efficient for the single source query.

The rest of this chapter is organized as follows. Section 3.2 introduces the analytic queries and provides an overview of our two architectures. Section 3.3 presents the traditional hybrid architecture for most existing spatial analysis tools, while Section 3.4 proposes the integrated architecture using our ASDO representation. Section 3.5 reports on a detailed experimental evaluation of our solutions, and we provide our SQL functions and codes online in [17]. Section 3.7 summarizes related works. We draw conclusions and review lessons learned in Section 3.8.

3.2 Spatial Analytic Queries and Applications

A spatial analytic query on a road network performs hundreds of thousands or even millions of shortest distance computations in the process of answering the query. These types of queries are commonplace in many applications such as logistics, tour planning, and the determination of service areas. Below, we provide a few sample use cases gleaned from real user postings on ESRI [4] web boards and those that we found elsewhere on the Internet.

Use Case 3.2.1. *I am a taxi operator running a fleet of taxis. I have a dataset of taxi trips each with a unique ID such that each trip has a latitude and longitude values for both a pickup and a drop off point, as well as for way points at irregular intervals. Such a dataset constitutes the trajectory information for each taxi trip. I want to obtain the total number of miles travelled by each taxi this month. This information is useful in computing the actual profit per km of all the vehicles in my fleet and determining which of my drivers are better performers.*

Use Case 3.2.2. *I am an operator of a large hospital and have the geocoded address of my patients and the locations of my clinics. Our hospital has more than 500 clinics across the country. Each patient is assigned to the nearest clinic. I want to get the average drive time for patients per clinic. This is an important metric in healthcare since the further one has to drive, the poorer are the health outcomes. This distance also informs us of the need to open new clinics or relocate existing ones to better serve our patients.*

Use Case 3.2.3. *I have a trucking company with 10 trucks that deliver thousands of packages for a popular retailer. A common operation that I run several times during each day is determining which packages should be loaded on to which truck and the order in which they should be delivered. This decision process constitutes a complex tour planning query that tries to minimize the total network distance travelled by all trucks, as well as to accommodate the priority assigned to each package. For this purpose, the input is a network distance matrix between the delivery locations of all current packages. An optimization program would decide how to assign packages to trucks and the order in which to deliver them. Note that even with 1000 packages, the query will compute up to 1 million pair-wise network distances.*

Use Case 3.2.4. *Our company has 2,800 maintenance locations in the state of Virginia. I want to cluster these locations into 2, 3, or 5 mile groups. For example, locations in the same 5 mile group can be reached by driving no more than 5 miles from a center point.*

For spatial analytic queries on road networks, there are two common reasons why such queries end up making a very large number of distance computations. First, spatial analytic queries are typically used for generating insights into the data in the form of reports or visual representations. So it is common for these queries to end up accessing large portions of the data. Second, the queries may join two or more datasets on the basis of the network distance to other objects on the road network, such as finding the nearest neighbors from one dataset for each location in another dataset, or group one or more datasets based on the closest distance

to objects in another dataset. Executing all of these operations can easily end up making millions of distance computations on the road network. For instance, just the simple query that obtains the network distance between all pairs of objects drawn from a set of 1000 objects to one another ends up making 1 million distance computations on the road network.

Since the spatial analytic queries are an important use-case whose efficiency depends on being able to perform millions of network distance computations efficiently on road networks, there is a need to examine which of existing available architectures are capable of efficiently processing these queries. Most existing tools for spatial analytic queries have several limitations, or use the basic Dijkstra's algorithm. For example, the Google Distance Matrix API [8] limits non-paying users to submit 100 shortest distances (10 origins and 10 destinations) per query, and obtain $2,500$ shortest distances per 24 hour period. For paying customers, the limits are 625 shortest distances per query, and $100,000$ shortest distances per 24 hour period. Esri [4] also claims that the ArcGIS Network Analyst extension, namely the Route, Closest Facility, and OD Cost Matrix solvers, are based on the well-known Dijkstra's algorithm for finding shortest paths. All these tools are not good enough to solve spatial analytic queries.

Spatial analytic queries make two distinct kinds of access patterns on road networks, and make millions of these accesses in the process of answering a query. The most basic pattern is called *one-to-one* pattern which computes the distance between a source and a destination on the road network. Another access pattern is *one-to-many* that makes several s-t pair computations from the same source ver-

tex. For instance, computing the K nearest neighbors for each point from a large dataset makes one-to-many access patterns. There are opportunities for speeding up one-to-many patterns even though they are nothing more than multiple one-to-one access patterns. We use the term *scan* to describe the actual implementation of the execution of an access pattern. Note that there can be many options for executing a scan including Dijkstra's algorithm, contraction hierarchies (CH) [39], etc.

Any architecture for answering spatial analytic queries must be optimized for performing a large number of distance queries on the road network. In particular, we present two such architectures and then compare and contrast their features.

The first architecture is a hybrid architecture that uses a database to store and query spatial datasets, but then uses an external module that loads the road network in the main memory and performs fast in-memory scans on the road network. This approach takes advantage of the large amount of available memory in modern computers as well as the high number of processing cores to be able to compute a large number of scans quickly. An analysis tool coordinates the data transfer and the issuance of scans to the road networks. A common example of such an approach is the process used by the ArcGIS Network Analyst to solve problems such as Use Case 3.2.2. The information pertaining to both clinics and patients is maintained in a database system. In order to compute the average driver distance for each clinic, the ArcGIS Network Analyst first retrieves the related data from the database, and then scans the network starting at the clinic using Dijkstra's algorithm which here is implementing a one-to-many access pattern. The scan process stops when it has obtained all the network distances of the clinic's patients.

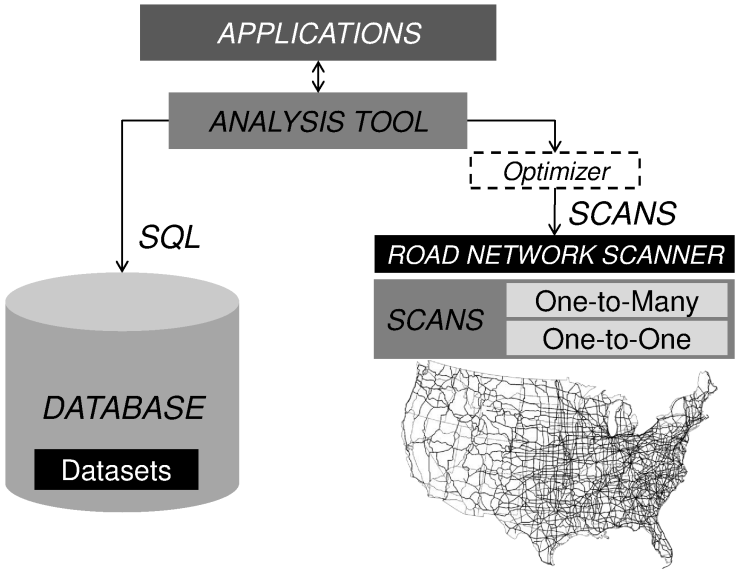


Figure 3.1: The HY architecture, which represents most existing spatial analysis tools

The second architecture incorporates the road network inside the database as a single relation. The road network is stored as the ASDO relational table indexed by a B-tree. Scans on the road network become lookups on the B-tree index which is very efficient to perform. This method relies on being able to perform the queries entirely inside a database and on using the declarative nature of an RDBMS to automatically optimize queries.

3.3 Hybrid Architecture

The most common architecture for responding to spatial queries on a road network is a hybrid (denoted by HY) one consisting of a database to store the spatial datasets and a module external to the database to execute the actual operations on the road network. Figure 3.1 shows such a representative architecture that combines

a database, an analysis tool, and an external module for network processing. An example of a system that deploys such an architecture is ArcGIS from Esri [4], a popular platform for performing deep analysis to make informed decisions. The analysis tool is at the heart of this architecture in the sense that it extracts the necessary data from the database, pre-processes it, and contacts the road network scanner to perform the necessary scans on the road network. In this architecture, the analysis tool partitions the query processing into two parts. The first part queries the database to access the spatial datasets, such as restaurants, gas stations, real estate information, warehouses, etc. When the volume of datasets is large (which is the usual case), this database is usually a conventional off-the-shelf database; if their volume is small, then they may even be loaded into main-memory from files (e.g., shapefiles) in which case we have a main-memory database. The second part uses the analysis tool's road network module to compute network distances between the objects. Actually, the network distance computations are incorporated into the processing and the result is either displayed to the user or stored back into the database. In this model, the network processing happens entirely outside the database while the analysis tool serves as the "glue" that coordinates computations between the database and the road network module.

The road network scanner is an in-memory processing module that implements the execution of the access patterns on large road networks. It contains at least one spatial index such as a k-d tree or R-tree to locate the given locations, and one or more priority queues to speed up the scan process. In order to fully utilize the computing power of multi-cores, this module also needs to employ several processing

threads. Each thread responds to one scan process at a time. Its not worth to parallelize the workload inside one scan process using several threads as previous work [33,44] has shown that parallelization of Dijkstra's algorithm and similar scan-based algorithms with traditional locks and barriers has disappointing performance. In particular, our implementation of HY built a k-d tree in the main thread which was pre-loaded with the vertices of the graph. The main thread uses the k-d tree to locate the source point of a given scan task, while the destination points of the scan task are obtained from the POI table. Next, our implementation sets up several scanning threads to process Dijkstra's algorithm (we could have also used another method like CH) to obtain distance results. All scanning threads share the graph representation, and each thread keeps a private scanned queue to store the visited vertices and a heap (as we are using Dijkstra's algorithm) to determine the next vertex to scan. Each time that the main thread is presented with a scan task, the main thread first locates the source point using the k-d tree, and then assigns the scan task with its associated source and destination points to a waiting scan thread. All threads use busy-waiting.

In addition, between the analysis tool and the road network scanner, an optimizer module would be useful to automatically optimize the scan plan. However, often, users dispense with this step as most existing analysis tools rely on the user's specification of the scan plan. At the end of Section 3.4, we give an example to see the importance of having an optimizer.

The road network scanner in Figure 3.1 shows two access patterns for retrieving shortest distances. The one-to-one access pattern is the commonest, although the

one-to-many access pattern is also commonly used as it is optimized for multi-destinations. Note that ideally there should also be a third access pattern of the form many-to-many. However, very few access pattern implementation algorithms are designed for a many-to-many access pattern because its effect can be obtained by resorting to multiple instantiations of the one-to-one and one-to-many access patterns as in [41].

Each specific algorithm that retrieves the shortest distances and paths implements one of the two access patterns. For example, Dijkstra's algorithm is good for the one-to-many access pattern, while CH [39] is good for the one-to-one access pattern. The road representation of a specific algorithm is the lowest component in Figure 3.1, e.g., Dijkstra's algorithm uses the original graph representation and TNR [25] uses the hierarchical tree representation.

Below we provide an abstraction of the operation of the scan procedures instead of the details of the algorithms that implement them. In particular, our HY architecture given in Figure 3.1 implements the following operators on road networks:

Definition 3.3.1. *The* SCAN*() operator scans the road network in memory using one of the scan-based algorithms, $\mathcal{A}$. Given $G(\mathcal{A})$, the graph representation of $\mathcal{A}$, and vertex s,* SCAN*(s) uses $\mathcal{A}$ to scan $G(\mathcal{A})$ starting at s.*

We now define two operators that are frequently used in the spatial analytic queries, SCAN_UNTIL_K() and SCAN_UNTIL_DIST(). They inherit the SCAN() operator.

Definition 3.3.2. SCAN_UNTIL_K*(k, s, P) scans the graph starting at vertex s, and returns the k nearest objects o to s, where $o \in P$ and P is the POI set.*

Definition 3.3.3. SCAN_UNTIL_DIST*(d, s, P) scans the graph region within network distance d from vertex s, and returns all the objects o, where $o \in P$ and P is the POI set, and o is within network distance d from s.*

SCAN_UNTIL_K() stops the scanning when it has visited k objects in P, and SCAN_UNTIL_DIST() limits the scanning to objects lying within a specified network distance. The set of P indicates the set of POIs, which is an overlay over the network graph. It can be a set of restaurants, gas stations, houses, as well as even all the vertices of the road network. As far as we know, every spatial analytic query contains at least one set of POIs. After defining the SCAN() operators, we can easily describe the processes of spatial analytic queries using a scan-based algorithm. For example, a KNN query can be solved by the SCAN_UNTIL_K(k, s, P) operator, and a distance matrix query, which has n sources and m destinations, can be solved by making n calls to SCAN_UNTIL_K(m, s_i, P) where s_i is the i^{th} source.

3.4 Integrated Architecture

The database community desires an integrated architecture, which means that all components and procedures reside in a database. This makes the architecture more compact and efficient, as the analytic query executes entirely within the database. The database knows how to optimize such queries, since the road representation appears as one or several relations in the database, and thus the query

appears like any other relational query to the database. The core challenge lies in how to embed the road representation in the database. For example, pgRouting [13] extends the PostGIS / PostgreSQL geospatial database to provide geospatial routing functionality. Oracle Corporation proposed the Network Data Model Graph (NDM) [49], which persistently manages the network connectivity in the database, while a Java API provides fast in-memory graph analytics. However, since the road representation in both pgRouting and NDM is the original graph representation, they are similar to the HY architecture except for storing the nodes and edges in their database. HLDB [20] proposed by Microsoft Research is the first practical system that can answer spatial queries on continental road networks stored entirely within a database. It stores the vertices of the road network, as well as sets of "forward" and "backward" hub labels (HL) of the vertices [21] in the database. Each $s - t$ query is solved by performing a JOIN on the corresponding "forward" and "backward" relational tables of s and t, respectively.

In this section, we propose the integrated architecture that makes use of our ASDO technology. The ASDO takes a road network as input, and reduces it to a single database relation that captures the network distances between every pair of vertices in the road network, which we introduced in Chapter 2. Using a cluster of 20 EC2 machines, it took us about 7 hours to compute the ASDO representation of the USA road network which contained 24 million vertices. The precomputation process decomposed the road network with n vertices into $O(\frac{n}{\epsilon^2})$ triples (A, B, d) stored in a relational table, such that A and B are denoted by blocks in a PR quadtree and d is the network distance that approximates the network distance between every

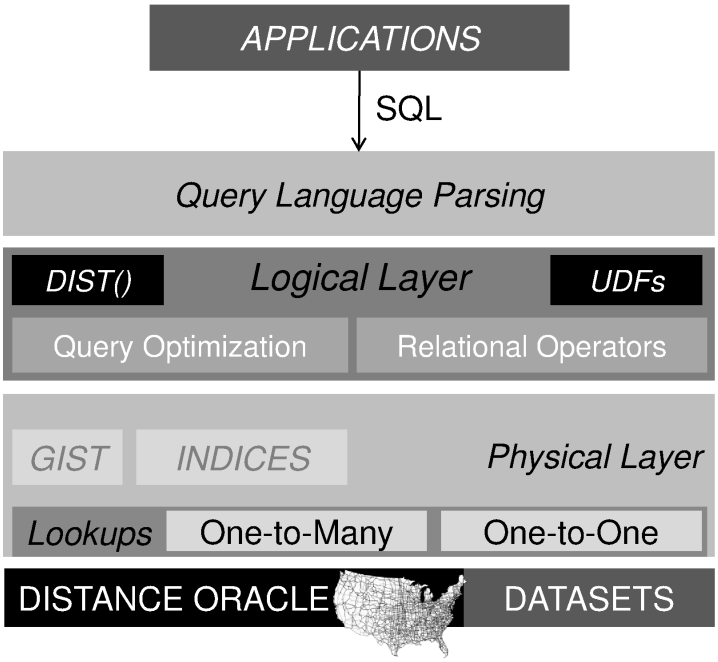


Figure 3.2: Integrated architecture DO for analytic queries using the distance oracle.

pair of vertices contained in A and B within an ϵ error tolerance. In particular, d is also said to be ϵ-approximate, which means that the resulting error in using d instead of the exact network distance between the any vertex in A and any vertex in B is bounded by the ϵ error tolerance. The resulting representation for the entire USA was about 55GB in size with an error tolerance $\epsilon = 0.25$. This is a reasonable setting for real road networks. In particular, in Section 2.3.4.1, we showed that although the error tolerance ϵ is 0.25, the approximate distance value of at least 20% of the vertex pairs has an error of less than 1%. Moreover, the average error for random queries is just 2.74%. The relational table corresponding to the distance oracle is indexed by a B-tree representation that allows disk efficient loopkups for approximate distances.

Using the distance oracle, we created an integrated architecture illustrated in Figure 3.2. The key difference from the hybrid architecture is the use of the distance oracle road representation, which has been embedded in a database as a simple relational table. To query the distance oracle, we implemented an SQL function called DIST(), which queries the distance oracle relational table to compute the road distance between any source and destination. In particular, given two latitude/longitude pairs, DIST() first computes a unique code which it looks up in the distance oracle relational table, and then uses a simple SELECT query that is facilitated by the B-tree index. For example, computing the network distance between the White House and the US Capitol Building in Washington, DC becomes as simple as the following query that

```
-- Road distance between White House and US Capitol
SELECT DIST(38.8977, -77.0366, 38.8898, -77.0091);
-- This produces 2144.7 (meters)
```

More user-defined functions (UDFs) and complex queries can also be easily expressed using the distance oracle. Let us consider the following example. Suppose that we have a relation `houses (id, lat, lon)` corresponding to the location of all houses available for sale and another relation `parks(id, lat, lon)` corresponding to the location of all parks, where lat and lon correspond to the latitude and longitude values of the corresponding locations. We want to find up to 100 houses with the maximum number of parks that lie within 0.5 km of road distance from the houses sorted by the number of such parks. The following code written completely

in SQL yields an efficient response to this query.

```
SELECT id, count(*) as count
FROM ( SELECT houses.id as id,
          DIST(houses.lat, houses.lon,
               parks.lat, parks.lon) as distance
       FROM houses, parks
     ) as foo
WHERE distance < 500  -- 0.5 km in meters
GROUP BY id
ORDER BY count DESC
LIMIT 100;
```

Here we see how to express a complex query with just a few lines of SQL. Now contrast this with performing the same query in the traditional setup which uses a module where the road network would be stored externally as in Figure 3.1. The road network would typically be accessible through an API such as SCAN_UNTIL_K() and SCAN_UNTIL_DIST() for computing shortest paths and distances. In this setup, the hybrid architecture would first obtain a table of houses and parks from the database. Next, we have two options to obtain the distance results using the SCAN_UNTIL_DIST(). The first is for each house, to compute the number of parks within 0.5 km. The second is for each park, to compute the houses within 0.5 km, and then to group the distance results by the house ids. Finally, for each house, count the number of parks and order them in descending order of the count.

Although the first option is straightforward for this query, It turns out that

the second option is more efficient as the number of parks is considerably smaller than the number of houses, which means the number of scans is lower. This example demonstrates that we need an optimizer for the hybrid architecture to decide the order of execution and make the query execution plan. This depends on being able to do selectivity factor estimation. On the other hand, the integrated architecture has the bonus of having a query optimizer as part of it although we did not need it in this example. In summary, the effort to implement a spatial analytic query as the above example in the hybrid architecture is considerably more complex than writing a few lines of SQL as in the integrated architecture.

3.5 Experiments

In this section, we present a detailed evaluation of the two architectures in order to compare and contrast their query performance. Section 3.5.1 describes the experimental setup and datasets. Sections 3.5.2 and 3.5.3 evaluate the performance of both HY and DO for the region and throughput queries, respectively. We synthetize the queries in these two subsections from the POI tables that we used. Next, Sections 3.5.4 and 3.5.5 show our solutions for the KNN and trajectory queries, respectively, in realistic settings.

3.5.1 Experimental Setup and Datasets

The integrated architecture (DO) is completely self contained in a PostgreSQL database system, where analytic queries can be expressed in SQL. It exposes a single

function DIST(lat1, lon1, lat2, lon2) that will return the ϵ-approximate network distance between a source and a destination location. The distance oracle for the whole USA was used for this experiment with $\epsilon = 0.25$ and is 55GB in size. Although this number seems large, the fact that the road network has close to 24 million vertices shows that this number is consistent with our predicted linear storage bound of $O(\frac{n}{\epsilon^2})$ space where n is the number of vertices in the road network.

The hybrid architecture (HY) uses an entirely in-memory implementation that compactly stores the spatial datasets, the USA road network, and the road network scanner. The road network scanner implements an efficient multi-thread implementation of Dijkstra's algorithm and it defines the SCAN() functions, SCAN_UNTIL_K() and SCAN_UNTIL_DIST().

We rented one Amazon RDS *db.m3.2xlarge* DB instance with PostgreSQL 9.3.5 for the DO architecture. For the HY architecture, we rented one Amazon EC2 *m3.2xlarge*. Both of these machines have identical hardware specs (8 vCPU and 30 GB memory) and were used in their default settings. The USA road network was from the 9^{th} DIMACS Implementation Challenge [3], which contained $23,947,347$ vertices and $58,333,344$ edges.

We used two POI tables for the evaluation. The *restaurant* table consists of $49,573$ fast food restaurants obtained from [5], and the *university* table consists of $5,964$ locations of universities from [7]. The schemas of both tables are identical and are *(id, latitude, longitude, gid, geom)*, where `gid` and `geom` are needed for the GiST index on the latitude/longitude values.

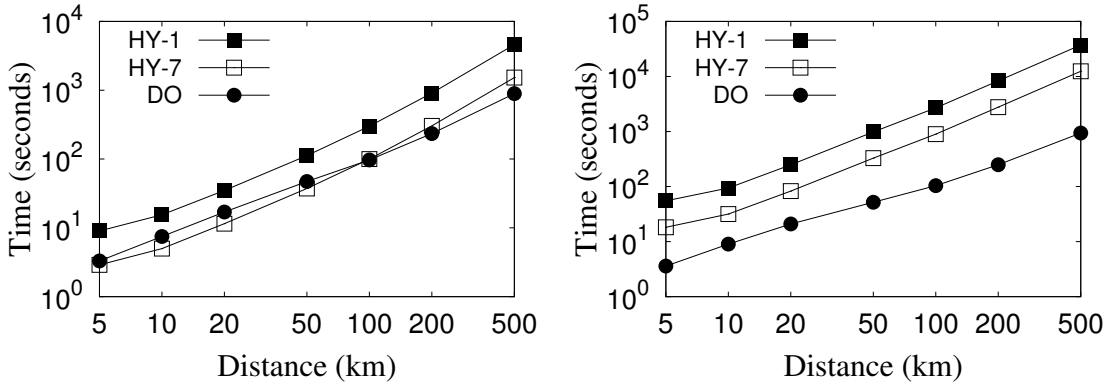
We also used a taxi trajectory dataset. This taxi dataset was from San Fran-

cisco Yellow Cab [1] collected by CRAWDAD [2]. It contained 11,220,058 GPS entries for 537 taxis covering a one month period in 2008 comprising 928,307 trips. The schema for the taxi trajectory relation is given in Table 3.1.

Table 3.1: Schema for table *taxi* storing the taxi GPS information.

Attribute	Explanation
id	crumb id (unique key, we added)
taxiid	each taxi has a unique id
tripid	globally unique trip id (we added)
lat	latitude in degrees
lon	longitude in degrees
occupancy	does cab have a fare? (1 = occupied, 0 = free)
ts	UNIX epoch time when GPS was recorded

An example tuple is as follows: [id, taxiid, tripid, lat, lon, occupancy, time], e.g.: [112133, 1, 422, 37.75134, -122.39488, 0, 1213084687]. Each taxi periodically records a GPS record on the server. We assume that each taxi takes the shortest path between successive GPS crumbs. Therefore, reconstructing the trip involves ordering the points by their timestamp (ts) (or equivalently by their id since we assigned the ids in order of increasing timestamp), thereby obtaining the road network distance between successive points and adding up the distance values.



(a) (b)

Figure 3.3: Time comparison between HY and DO varying with the farthest distance values for (a) restaurant is destination, and (b) university is destination

3.5.2 Region Query

A distance query returns the destinations lying within a given network distance of X kilometers around a given location denoted by its latitude and longitude values. The example query we use for evaluation here is one that for each university, finds all restaurants lying within X kilometers.

To solve this query, HY invokes the SCAN_UNTIL_DIST() operator that for each of the universities, scans the graph until obtaining all vertices within X kilometers from the university. This operation is very efficient and limits the scans to one per university.

In the DO architecture, to avoid computing the distance between all pairs of universities and restaurants, we need a simple way of reducing the number of invocations to the distance oracle. One way to do this is to take advantage of the fact that the Euclidean distance is a lower bound on the road network distance and

thus we restrict the pairs of objects that we examine to be within their Euclidean distance. This is achieved by using a query search window of width $2X$ around each university and only examining the restaurants lying in it. This translates to a query window of width $2X/111$ degrees assuming that 1 degree of latitude/longitude roughly equates to a geodesic distance of 111 kms. The SQL statement for this query is as follows.

```
SELECT *  FROM
   (SELECT x.id as id1, y.id,
           dist(x.lat, x.lon, y.lat, y.lon) as d
      FROM University x, Restaurant y
     WHERE y.lat between x.lat-deg AND x.lat+deg
       AND y.lon between x.lon-deg AND x.lon+deg
     ORDER BY dist
   ) as foo
WHERE d<X
GROUP BY id1
```

Figure 3.3 shows the execution time of DO and HY when varying the width of the query region. For HY, we show the performance of running 1 and 7 scanning threads using HY-1 and HY-7 respectively. The reasons for using 7 scanning threads are explained in Appendix 3.5.6. Figure 3.3(a) shows the results of region queries that find the restaurants within X kilometers of each university, while Figure 3.3(b) interchanges the sets that form the sources and destinations so that now we find the universities within X kilometers of each restaurant.

This experiment is to HY's advantage in the sense that it can amortize the costs of the scans from a single source to multiple destinations. Nevertheless, Figure 3.3(a) shows that for smaller values of the distance X, HY is slightly better than DO but this advantage vanishes as X increases with DO performing better than HY for $X > 100$. The setting of Figure 3.3(a) where we find the restaurants near the universities represents the worst case for DO as we expect many restaurants to be clustered around each university compelling DO to query the distance oracle once for each pair. On the other hand, when we change the setting so that we find the universities near the restaurants as in Figure 3.3(b), we find that the execution time of HY is at least one order of magnitude greater than DO since there may not be too many universities around each of the restaurants thereby greatly reducing the number of queries to the distance oracle. While this experiment shows that DO is sensitive to the density of the destinations, in the worst case it still performs as well as HY, while easily outperforming it in other cases.

To further explore the effect of density on DO's performance, we performed the following experiment. Define *density* to be the ratio of the number of destinations found to the number of vertices of the road network visited by HY during its scan around each source point for a given region. In some sense, density controls the *work* efficiency of HY vis-a-vis DO in that the larger the density, the greater is the benefit obtained by HY from amortizing the scans. Figure 3.4 shows the execution times of HY and DO when performing the region query that finds all synthetically generated destinations within 50 km of all universities. We created destinations around each university by generating points with a probability denoted by the density value.

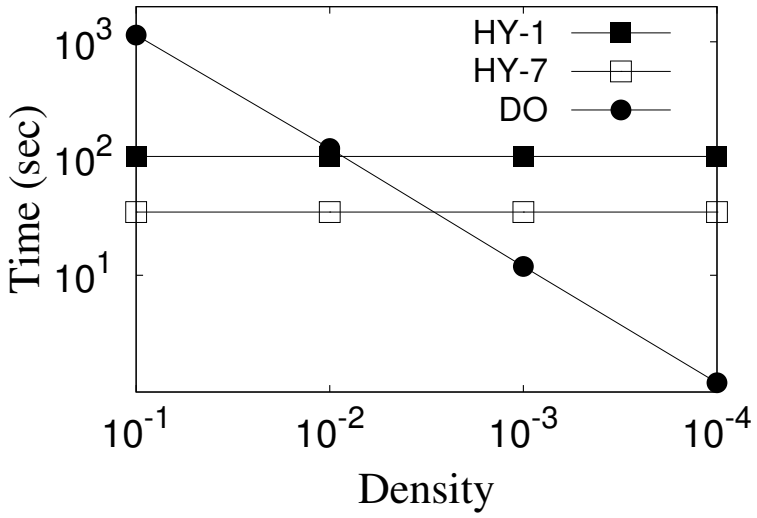


Figure 3.4: Execution time versus a synthetically varying density of destinations for 5,964 distance queries (corresponding to the size of the university sources relation at distance 50 km.

Figure 3.4 shows that the density does indeed affect DO as we had expected but does not affect HY for the region query. In particular, as the density decreases (i.e., the points become sparse), DO improves dramatically as the number of invocations of the distance oracle is greatly reduced. Note that although DO is slower when the density is larger than 0.01, it is fairly obvious that for real world datasets, a density of more than 1 restaurant per 100 vertices is extremely large to be realistic.

3.5.3 Throughput Query

From the previous results we see that DO provides a single DIST() function that computes the road distance between any pair of source and destination locations on the road network. HY, on the other hand, is optimized for one-to-many distance

computations since the scan amortizes the work done for scanning from a single source to multiple destinations. To better understand the performance of each architecture we compare them using a distance matrix query.

Table 3.2: Comparison between s-t pair and one-to-many

Query	Metric	DO	HY-7
Distance Matrix	Time	8853.9 sec	20139 sec
	Throughput	33392 dist/sec	14680 dist/sec
10k random pairs	Time	0.327 sec	2026 sec
	Throughput	30581 dist/sec	4.9 dist/sec

In this query, we use the university dataset as the source locations and the restaurant dataset as the destination locations. The query computes the distance matrix from all universities to all restaurants. The query can be executed by either performing $5,964$ one-to-many queries, or alternatively $5,964 \times 49,573$ one-to-one queries. While HY is optimized for the former access pattern, DO can only perform the latter access pattern. Regardless of how the distances are computed, it takes 295.6 million distance computations on the road network to compute this distance matrix. The following SQL statement computes the distance matrix for DO.

```
SELECT x.id, y.id,
       dist(x.lat, x.lon, y.lat, y.lon) as dist
  FROM University x, Restaurant y
```

Table 3.2 shows the performance of the DO and HY architectures. DO com-

putes the distance matrix in 8853.9 seconds, while HY does it in 20139 seconds. The throughput for DO is 33.3k distances/second while it is 14.6k distances/second for HY. Note that this is in spite of choosing a query workload that is favorable to HY. This shows that for a practical real query, DO is still 2.4× better than HY in terms of throughput.

The next question is how would HY perform if restricted to only use the one-to-one access pattern. To provide this comparison, we randomly pick $10,000$ source university and destination restaurant pairs from the tables. While DO takes 0.327 seconds to compute the distances, HY takes 2026.4 seconds which amounts to just less than 5 distances/second, while DO can computer over $30,000$ distances/second. This shows that HY is only appropriate if the query results can be obtained using a one-to-many access pattern as its performance for a one-to-one access pattern is prohibitively slow.

3.5.4 KNN Query

Next, we compare the performance of DO and HY for KNN queries where the inputs are two datasets S and R, and the goal is to find the K nearest neighbors of each point in S from points drawn from R. The workload for this subsection includes performing $5,964$ KNN queries for each of the universities returning K nearest restaurants.

The HY architecture invokes SCAN_UNTIL_K(), which uses an in-memory graph representation and stores the $49,573$ restaurants relation in a k-d tree data

structure. During processing, HY uses 7 threads to scan the road network. Each thread starts scanning from one of the university locations and for each vertex it performs a lookup on the k-d tree to determine if there are any restaurants in its vicinity within a certain distance range. If yes, then they are enqueued with the appropriate network distance if they have not been visited before. This check is not necessary if while building the k-d tree, each restaurant is associated with its nearest vertex. This process is entirely in-memory and extremely efficient to perform.

The DO architecture computes DIST() between each university and each restaurant in a candidate set of restaurants that have the potential to be the K nearest neighbors. This candidate set is obtained by first using the GiST spatial index in Postgres to compute the K Euclidean nearest restaurants from the restaurant relation for each university and then using DIST() to compute their corresponding network distances. Let d be the maximum of these network distances for the university being processed. Next, again use GiST to compute all nearest restaurants for each university whose Euclidean distance is less than or equal to d and then use DIST() to compute their corresponding network distances and retain the K closest ones.

This method is correct because the Euclidean distance is a lower-bound on the network distance. The lower bound property guarantees that we find the K network neighbors within the candidate set. The following SQL query captures the steps indicated above. In the subquery *kdn*, we compute the Euclidean distance to the K neighbors using the GiST index and then compute the maximum network distance among K neighbors for each university.

```
SELECT kdn.id as id1, R.id as id2,
       dist(kdn.lat, kdn.lon, R.lat, R.lon)
FROM (
  SELECT y.id as id, y.lat as lat, y.lon as lon,
    ( SELECT max(dist)
      FROM (
         SELECT dist(x.lat, x.lon, y.lat, y.lon)
         FROM restaurant x
         WHERE x.gid != y.gid
         ORDER BY x.geom<->st_setsrid(y.geom, 4326)
         LIMIT K
      ) as foo
    ) / 111000 as deg
  FROM university y
) AS kdn, restautant R
WHERE R.lat between kdn.lat - kdn.deg
                and kdn.lat + kdn.deg AND
      R.lon between kdn.lon - kdn.deg
                and kdn.lon + kdn.deg;
```

Figure 3.5 shows the execution time of the KNN queries for different values of K. We see that HY has nearly identical performance compared to DO for smaller values of K less than 500. It becomes $2-3$ times worse for larger values of K such as for $K = 49,573$. At $K = 49,573$, the query degenerates to computing the distance

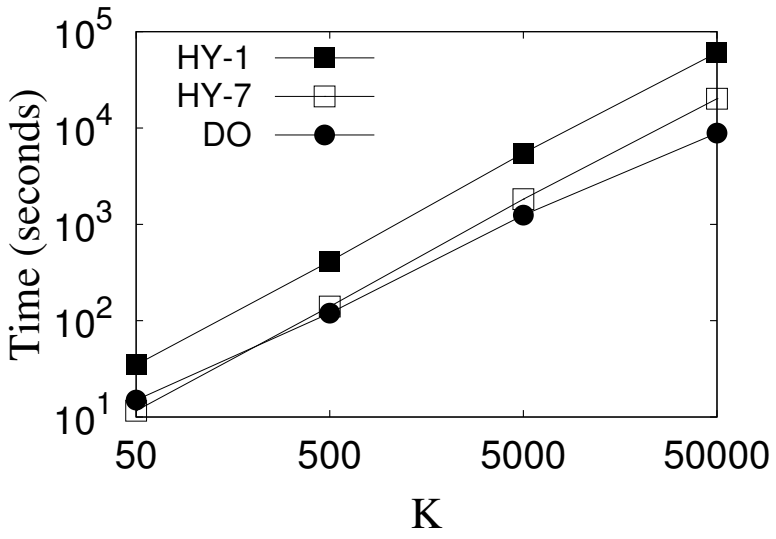
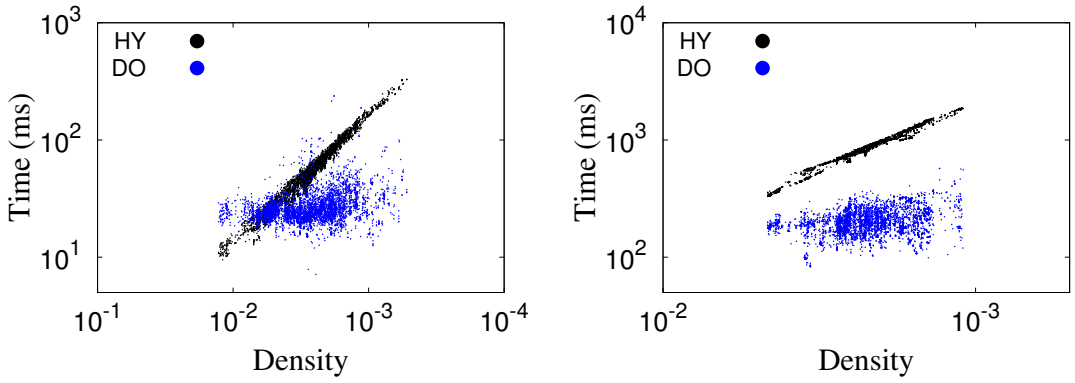


Figure 3.5: The execution time of 5,964 KNN queries where K = 50, 500, 5000, and 49573.

matrix between the source and destination tables.

Figure 3.6 illustrates the effect of the density on the execution time of DO and HY for the KNN queries. In particular, each point in the figure corresponds to the performance of one KNN query for either HY or DO. Here we use a real world dataset and thus for the values of K that we used, the density of most scans is less than 0.01. Compared to the region query discussed in Section 3.5.2, the effect of varying the density has a different effect on the execution time of DO and HY. In particular, for the KNN query, the execution time of HY increases significantly as the density decreases, while the execution time of DO does not change much. This is because the number of DIST() invocations for DO is proportional to K in real world datasets. On the other hand, SCAN_UNTIL_K() for HY needs to scan further to visit at least K destinations when the density of the nearby destinations is sparser.



(a) (b)

Figure 3.6: The execution time of the KNN query as a function of the density for (a) K = 500 and (b) K = 5000.

3.5.5 Trajectory Query

We now examine a simple trajectory query implemented on both the HY and DO architectures. The goal of the query is to compare the time performance on a real trajectory dataset consisting of GPS readings. GPS devices are becoming commonplace and are now deployed on many different commercial and non-commercial vehicles. A company operating a fleet of taxis usually has a GPS installed in all of its vehicles, which enables an operator to know the locations of its vehicles. For instance, when a customer requests a ride, the taxi operator uses the current locations of all of its vehicles to send the nearest vehicle to the customer. Of course, there are more complex analyses that an operator may want to perform from historical (i.e., a day/week/month/year) worth of GPS information collected from vehicles which can shed light on several aspects of their businesses.

For example, consider a query that seeks the execution time of computing the

total trip distance of each taxi. More SQL queries can be found in Section 3.6. Executing it using DO involves a few simple steps as detailed below.

1. Extract all points of a given trajectory denoted by tripid
2. Create an ordering of the points
3. Compute road network distance between consecutive points
4. sum the distances to produce the trajectory distance
5. sum the trajectory distances to produce the taxi trip distance

On the other hand using HY to respond to this query involves sorting all GPS records according to the ts attribute in the initialization. Next, defining a *segment* as two consecutive GPS locations on the same trip, we compute the distance of each segment by assigning it to one scanning thread. Thus, one segment contains one source and one destination location. Intuitively, each segment query is better described as a one-to-one access pattern, so that DO should be better than HY in this case. However, since the GPS sensors report their locations frequently, the distance between two consecutive GPS reports is very short, which benefits HY. Prior experiments for one-to-many access pattern queries showed that HY is as good as DO for short scanning distances.

Figure 3.7 demonstrates that DO is much better for the trajectory query. Each point in Figure 3.7 corresponds to one taxi. The x-axis is the number of segments for each taxi. For each method, we first sorted the 537 taxi points by the number of segments, and then connected the 537 taxi points with a line. DO computed the travel distance of one taxi within 0.2 to 0.5 seconds when the number of segments is

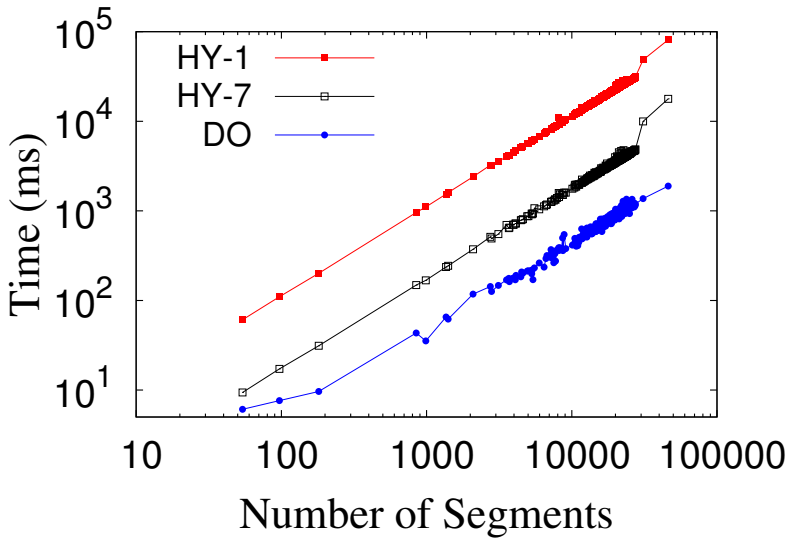


Figure 3.7: The execution time of computing the total travel distance of each one of 537 taxis.

around $10,000$. It is at least one order of magnitude faster than HY with 7 scanning threads, and two or more orders of magnitude faster than HY with just 1 scanning thread.

3.5.6 HY Performance Tuning: Number of Threads

The reason we use 7 scanning threads in our experiments is that the EC2 machine only has 8 cores. In our implementation, Dijkstra's algorithm in HY starts T threads for scanning. Figure 3.8 shows the execution time for the distance matrix query as the number of threads started by the main thread is varied. From the figure we observe that the execution time for 7 scanning threads is between $\frac{1}{4}$ and $\frac{1}{3}$ of the time when we have just one scanning thread. As we expected, in order to utilize all of the available computing power, the optimum number of threads is

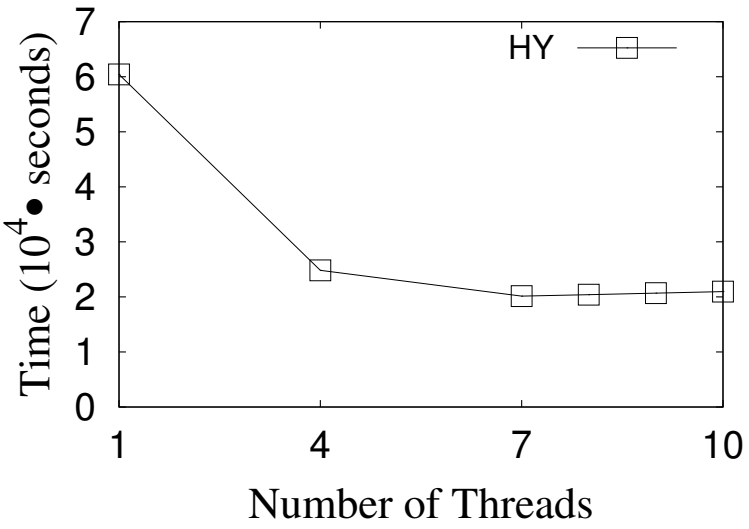


Figure 3.8: Execution time of a multi-thread Dijkstra's algorithm implementation for $5,964$ SCAN_UNTIL_K() with $K = 49,573$

equal to the number of cores minus one. This means that one core runs the main thread, and the remaining cores run the remaining threads, one thread per core.

3.6 Trajectory Solution Examples

Q1. List of all trips made by a taxi.

```
SELECT tripinfo(tripid) FROM taxi where taxiid = 1
GROUP BY tripid;
```

Q2. How many kilometers did each of the cars travel with passengers?

```
SELECT sum(foo.tripdist)/1000 as distance_in_km
FROM
   (SELECT (tripinfo(tripid)).tripdist as tripdist
```

```
    FROM taxi WHERE taxiid = 1 and occupancy = 1
    GROUP BY tripid) as foo;
```

Q3. Average distance they travel without passengers?

```
SELECT avg(tripdist) FROM
   (SELECT tripdist(tripid) as tripdist FROM
         (SELECT tripid from taxi
            WHERE occupancy = 0
            GROUP BY tripid) as foo
   ) as foo1;
```

Q4. Did my driver take a huge detour?

To see if the driver has been taking detours, we need to first figure out how to compute the direct distance between the first and last points in the trajectory. We define the following function, tripdist_sd.

```
CREATE FUNCTION tripdist_sd (which_trip_id bigint)
  RETURNS TABLE(
     tripdist float
  ) IMMUTABLE AS
$$
BEGIN
   RETURN QUERY
     SELECT dist(t1.code, t2.code) FROM
        (SELECT * FROM trip(which_trip_id) as t
         order by (t).id LIMIT 1) as t1,
```

```
        (SELECT * FROM trip(which_trip_id) as t
         order by (t).id DESC LIMIT 1) as t2;
END;
$$ LANGUAGE plpgsql;
```

Now we can compute the difference between direct from source to dest vs. driver's routing. For instance, we can compute the detour for tripid = 500. In this case, the direct distance 2104.6 vs. the driver's route is 2325.1 meters. In other words, the driver drove 200 more meters . This begs the question, are there some drivers take many circuitous routes. In particular, we are not interested in 200 meter detours but rather routes that are more than 5 kms longer than the direct path from source to destination. We use the following query to obtain the top 100 routes with the maximum detour. We limit the trips to those that are at most 50 kms.

```
SELECT taxiid, tripid, tpsd, tp, (tp - tpsd) as diff
  FROM
     (SELECT tripdist(tripid) as tp,
             tripdist_sd(tripid) as tpsd,
             taxiid, tripid
        FROM taxi
        WHERE occupancy = 1
      GROUP BY taxiid, tripid
     ) as foo
  WHERE tp is not null AND tpsd is not null
        AND tp < 50 * 1000 AND tpsd < 50 * 1000
```

```
ORDER BY diff DESC
LIMIT 100;
```

We immediately realize by looking at the output of this query is that drivers make many loops. In particular, the taxi makes a huge loop and comes back to the same point, just like airport shuttles. So which are the taxis that detour too much? Also what is the average length of a detour?

```
SELECT taxiid, count(*) as num_trips,
       avg(tp-tpsd) as average_detour
  FROM
    (SELECT tripdist(tripid) as tp,
            tripdist_srcdst(tripid) as tpsd,
            taxiid, tripid
       FROM taxi
       WHERE occupancy = 1
       GROUP BY taxiid, tripid
     ) as foo
  WHERE tp is not null AND tpsd is not null
        AND tp < 50 *1000 AND tpsd < 50 * 1000
        AND tp >= tpsd
GROUP BY taxiid
ORDER BY average_detour desc;
```

The result shows that Taxi 518 had 1082 trips and had a average detour of 14 Kms. After further examination, we know that this is the taxi with a buggy GPS. Besides

Taxi 518, Taxi 19 made 594 trips 2.978 kms detour, which is roughly 1800 kms more than direct routes. Maybe this driver has many carpool rides.

Q5. Where is the nearest taxi?

Finding a taxi involves finding a taxi in terms of both distance and time. In a live system the time is the current epoch timestamp. The following query finds the nearest taxi to the hard coded location (37.75, -122.4) for the customer and time when the customer requested (1211840888). This query extracts taxi's position with in a 10 minute time window.

```
SELECT * from (
    SELECT taxiid,
           last(occupancy) as taxi_occupancy,
           last(lat) as taxi_lat,
           last(lon) as taxi_lon,
           last(code) as taxi_code, dist(last(code),
           Z2(37.75, -122.4)) as taxi_distance
    FROM (
         SELECT * FROM taxi
           WHERE ts BETWEEN 1211840888 - 10 * 60
                 AND 1211840888
         ORDER BY TS
        ) as foo
    GROUP BY taxiid
 ) as foo1
```

```
WHERE foo1.taxi_occupancy = 0

ORDER BY taxi_distance

LIMIT 10;
```

3.7 Related Work

It is well-known that Dijkstra's algorithm [37] is very efficient for single source queries such as finding the nearest K restaurants to a given location. However, for an s-t query, Dijkstra's algorithm has to scan many irrelevant vertices to reach the given target vertex. A number of techniques have been proposed to overcome the drawbacks of Dijkstra's algorithm for s-t queries on road networks. They fall into two main categories: memory-based methods and database-centric methods.

Memory-based methods: Most of the state-of-the-art approaches are memory-based, They can be subdivided into two groups. The first group are *graph-based*, which are based on the observation that some vertices in a spatial network are more important for shortest path queries, while offering different trade-offs between pre-processing time, storage usage, and query time. Goldberg et al. [40] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. The Construction Hierarchy (CH) [39] assigns an importance score to each node and replaces some original edges by shortcuts. [21, 30, 43, 55] precompute the shortest distances between landmarks or hub nodes and other vertices, and then answer the shortest distance queries by assuming that the shortest path passes through one landmark or hub node. [22, 25, 34, 61, 76] build an explicit hierarchy graph to over-

come the drawback of Dijkstra's algorithm. The second group are *spatial based methods*, which overcomes the drawback of Dijkstra's algorithm by using geometric techniques. RNE [69] applies a Lipschitz embedding [42] to a spatial network so that vertices of the spatial network become points in a high-dimensional vector space. [60, 62, 72] use the fact that the set of shortest paths from vertex u to all other vertices can be decomposed into subsets based on the first edges on the shortest paths from u to them. SILC [60, 62, 63] stores these subsets in a variant of a region quadtree where all vertices stored in a quadtree block are in the same subset.

Database Centric methods. On the other hand, approaches rooted in databases mainly focus on database-centric methods. [64, 65, 67] exploit the spatial coherence so that if two clusters of vertices are sufficiently far away, then distances between pairs of points in different clusters are similar. PCPD [67] gives one exact shortest path algorithm, while the ϵ-distance oracle [64, 65] propose an approximate shortest distance algorithm, which balances the tradeoffs between accuracy and storage. HLDB [20] is a recent practical database-centric method that is based on hub labels (HL) [21], which is a popular memory-based method. HLDB [20] claims that most of the memory-based approaches surveyed in [35] are difficult to embed into a database system and to use with SQL queries since they rely on complicated data structures such as graphs and priority queues. One of the main contributions of HLDB is embedding the memory-based HL method into a database.

On the other hand, there are also a few approaches that focus on speeding up specific spatial analytic queries. Some are based on the techniques that speed up the s-t queries. Knopp et al. [41] explain how to use highway hierarchies [61] for

computing many-to-many shortest distances. Shahabi et al. [69] and Samet et al. [60] show how to speed up the K nearest neighbor search by using different source-target techniques. Delling et al. [36] utilize partition-based algorithms developed for s-t queries to handle POI queries. Cho et al. [31] propose UNICONS for continuous nearest neighbor queries, and then propose ALPS [32] for top-k spatial preference search.

3.8 Summary

The HY architecture represents a *procedural* way of performing spatial analytic queries since the analysis tool is the "glue" that coordinates computations between the database and the road network module. The DO architecture represents another end of the spectrum where the spatial analytic query is expressed in a *declarative* manner. The declarative nature of queries means that the user expressed what the query should do and the database automatically figures out how the query should be executed. On the other hand, HY by being procedural represents a custom development effort where most of the responsibility for optimization lies with the analysis tool. This can be viewed as a drawback of the architecture since an optimization opportunity may be lost in the processing of the spatial analytic queries that is outside the database system.

From the perspective of ease of use, DO is better than HY since users can easily express complex queries using SQL. There is no need for much of a learning curve since we only extended SQL by one function (i.e., DIST()). DO can be implemented

on any database system as it requires no modification to the database system which means that now road networks can be incorporated with any legacy database that is already hosting spatial datasets.

DO is far superior to HY when it comes to a one-to-one access pattern, which is commonplace in trajectory queries, where GPS crumbs are recorded periodically and the road distance between them needs to be computed by applying a one-to-one access pattern. On the other hand, HY is better than DO for some one-to-many access patterns such as the region query in the case of a high density of destinations vis-a-vis the visited vertices of the graph, as well as when the maximum scanned distance is not large.

Our synthetic experiment for the region query showed that once the density became less than 1 object in 100 vertices, DO is a better choice. To put this in perspective, if there are more than 240k objects (e.g., restaurants) in a dataset on the USA road network (recall it consisted of 24 million vertices), then HY could be slightly faster than DO. However, if the query applied a predicate on the objects (e.g., only Indian Restaurants) then the density may be far lower again thereby rendering DO more suitable.

Chapter 4: SPDO: High-Throughput Road Distance Computations on Spark Using Distance Oracles

4.1 Overview

Some spatial analytic queries that use distance along a road network require performing millions of shortest distance computations [66]. As an example, consider the heat map in Figure 4.1 that shows the average commute distance in kilometers for residents of California. This query is of immense interest of transportation planners and was computed using the LEHD data [10] from the US Census Bureau by performing $13,645,807$ road network distance computations. Such analytic queries that compute millions of network distances are commonplace in logistics, route planning, and spatial business intelligence. Existing solutions usually use the geodesic distance (Euclidean distance) instead of the network distance, which makes their results inaccurate [52]. For instance, a delivery company that delivers 1000 packages would compute a distance matrix that captures the distance between every pair of destination locations to plan the routes. Using geodesic distance is easy but only the shortest distance on the road is optimal. We need a framework to perform tens of millions of distance computations on road networks quickly to cater

to the requirements of delivery companies such as AmazonFresh, Google Express, UberRush etc., that seek to respond quickly to the dynamic supply-demand arising in their business.

A distributed framework is used to achieve a high throughput. It relies on the simplicity of the underlying computation task but as this task becomes more complex, the greater the difficulty in embedding it in a distributed framework. In particular, most methods that focus on reducing the latency time rely on complicated data structures such as graphs and priority queues. These methods don't take into account the difficulty of using them in a distributed framework. A compromise solution for a spatial analytic query is to partition the query workload across multiple, say M, machines as shown in Figure 4.2(a), which is a common way of approaching this problem. In this case, each task machine's memory is preloaded with the same datasets and graph. The gateway machine is responsible for partitioning the query workload, assigning the sub-queries to the individual task machine, and collecting the results from the task machines. The drawbacks of this solution are:

1. Any update for the datasets or graph needs to be processed M times.
2. Since the distribution of the workload is not automatic, developers who use this solution need to manually maintain the connection between the gateway machine and each task machine, detect the failures during computing, and re-execute the sub-queries if any failure occurs.
3. If the size of the graph and datasets are greater than the amount of available memory, then users need to increase the amount of memory in each of the

M machines, or place the graph and datasets on disk which greatly sacrifices processing time.

Thus, we need a more general distributed framework to handle spatial analytic queries.

In this chapter, we develop a distributed framework called SPDO (pronounced *speedo* denoting *Spark and Distance Oracles*) using Apache Spark [75] which is optimized for high-throughput network distance computations (see [71] for another approach). We extend our work on ASDO which precomputes and stores a compressed version of shortest distances between all pairs of vertices in a road network within an error tolerance ϵ. The resulting representation takes $O(\frac{n}{\epsilon^2})$ space, where n is the number of vertices in the road network and ϵ is an approximation error bound on the result. Previously, [52] showed how to map the distance oracle representation to an RDBMS system and how to solve complex analytic queries on a road network. Here we show how to map distance oracles to a distributed key-value store (i.e., hash abstraction) which we choose to be Spark. Combining Spark and distance oracles is a good match. In essence, Spark provides a highly scalable fault-tolerant distributed framework with the ability to cache large datasets in memory using RDD [75], while distance oracles provide a compact representation of network distances that requires very little computation at run time. Furthermore, Spark is a popular open-source distributed framework for general purposes, which is more than a key-value store. We can easily develop functions in Spark combining distance oracles and other techniques that are not efficient in a key-value store. In particular, we use the *IndexedRDD* library on Spark which is a memory resident,

key-value store. The high-throughput of our proposed framework is achieved due to the ability to spread query processing across multi-machines in a Spark cluster as well as the in-memory representation of distance oracles.

Mapping distance oracles to a distributed key-value store is challenging as it requires converting any source-target distance query (s-t query for short) into a small number of lookups into the distributed key-value store. It also requires being able to partition the work between the master and task machines in Spark. Finally, the network communication can be a significant bottleneck if the access to the distributed key-value store storing the distance oracles is not well designed.

The main contributions of this chapter are: 1) A high-throughput architecture using distance oracles and Spark for a large set of spatial analytic queries; 2) Three variants of distributed key-value algorithms for our architecture; 3) An analysis of the time and space complexity of our methods, and a detailed comparison with state-of-the-art methods for realistic datasets and applications. We released the SPDO codebase and associated precomputed distance oracles in GitHub [1]. SPDO needs just a few lines of code in order to be incorporated with an existing Spark project that needs to compute large number of network distances. The use-case shown in Figure 4.1 makes 13.6 million distance computations in 13 seconds on 5 machines, which roughly works out to more than $200K$ distance computations/sec per machine. In contrast, one of the fastest latency methods took more than 20 minutes for the same query on 5 machines as well, which is at least two orders of magnitude slower than our approach.

[1] `https://github.com/shangfu/SPDO.git`

The rest of this chapter is organized as follows. Section 4.2 explains the theoretical grounds for mapping work about distance oracles to a hash structure, and Section 4.3 presents three variants of our distributed key-value algorithms, denoted as Basic, BS, and WP, respectively. Section 4.4 describes a detailed experimental evaluation of our methods, and also provides two real applications using our methods. Section 4.5 reviews the related work. Concluding remarks are drawn in Section 4.6.

4.2 Hash Access for Distance Oracles

Table 4.1: Notation Summary of SPDO

Symbol	Meaning
n	the number of vertices in the graph
N	the number of s-t queries
ϵ	the error bound of the ASDO representation
$mc()$	Morton code function
D	the maximum depth of the DO-tree
M	the number of task machines

Table 4.1 summarizes the notation that we use in this chapter. Recalling Figure 2.2(a) in Chapter 2, given a spatial domain S, the Morton order of blocks in S can be obtained by subdividing the space into $2^D \times 2^D$ equal sized blocks

called *unit blocks*, where D is a positive integer called the maximal decomposition depth. Figure 2.2(a) shows Morton codes in the same domain when D is equal to 0, 1, and 2, respectively. Each unit block i is referenced by a unique Morton code $mc(i)$. There are two ways to represent Morton codes, numeric representation and string representation. As in Figure 2.2(a), the number representation cannot distinguish the number 0 at depth 1 from the number 0 at depth 2. So the completed numeric representation should also contain the corresponding depth information. For instance, $(0, depth\ 2)$ is equivalent to "0000", and $(0, depth\ 1)$ is equivalent to "00". Later, we use the string representation to explain ideas, while we use the numeric representation in practice since it is more efficient.

In the past several years, many key-value stores appear such as Berkeley DB [48], HBase [29], Redis [16], etc. In the distributed environment, a key-value store supports the hash access model well, which is like a HashMap data structure in Java, so that users can find a given key and its value in $O(1)$ time. Note that although some key-value stores also support sorting keys in order, the hash access model is better for parallel processing the workload.

Spark [75] is not a pure key-value store, but a general-purpose cluster computing framework. *IndexedRDD* [9] supports almost all the features of a key-value store for Spark. The cluster of Spark has one master machine and M task machines. In contrast to Hadoop's two-stage disk-based MapReduce framework, Spark's in-memory primitives provide performance up to 100 times faster for certain applications. Recalling that each well-separated pair is a key-value pair, our scheme for storing ASDO works seamlessly on any key-value store. Thus, we can load all well-

separated pairs in Spark's distributed memory. Even for the ASDO of the USA road network, several machines with 32 or 64GiB memory are enough. Embedding ASDO in Spark is the best solution so far for the spatial analytic queries, which have millions or billions of source-target pairs.

However, if the distributed key-value framework is a hash access model, we can neither build an ordered index like in RDBMS nor redefine the comparison operator as in [64]. Thus, given a batch of s-t pairs, how to efficiently find the unique well-separated pair for each s-t pair is the core problem in a distributed key-value framework.

In this section, we show how the distance oracle can be mapped to a hash structure which will be implemented on top of Spark using RDD. The ASDO of [52] stores the Morton codes in sorted order inside a RDBMS by using a B-tree index structure and redefines the comparator operator. Each source-target query performs a tree lookup in the B-tree which takes $O(\log n)$ I/O operations. This method is ideal for disk-based systems that store the distance oracles on disk pages but we want to develop the necessary theory in order to be able to map the distance oracle to a hash structure which is memory resident. This is in contrast with a B-tree which is typically good for disk-based access.

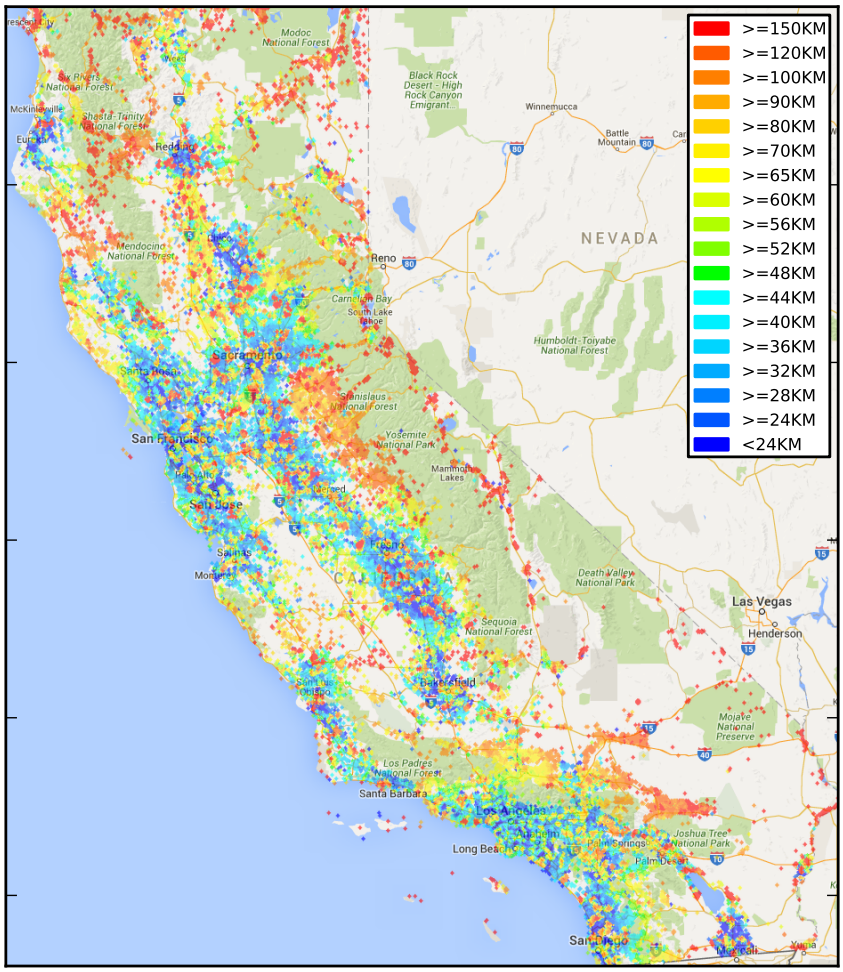


Figure 4.1: Geographical heat map for the average drive distance from living place to workplace for people in California: a pixel's color in this figure denotes the average drive distance of people residing in the pixel's region. The query workload is $13,645,807$ shortest distance computations, and our distributed key-value method on a Spark cluster with 5 task machines took 13 seconds. In contrast, state-of-the-art methods, e.g., CH [39], running on the same 5 machines in parallel, needed more than 20 minutes.

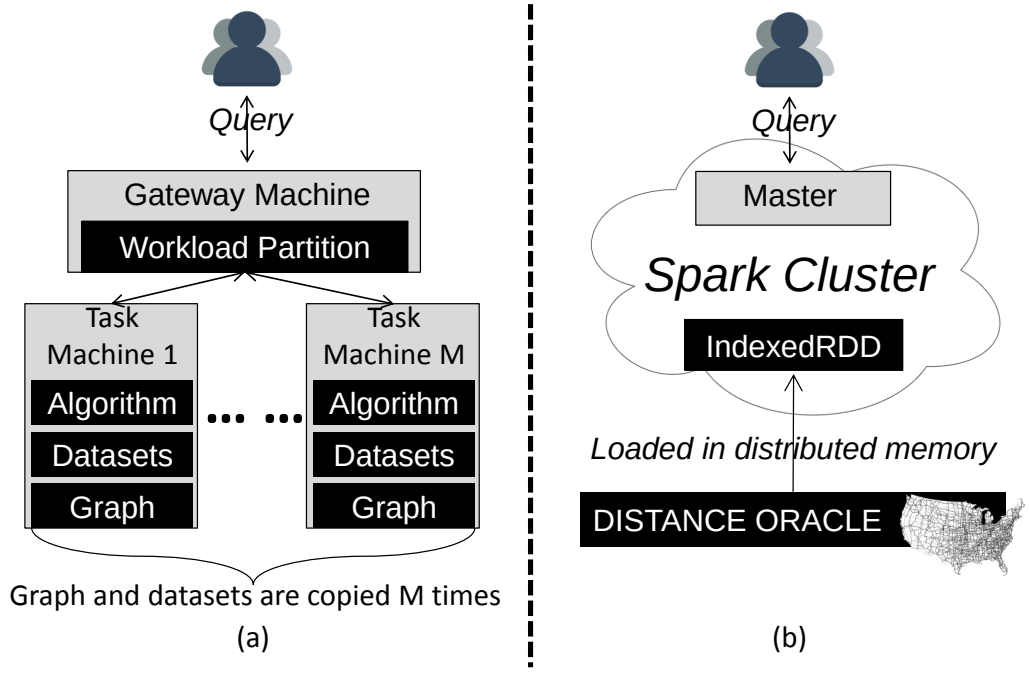


Figure 4.2: (a) The architecture of most traditional analysis tools, where all the task machines are the same; (b) The architecture of our key-value distributed methods using Spark. Note that any distributed framework that supports key-value operations can be substituted for the Spark cluster.

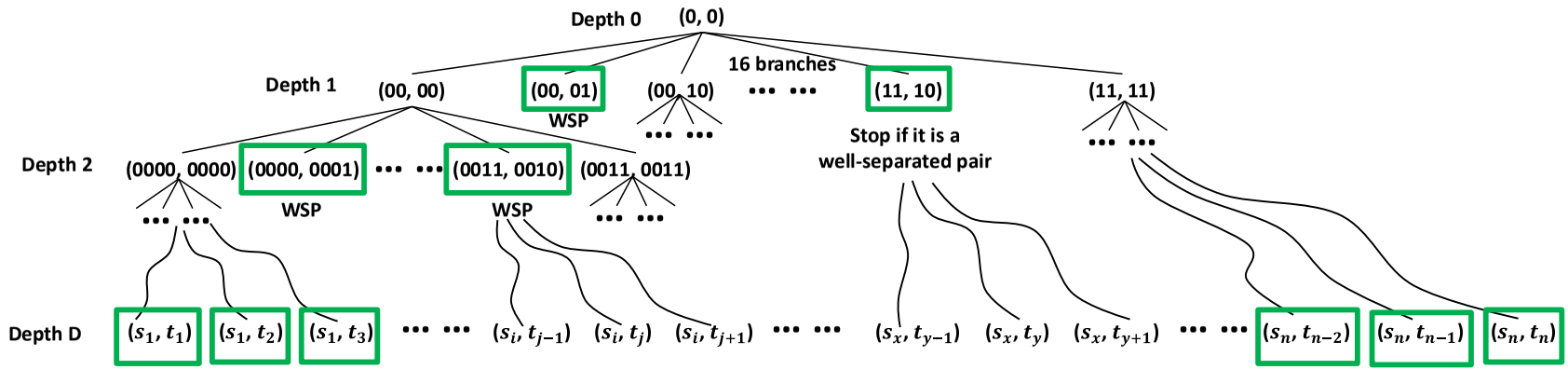


Figure 4.3: Example of the DO-tree which represents the distance oracles pre-computation: each node is a 4-dimensional Morton code denoting a pair of 2-dimensional Morton codes. Each node is decomposed into 16 nodes unless it corresponds to a WSP, i.e., the nodes inside a green rectangle, in which case no decomposition takes place. The nodes at the maximum depth D are pairs of leaf quadtree blocks that contain a single vertex and they are trivially a WSP.

The construction of a distance oracle creates a tree structure, referred to as the *DO-tree* such that its leaves form the block pairs which make up the distance oracle. Figure 4.3 shows the *DO-tree*, which has several properties that we develop to build a theoretical foundations for our hash data structure. Recall that the distance oracle is constructed by taking a PR-quadtree on the spatial positions of the vertices. We start with a block pair formed by the root of the PR-quadtree. This block pair forms the root block of the DO-tree. At each step of the distance oracle construction, we test to see if a block pair forms a WSP. We do this by checking the ratio of the network distance between two representative vertices, one drawn from each of the block pairs, to the *network radius* of the blocks. If the block pairs form a WSP by virtue of the ratio being greater than $\frac{2}{\epsilon}$, then we halt further decomposition. This block pair forms a leaf block of the DO-tree. If the block pair is not a WSP, then we decompose the block pair into 16 children block pairs and continue to test them for the satisfaction of the WSP condition. The block pairs that do not form a WSP correspond to the non-leaf blocks in the DO-tree. Due to the nature of how the DO-tree is constructed, each non-leaf node of the DO-tree has 16 children nodes. Furthermore, the maximum depth D of a leaf node in the DO-tree is the same as the input PR-quadtree. A block pair at depth D in the DO-tree corresponds to leaf blocks in the PR-quadtree, each containing a single vertex. These block pairs trivially form a WSP since we record the exact network distances for these cases. It can also be noted that not all the leaf blocks in the DO-tree are at depth D.

We can define the uniqueness property of the DO-tree which serves as the basis of being able to find a block pair from a hash structure that we will define

later. Uniqueness means that given any pair of vertices denoting a source s and a destination t, there exists exactly one leaf node in the DO-tree that contains the source and destination vertices. This property is due to the original property of the distance oracle that there is exactly one WSP containing any source and destination pair as well as the mapping of WSP to leaf blocks in the DO-tree. We state this as a property below.

Property 4.2.1. *Given a source-target query (s, t), there is exactly one leaf node that contains both s and t, although note the subtle distinction that there may be several non-leaf nodes in the DO-tree (e.g., the root of the DO-tree) that contain s and t. This leaf node in the DO-tree is the only node that can provide the ϵ-approximate network distance between s and t.*

From Property 4.2.1, we know that there exists exactly one leaf block that contains the source and the destination. Finding it requires generating all possible leaf nodes that can possibly contain the source and destination starting with the smallest possible leaf node.

Lemma 4.1. *A hash table $\mathbb{H}_1$ of size $O(n/\epsilon^2)$ can be constructed that enables the retrieval of the network distance between any pair of vertices in $O(D)$ lookups.*

Proof. The hash table $\mathbb{H}_1$ is constructed using only the leaf nodes of the DO-tree. Since the leaf nodes correspond to different blocks in the PR-quadtree, they form a unique four-dimensional Morton code. The hash table uses the four-dimensional Morton codes as the key and the approximate network distance as the value. A simple way to find the desired leaf node using such a hash table is to perform

$(D+1)$ lookups. Given a source s and destination t, we start out by forming a four-dimensional Morton code $mc(mc(s), mc(t))$ at depth D containing both s and t. We test to see if $\mathbb{H}_1$ contains this key, and if so, then we can obtain the approximate network distance of s and t. If $\mathbb{H}_1$ does not contain the key, then we check to see if $\mathbb{H}_1$ contains the parent of the block pair. We can obtain the parent by performing a bit-shift operation in $O(1)$ time. For example, if the initial four-dimensional Morton code is `001100101100`, then the parent block is obtained by left bit shifting 4 times to obtain `00110010`. We are guaranteed that the search process will find a key within $D+1$ lookups by virtue of the satisfaction of Property 4.2.1. □

The advantage of looking up values in $\mathbb{H}_1$ is that the $O(D)$ lookups can be performed concurrently as opposed to performing them sequentially. The reason is that exactly one of the D keys that can be generated from a given source and destination vertices will be found in the hash table, as we do not store the non-leaf nodes of the DO-tree in $\mathbb{H}_1$. This property can be useful in designing a lookup function for querying $\mathbb{H}_1$. Although querying $\mathbb{H}_1$ $D+1$ times in parallel may result in a lesser response time, querying $\mathbb{H}_1$ in sequence can result in higher throughput as it takes far fewer lookups.

We can further improve the performance of the hash structure by storing both the leaf and the non-leaf nodes in the hash table, which dramatically reduces the number of lookups needed. In order to do this, we first show that the number of non-leaf nodes in the DO-tree is also $O(\frac{n}{\epsilon^2})$. From the nature of distance oracle construction, we know that the total number of leaf nodes is $O(\frac{n}{\epsilon^2})$ since each leaf

node corresponds to exactly one WSP. To compute the number of non-leaf nodes, we use a similar approach to that taken in [28, 64]. One internal node that is not a WSP produces 16 nodes. Since the number of WSPs is $N_{wsp} = O(\frac{n}{\epsilon^2})$, the number of examined nodes is: $N_{tot} = N_{wsp} + \frac{1}{16}N_{wsp} + \frac{1}{16^2}N_{wsp} + ... = \frac{16}{15}N_{wsp}$. Another way of showing this is pointing out that the DO-tree is a tree with out-degree of 16. The number of non-leaf nodes for any such tree is the same order of magnitude as the number of leaves. Hence, the total number of nodes in the DO-tree is also $O(\frac{n}{\epsilon^2})$.

Lemma 4.2. *A hash table $\mathbb{H}_2$ of size $O(n/\epsilon^2)$ can be constructed that can retrieve the network distance between any pair of vertices in $O(\log D)$ lookups.*

Proof. The hash table $\mathbb{H}_2$ stores both the leaf and non-leaf nodes of the DO-tree. Our goal is to find the leaf node containing a source and a destination but to use non-leaf nodes in order to guide the search process. We find the leaf node by performing a binary search on the depths of the DO-tree. Given a source s and a destination t, we generate a four-dimensional Morton code of s and t at depth $D/2$. If the hash table contains the key, then one of two options is possible. In particular, the key corresponds to a non-leaf node in the DO-tree or it could be a leaf node but we are not sure which is the case unless we make sure that no other node exists at a deeper depth. To ensure this, we generate another Morton code at a depth between $(D/2, D]$. We continue doing this till we find a case where a node exists but we cannot find any children block in $\mathbb{H}_2$. Since this process is a binary search on the depths of the DO-tree, the number of lookups is $O(\log D)$. □

It is important to note that in contrast to $\mathbb{H}_1$ which could support concurrent

lookups, the hash table $\mathbb{H}_2$ can only perform sequential lookups. The reason for this is that finding or not finding nodes in the hash table indicates how the search would proceed in the next step. However, $\mathbb{H}_2$ can result in far fewer lookups compared to $\mathbb{H}_1$ since the number of lookups has been reduced to $O(\log D)$ from $O(D)$. Note that in almost all cases D is bounded by $O(\log n)$ [64], which means that $\mathbb{H}_1$ provides $O(\log n)$ access while $\mathbb{H}_2$ provides $O(\log \log n)$ access to the distance oracle.

4.3 Implementation in Spark

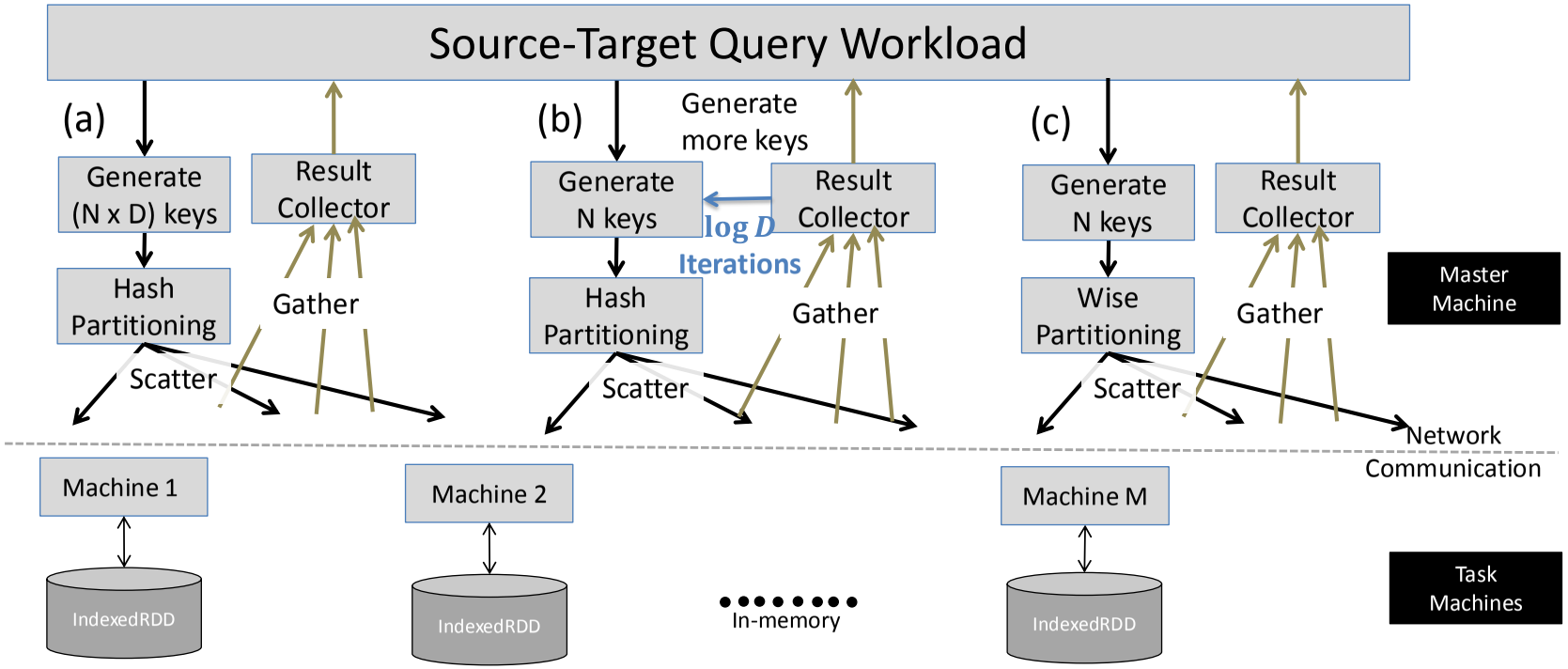


Figure 4.4: The three implementations of our methods, namely (a) Basic, (b) Binary Search and (c) Wise Partitioning methods, on top of Apache Spark

We first describe the setup of the Spark processing framework at a conceptual level before describing the different ways in which we implemented the distance oracles. The Spark computing cluster consists of a single master machine and M task machines. Our goal in this chapter is to evaluate a large number of network distance queries which are posed as a large set containing N source-target pairs. This workload can be generated by an analytic query such as in Figure 4.1 but for the sake of exposition, for our setup here the workload is available as a CSV file of source and destination locations stored in HDFS. The distance oracle for a large road network has also been precomputed and is stored on the HDFS. Associated with each task machine is an in-memory high performance key-value store abstraction called IndexedRDD [9], which caches part of the distance oracle in its memory. The keys in our case are the four dimensional Morton codes corresponding to the node in the DO-tree and the value is the corresponding approximate network distance. Spark uses an arbitrary *hash partitioning* method to distribute the nodes of the DO-tree uniformly across all M task machines. IndexedRDD is implemented by hash-partitioning the entries by key and maintaining a radix tree index called *PART* within each partition. It has been shown that PART [12] achieves good throughput and space efficiency that is on par with a mutable hash table. Hence, for all practical purposes any lookup operation into the IndexedRDD can be considered as taking $O(1)$ time.

A Spark program consists of a master and a task programs. Algorithm 2 provides an abstraction of the master program, while the workload in the task program of each task machine is the key search in its corresponding IndexedRDD,

Algorithm 2: MASTER Program in Spark

1 $DO \leftarrow$ load distance oracle from HDFS as an RDD and specify partitioner;

2 $hash \leftarrow$ IndexedRDD(DO).cache();

3 $Q \leftarrow$ List of source-target pairs;

4 $result \leftarrow$ GETDIST ($hash$, Q);

where the keys are assigned by the master machine. In the remaining section, we discuss three variants of Algorithm 2 that only differ by the variant of GETDIST() that they use.

4.3.1 Basic Method

Algorithm 3: GETDIST($hash$, Q) for Basic

Data: $hash$: IndexedRDD; Q: Batch of s-t queries.

Result: $Result$: Network distances for each s-t in Q

1 $codes \leftarrow$ compute D Morton codes for each pair in Q;

2 $result \leftarrow hash.multiget(codes)$; `/* runs in task program */`

3 **return** $result$;

The simplest way to implement a distributed hash table $\mathbb{H}_1$ is to expand each of the N source-target pairs into their D four-dimensional Morton keys. This relies on the concurrent aspect of $\mathbb{H}_1$ which ensures that all the D accesses can be made concurrently but only one of the keys will find a key in the hash table. The master machine reads N source-target pairs from HDFS, forms $(N \times D)$ keys, and assigns the keys to M task machines through a hash partitioning method. Note that the

hash partitioning method is the same as the one used to distribute the nodes of the DO-tree uniformly across all M task machines. Next, each task program checks if the assigned sets exist in its local hash map (i.e., IndexedRDD). Next, it reports the keys that it found along with their corresponding values (i.e., approximate network distances) to the master. Finally, the master collects the results from the M task machines, and returns to the user. There is really no need to check if the master obtained two distance values for a source-distance pair or if it missed finding one since Property 4.2.1 of $\mathbb{H}_1$ ensures that they cannot occur.

Figure 4.4(a) illustrates the flow plan of the Basic method in Spark with one master machine and M task machines. In particular, after the precomputation of ASDO, we have $O(\frac{n}{\epsilon^2})$ WSPs. In the set-up stage, we define an arbitrary *Hash Partitioner* in Spark, denoted as *HP*, to randomly partition the WSPs into M task machines. Each task machine loads the corresponding WSP set into its memory, and then builds a local HashMap for the WSP set using IndexedRDD. In the query stage, when the master machine receives N source-target pairs, it forms $(N \times D)$ keys and scatters the keys through HP.

4.3.2 Binary Search Method

The binary search (BS) method is an implementation of $\mathbb{H}_2$ which can retrieve a shortest network distance using $O(logD)$ operations as shown in Algorithm 4 and Figure 4.4(b). The task program in BS is exactly the same as it is in the Basic method, except that the HashMap (i,e. the IndexedRDD) contains both the leaf

Algorithm 4: GetDist($hash$, Q) in Master for BS

Data: $hash$: IndexedRDD; Q: Batch of s-t queries; D: the max depth of DO-tree

Result: $Result$: Network distances for Q

```
1  for i ← 0 to Q.length do
2      minD[i] ← 0;
3      minD[i] ← D;
4      code[i] ← compute Morton Code at depth D;
5  for j ← 0 to log D do
6      result ← hash.multiget(code);
7      for i ← 0 to Q.length do
8          Update minD[i] or maxD[i] based on result[i]; /* Binary Search */
9          code[i] ← compute Morton Code at depth (minD[i]+maxD[i])/2;
10 return result;
```

and non-lead nodes in the OD-tree. When the master program receives the N source-target pairs as inputs it first generates the Morton codes corresponding to depth D. These N Morton codes are provided to the M task programs by the hash partitioning method that checks for their existence. If a key is found in the hash table, then the search process is performed since any node found in the hash table at depth D in the DO-tree is a leaf node. The value of the key found is the approximate network distance.

If a key is not found at depth D, then a Morton code corresponding to depth $\frac{D}{2}$ is generated for the source-target pair. If the task program finds the key in the hash table, then it returns the success of finding and the value of the key. The master program in turn issues a new query with a key corresponding to Morton code at depth $\frac{3D}{4}$. In general, the new depth is the middle value of the depths that we tried in the prior two iterations such that one search resulted in success and the other in failure. The search continues until finding a depth d present in the hash table and a depth $d+1$ not present. This process can continue $\log D$ times as we are performing a binary search on the D depths of the OD-tree.

4.3.3 Wise Partitioning Method

Both the Basic and the BS methods have a common problem which is that the workload of the master machine is much higher than that of the task machines. In the BS method, each task machine receives $\frac{N}{M}$ keys at each iteration but the master needs to collect N keys and issue more queries. As each task machine simply looks

Algorithm 5: GETDIST($hash$, Q) for WP

Data: $hash$: IndexedRDD; Q: Batch of s-t queries; d, D: min, max depths of DO-tree in $hash$

Result: $Result$: Network distances for Q

1 $code \leftarrow$ compute the Morton code for each s-t pair at depth D;

2 $result \leftarrow hash.logSearch(code, d, D)$; `/* Binary search happens at each task machine */`

3 **return** $result$;

up a local hash map, its computational workload is much smaller than that of the master machine. For the Basic method, the master machine needs to generate $D \cdot N$ keys and process N results, while each of the task machines simply processes $\frac{1}{M}$ of the workload.

To make the workload more balanced (i.e., to increase the workload of the task machines), we replace the default hash partitioner HP with a partitioning method that we developed which we term the *wise* partitioner (WP). The wise partitioner improves up on the BS method by moving the $\log D$ iterations into the tasks as shown in Figure 4.4(c). In particular, in the BS method, the default hash partitioner HP randomly (and uniformly) scatters the queries among the M tasks during the task setup stage. The HP function is meant to uniformly distribute the keys among the M task machines and in that sense it does not preserve any locality in the data. Because of this, considering one s-t pair, the D keys in the Basic method and the $\log D$ keys in the BS method would likely be present on different task machines.

This is also why the master machine takes on a heavy workload in the Basic and BS methods. Recall that the BS method must coordinate the search among multiple task machines, collect results from all task machines, and even generate new keys.

To move all of the $\log D$ iterations into the task machines, each task machine needs to ensure that all of the D keys for a given s-t query must be contained in its local HashMap or none of it should be present in the local hash map. The wise partitioner algorithm achieves the partitioning of $O(\frac{n}{\epsilon^2})$ WSPs into M task machines such that all of the D keys for each s-t query are hashed to the same task machine.

The WP takes advantage of the presence of the non-leaf nodes in the DO-tree which help find the leaf nodes corresponding to WSP nodes. WP is constructed as follows. First, truncate the DO-tree at a depth d so that we obtain a forest of subtrees. d is chosen so that there are no leaf nodes at a depth less than d. All the non-leaf nodes that are at a depth less than d are discarded. We require that the number of subtrees in the forest is greater than M and typically it is much greater than M because larger blocks at lower depths tend not to form a WSP with other larger blocks. If the value of d results in fewer subtrees than M, then we simply choose a larger value of d but subdivide those leaf blocks further until they reach a depth of d. Although choosing a value of d appears to be a trial and error process, the key idea here is to make sure that we decompose the DO-tree into at least M subtrees. We found this not to be a problem for any road network dataset that we used in our experiments.

Once the DO-tree has been decomposed into subtrees, we assign an entire subtree to the same task machine while the subtrees themselves are assigned using

HP. Each subtree is stored in a local hash map and the BS method now finds the leaf nodes and its ancestors in the same task machine. Task machines perform a binary search as before except that the depth range is $[d, D]$ instead of $[0, D]$. The task machine checks to see if there is a key for a source-target pair at depth D. If it is not found, then it checks at depth $\frac{d+D}{2}$ and so on, until $\log(D - d + 1)$ iterations have been performed. Now, communicate the distance value to the master machine.

4.3.4 Analysis of Methods

Table 4.2: Analysis on the three distributed work flows of SPDO

Design	Iterations	Master	Each Task Machine	Network Communication
Basic	1	Time: $O(N \cdot D)$ Space: $O(N \cdot D)$	Time: $O(\frac{N \cdot D}{M})$ Space: $O(\frac{N \cdot D}{M}) + O(\frac{n}{\epsilon^2 M})$	Send: $O(N \cdot D)$ Receive: $O(N)$
BS	$\log D$	Time: $O(N \cdot \log D)$ Space: $O(N)$	Time: $O(\frac{N \cdot \log D}{M})$ Space: $O(\frac{N}{M}) + O(\frac{n}{\epsilon^2 M})$	Send: $O(N \cdot \log D)$ Receive: $O(N \cdot \log D)$
WP	1	Time: $O(N)$ Space: $O(N)$	Random — Time: $O(\frac{N \cdot \log D}{M})$; Space: $O(\frac{N}{M}) + O(\frac{n}{\epsilon^2 M})$ Worse — Time: $O(N \cdot \log D)$; Space: $O(N) + O(\frac{n}{\epsilon^2 M})$	Send: $O(N)$ Receive: $O(N)$

Table 4.2 summarizes our time and space complexity analysis of the three variants of GetDist used in the master program. However, there are couple of important considerations we need to keep in mind before embarking on our analysis. First, in distributed memory environments, network communication is a significant bottleneck greatly exceeding the CPU and IO since most of the dataset is memory resident. Second, Spark retains data in memory across iterations so multiple iterations in BS are not much of a bottleneck.

When it comes to the amount of work performed by the master machine, it is clear that WP outperforms BS, but both of these methods are significantly better than Basic. In terms of space complexity, WS and BS are identical and both are better than Basic. From the perspective of the task machines, since both BS and WS implement $\mathbb{H}_2$, they take up a bit more space than Basic which implements $\mathbb{H}_1$. Assuming that all N queries are uniformly distributed in space, each task machine is expected to obtain the same number of keys during the query stage. Thus, Basic needs additional $O(\frac{N \cdot D}{M})$ space for queries, and both BS and WP need additional $O(\frac{N}{M})$ space. Since the lookup time for a HashMap is $O(1)$, the relationship between the time complexity of each task machine is WP= BS $<$ Basic. As we see, in terms of big O, the time complexity of BS and WP in the task machines is the same. However, BS invokes the task machines $\log D$ times, while each task machine in WP makes $\log D$ iterations. This makes WP much more efficient than BS as there is a significant decrease in the network communication cost. Note that during network communication, the sending cost for the master machine dominates the cost. Note also that the Basic, BS, and WP methods need to send $O(N \cdot D)$, $O(N \cdot \log D)$,

and $O(N)$ keys respectively. Therefore, so far, it seems that WP is better than BS, and that BS is better than Basic in general. However, some analytic queries may be very local in nature as they query a large number of proximate source-target pairs. For example, suppose that Spark has loaded the distance oracle of the entire USA road network in memory, and a user wants to know the distance matrix between the hospitals in San Francisco and the locations of their patients. In this case, it could be that all N queries are in the same subtree of the DO-tree, which means that they will be assigned to the same partition by WP. The result is that the time complexity of the working task in WP is $O(N \cdot \log D)$ and its extra space is $O(N)$ for queries. This is the worst case of WP, while both Basic and BS keep the same time complexity, which are $O(\frac{N \cdot D}{M})$ and $O(\frac{N \cdot \log D}{M})$, respectively.

The bottleneck of both Basic and BS is network communication, where the total time complexity of Basic is $O(N \cdot D)$, and the total time complexity of BS is $O(N \cdot \log D)$. WP is better than BS only when the N s-t queries can be assigned to the task machines so that they each have approximately the same number of s-t queries, in which case the total time complexity of WP is $max($ $O(N)$, $O(\frac{N \cdot \log D}{M}))$. If the N s-t queries are assigned to the same task machine or to just a few task machines, then the task machines may become the bottleneck. In addition, since in real applications, the master machine is usually much more powerful than the task machines, BS is a better choice in general.

4.4 Evaluation

In this section, we present a detailed evaluation of our distributed key-value solutions in comparison with the CH method [39] which is a state-of-the-art algorithm for finding a single shortest path in a road network, the distance oracle implementation from [64], and an efficient implementation of Dijkstra's algorithm. The comparisons are detailed in in Section 4.4.2. We evaluate our experimental results on a variety of datasets including a dataset corresponding to the entire USA road network and provide the details in Section 4.4.2. Our comparisons use four workloads: a batch of s-t pairs in Section 4.4.3, distance matrix queries in Section 4.4.4, and job accessibility map in Section 4.4.5. We provide both a local and a distributed implementation of the methods, where in the *local mode* we use a single machine to study relative performance without network communication, and in the *distributed mode* we use a cluster with large number of task machines.

4.4.1 Comparison Methods

We compare the performance of three implementations of our distributed key-value method on the Spark framework. In particular, we compared the Basic method discussed in 4.3.1, the binary search method (BS) in 4.3.2, and the wise partitioning method (WP) discussed in 4.3.3.

DO. We compare against our ASDO method as it is representative of methods that can perform network distance computations inside a database. In this case, we load ASDO as a relational table in PostgreSQL and index it using a B-tree. In the

local mode, we use a single instance of PostgreSQL, while in the distributed mode each machine in the cluster runs an identical copy of the ASDO. Load balancing is achieved using a Java middleware program in the master machine that evenly distributes the query workload to the task machines and later combines the result.

CH. We use the CH method proposed in [39] for comparison as a representative of methods that optimize the execution of single source shortest paths. Note that CH optimizes latency which is the result of computing a single s-t query as quickly as possible, while our approach optimizes throughput. In the local mode, the query is processed using a local CH server implemented in C++, while in the distributed mode, a Java middleware program in the master machine distributes the query workload among the CH programs running on each task machine and later combines the results.

Dijkstra. We compare our method with a high performance implementation of Dijkstra's algorithm [37] from [52], denoted as *Dijkstra* later, since it is a representative of traditional shortest path methods. As in the DO and CH cases, we use a Java middleware to distribute the workload when comparing the performance of algorithms for the distributed mode.

In the case of DO, PostgreSQL is process-based (not threaded) in the sense that each database session is a single system process. In other words, a database connection cannot utilize more than one CPU [14]. To make the comparisons fair, we restrict all methods (i.e., our Spark-based, CH method, and Dijkstra) to utilize just one CPU in each machine.

4.4.2 Datasets and Cluster Setup

Table 4.3: Dataset Characteristics in SPDO

Name	NYC	Bay	US
Region	NYC	Bay Area	USA
# of Nodes	264,346	758,104	23,947,347
# of Arcs	733,846	1,663,662	58,333,344
Maximum Depth ($0.1m$)	20	21	25
Practical Depth D ($100m$)	10	11	15
# of WSPs with $\epsilon = 0.25$	55M	278M	4.6B

Table 4.3 provides the characteristics of the road network datasets used in our evaluation. The NYC and US road networks are from the 9^{th} DIMACS Implementation Challenge [3], and the Bay road network is from OpenStreetMap [11] extracted using the TAREEG [19] tool. The distance oracles that we used in our experiments provide a resolution of 100 meters, which means that the maximum depths D of the DO-tree for the NY, Bay, and US datasets are 10, 11, and 15, respectively. This means that if the source and the destination are closer than 100 meters, then we simply return the geodesic distance between them. We did not take the query time for such queries into consideration in our evaluation. For the rest of the queries where sources and destinations are farther than 100 meters, the ASDO is guaranteed to provide the ϵ-approximate network distance. The length of the Morton code

for a leaf node of a DO-tree is $(4 \cdot D)$. Therefore, we need at most 40, 44, and 60 bits to represent an individual WSP for the NYC, Bay, and US datasets, respectively.

Besides the road network, we use three location datasets in our evaluation. The *Restaurant* dataset consisting of the locations of $49,573$ fast food restaurants in the entire USA was obtained from [5]. We use the LEHD dataset [10] from the US Census Bureau which provides detailed origin-destination employment statistics as pairs of census blocks. Each census block pair has the count of how many people live in one census block and commute to another census block for work. We use two datasets from LEHD, one for the state of New York, called *NY-JOB* consisting of $6,834,157$ location pairs and another for the state of California called *CA-JOB* consisting of $13,645,807$ location pairs. The *local mode* experiments on a single machine ran on an Intel Xeon(R) E3-1225 v3 CPUs @ 3.2GHz (4 cores) with 16 GB RAM. The *distributed mode* experiments ran on a cluster with one master machine and 25 task machines. Each machine consists of $2 \times$ 6-core Intel Xeon E5-2620 v3 CPUs with 64GB RAM and 10GbE ethernet network. Our implementations use Spark 1.3.0, while for the DO method, each task machine has PostgreSQL 9.3.5 installed.

4.4.3 Source-Target Pairs Workload

In this section, we generate a large workload of sources and targets on the road network by uniformly sampling the location pairs from *NY-JOB* restricted to New York City. Such a workload measures the throughput of our and other

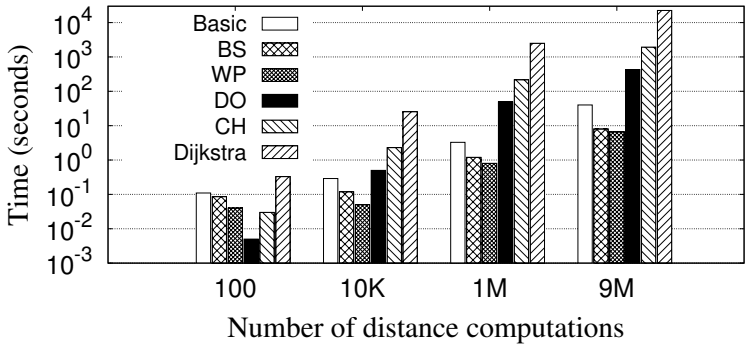
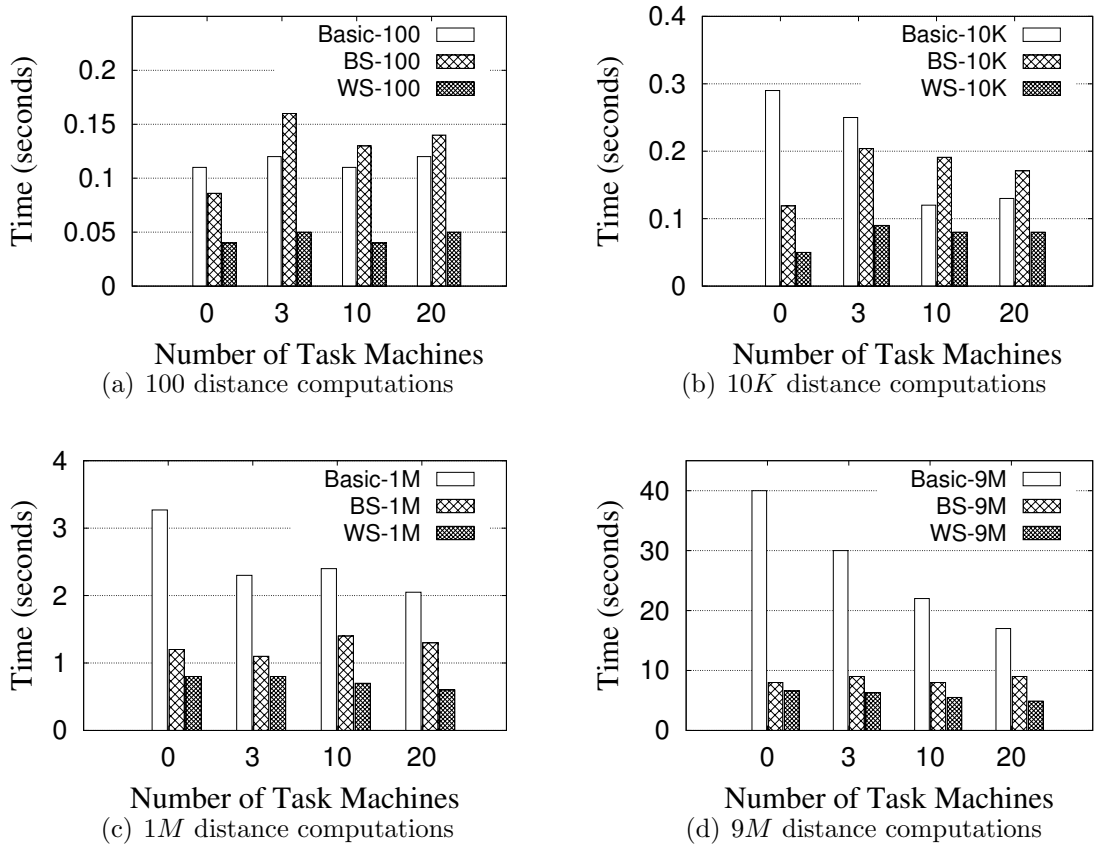


Figure 4.5: Execution time in local mode in NYC for 100, 10 thousand, 1 million, and 9 million s-t pairs. The y-axis is logarithmic showing that our BS and WP methods are significantly faster than other methods.

comparative methods on a workload where there may not be significant commonality across different queries. Section 4.4.4 compares these methods on another workload where network distances from one source to multiple targets with the goal of taking advantage of location commonality.

Since our Spark-based method is a solution that can run on multiple task machines, we want to understand the bottleneck due to the network computation. In order to study this effect, we must first study the performance of our method on a single machine in this section since this represents the case when there is no network communication. Later in this section, we show experimental results on a cluster of task machines. We vary the number of task machines to study its effect on the performance of our Spark-based methods.



(a) 100 distance computations
(b) $10K$ distance computations
(c) $1M$ distance computations
(d) $9M$ distance computations

Figure 4.6: Execution times of computing a batch of s-t queries in New York City using the Spark cluster when varying the number of task machines. The case of 0 task machines corresponds to local mode. As the y-axis is linear scale and the performance of BS and WP is similar while we increase the number of task machines, we see that the bottleneck of our BS and WP methods is the master machine.

4.4.3.1 Local Mode

In the local mode, all experiments are performed on a single machine, so that there are no network communication issues. For this set of experiments, we use the NYC dataset and the s-t queries sampled from *NY-JOB*. We use the smaller NYC dataset since all the WSPs for our Spark method must fit in memory, which would

not be possible using a much larger dataset such as the US dataset.

Figure 4.5 shows the execution time of our Spark-based methods and other competing methods for varying the number of s-t queries. In particular, we vary the size of our workload from 100 to 9 million network distance queries. The result in Figure 4.5 shows that the both the BS and WP methods are better than Basic method, and WP is slightly better than BS since it pushes the $logD$ searches into the task machines. In terms of throughput in the local mode, BS and WP achieves a throughput as high as 1.125 million and 1.363 million distance computations/second in NYC, respectively. A reason for similar performance of BS and WP is that without network communication costs, there is little difference between a binary search performed at the master program or at the task program.

Not surprisingly, Dijkstra's algorithm performed the worst since it needs to invoke a best-first scans [52] for each query in the workload, which can be expensive. Both DO and CH are better than our methods for the workload of size 100, because our methods have the fixed overhead of job setup and scheduling in Spark, which is the dominant cost for the small query workload. Both WP and BS are significantly better than competing methods (orders of magnitude for larger query workload for 1M and 9M cases) as the size of the workload increases. Even the Basic method outperforms all the competing methods for the query workload larger than 100 s-t pairs. These results show that for a single machine and a small road network dataset, our methods are significantly better than all the other competing ones in the absence of network communication issues.

4.4.3.2 Distributed Mode

In this section, we study the effect of adding more task machines on the performance of our Spark-based methods. We first show in Figure 4.6 how the number of task machines influences the time performance of our methods. The query workload here is exactly the same as one in the local mode in the previous section. The case of 0 task machines corresponds to local mode. In this figure, BS is only slower than Basic when the number of distance computations in the workload is very small (e.g., 100 and a pair of instances of 10K). This means that the Basic algorithm's strategy of generating D keys per query still does not blow up the space for such smaller datasets. However, as the datasets get larger than 10K, this turns out to be a bad strategy since BS is far superior to Basic for the remaining cases. Comparing BS and WP in Figure 4.6, we see that both have very similar performance with WP being always better than BS. The improvements that we see here are due to the decrease in communication costs between the task machines and the master as well as the reduced load on the master.

Another key observation is that increasing the number of task machines does not necessarily result in better performance. This is especially true for BS and WS. It is consistent with our analysis in Section 4.3.4, which indicates that the bottleneck of BS and WP is the master machine. We recommend that the number of task machines in a Spark cluster be set so that the total size of the distance oracles fits in the total distributed-memory of the Spark cluster. Therefore, 1-3, 1-5, and 20-25 task machines be utilized in a Spark cluster for the NYC, Bay, and

US datasets, respectively. In order to process more distance computations with a Spark cluster consisting of M task machines, people can also build $\frac{M}{20}$ sub-clusters, where each sub-cluster can now load the distance oracles of the entire US dataset.

Table 4.4: Throughput of the 6 methods for the US dataset running on 20 task machines in SPDO

Method	Basic	BS	WP	DO	CH	Dijkstra
dist/sec/machine	5.0K	25.0K	73.8K	18.8K	385	1.6

We now analyze the performance of our methods and competing approaches in terms of throughput. Table 4.4 summarizes the throughput of the 6 methods running on a cluster of 20 machines for the random s-t queries for the US dataset. WP is the best one, which achieves a throughput as high as $74K$ distance computations/second per machine, which is nearly $4\times$ better than the DO approach. Note that the total throughput of WP in the cluster is 1.47 million distance computations per second. The throughputs of Basic and BS methods are much lower than the one of WP due to the network communication of the master machine is the bottleneck. CH and Dijkstra methods have lower throughputs since computing the network distance using CH and Dijkstra is much slower especially if the source and target are far from each other.

4.4.4 Distance Matrix Workload

The distance matrix query is the simplest form of an analytic query that takes a set of n locations on a road network and computes the network distance

between every pair of locations. In other words, it computes an $n \times n$ matrix as the output which requires computing n^2 network distances. Typically, these distance matrices find use in logistics queries where the network distances between all pairs of objects (e.g., locations of packages to be delivered) on a road network are computed which in turn forms the input to complex optimization problems. In the following experiments, we use the distance matrix construction query to evaluate the performance of various methods.

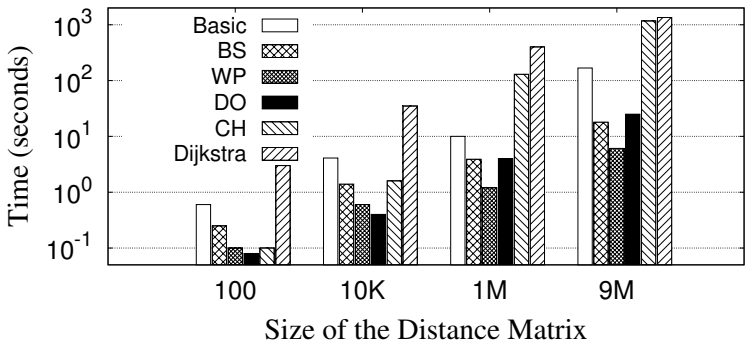


Figure 4.7: Distance matrix computation queries for various methods on US dataset with 20 task machines

Figure 4.7 corresponds to the distance matrix query using the road network of the US for randomly chosen 10, 100, 1000 and 3000 locations from the *Restaurants.* dataset and compute a distance matrix for these inputs. For this experiment, we use the USA road network dataset and 20 task machines. For Dijkstra's algorithm, our implementation automatically optimizes any $n \times n$ distance matrix queries into n one-to-many best-first scans. See [52] for details of this optimization.

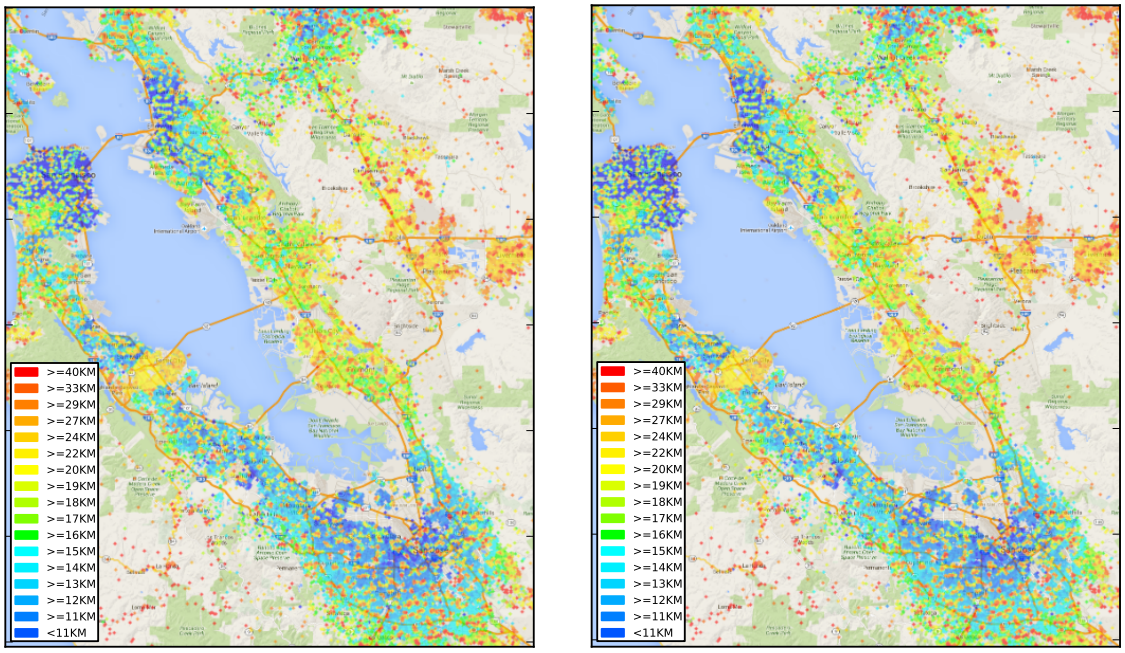
The BS and WP methods are clearly superior to every other method, DO included, whenever the size of the matrix becomes 1000×1000 or larger. In the

3000×3000 distance matrix, WP achieved a total throughput of nearly 1.5 million distance computations per second for the 20 node cluster or close to $75K$ distance computations/second per machine. For smaller queries, we found that the cost of setting up the query dominates the computation times for the Spark methods.

We use the above experiment to shed light on another performance tuning aspect of the Basic, BS and WS methods. The distance oracles of the USA road network take up about 330GB so each task machine needs much memory to maintain a local hash map, i,e, IndexedRDD. In this environment that consumes so much memory, we found that the number of partitions of the distance oracles in the task machines also influences the performance of Spark. For example, in the above experiment, the distance oracle of the US is partitioned into 5000 parts by IndexedRDD. Decreasing the number of partitions of the distance oracle, e.g., 1000, results in a lower time cost in the best case, but worse fault tolerance, which means Spark is more frequent to rerun some sub-tasks of a job because of failure.

4.4.5 Job Accessibility

An important application that performs millions of network distance computations is the analysis of how far people travel to work. The State Smart Transportation Initiative (SSTI [18]) pointed out that measures of accessibility can reveal if a transportation system meets peoples needs [18], not to mention revealing the economic vibrancy of a census block. The dataset that is used for such an analysis is the LEHD dataset [10] from the US census which first subdivides the map into



(a) Using the WP method

(b) Using the CH method

Figure 4.8: Average drive distance from home to workplace in the Bay Area region, which contains 2.1 million source-target pairs from *CA-JOB*: (a) results computed by WP with 3 task machines in 2 secs; (b) results computed by CH with 3 task machines in 5 mins. Results in (a) are almost the same as (b). Although the distance values yielded by the distance oracles are ϵ-approximate, with $\epsilon = 0.25$, they are definitely sufficient for such analytic queries.

census blocks and for each block pair tabulates the number of people that commute from one block (where they live) to another block (where they work). A natural query is one that seeks for each census block the average distance traveled to work by each of its inhabitants. Such a query requires computing millions of shortest path queries. For instance, *CA-JOB* has more than 13 million such census block pairs and a visualization of such a query using this dataset was shown in Figure 4.1. Our WP method generated Figure 4.1 in 13 seconds using 5 task machines, while

using the same number of task machines CH needs 20 minutes. Figure 4.8 shows the result for a small section of California, i.e., San Francisco Bay Area. Figures 4.8(a) and 4.8(b) show the results of using the WP and CH methods, respectively.

4.5 Related Work

The methods for computing shortest distances fall into two main categories: scan-based methods and lookup-based methods. Scan-based methods are usually memory-based, which require many data structures to keep the scan information such as the graph and priority queues. Lookup-based methods have precomputed and stored many shortest distances result, and then just retrieve and merge the distance result for online queries. In our experience, lookup-based methods are more likely to be embedded in a distributed framework.

The most famous scan-based method is Dijkstra's algorithm [37], which is very efficient for single source queries named *one-to-many pattern* in [52], e.g., find the nearest K destinations around a given source. However, for an s-t query, Dijkstra's algorithm has to scan many unneeded vertices to reach the given target location. To address the deficiency of Dijkstra's algorithm on road networks for the s-t queries, a variety of scan-based techniques have been proposed based on noting that some vertices in a spatial network are more important for shortest path queries, while offering different trade-offs between preprocessing time, storage usage, and query time. In particular, [40] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. *Contraction Hierarchies* (CH) [39] assigns an importance

score to each node and replaces some original edges by shortcuts. [22, 25, 34, 61, 76] build an explicit hierarchy graph to overcome Dijkstra's algorithm's drawback.

The lookup-based methods usually need to store some precomputed results. [21, 30, 43, 55] precompute the shortest distance between landmarks or hub nodes and other vertices, and then answer the shortest distance queries by assuming the shortest path is through one landmark or hub node. HLDB [20] based on hub labels (HL) [21] is a recent practical method that embeds the shortest distance computation into an RDBMS. *Road Network Embedding* (RNE) [69] applies a Lipschitz embedding [42] to a spatial networks, so that vertices of the spatial network become points in a high-dimensional vector space. [72] takes advantage of the fact that the shortest paths from vertex u to all other vertices to decompose the vertices into subsets based on the first edges on the shortest paths from u to them. *Spatially Induced Linkage Cognizance* (SILC) [60] is based on the observation mentioned in [72] which decomposes vertices into multiple quadtree blocks for each vertex u so that the shortest paths from u to all vertices in a block are reachable from the same outgoing edge from u. Our previous work [64, 67] exploit the spatial coherence so that if two clusters of vertices are sufficiently far away, then distances between pairs of points in different clusters are similar. The *Path-Coherent Pairs Decomposition* (PCPD) [67] gives one exact shortest path algorithm, while the *ϵ-Distance Oracles* (ϵ-DO) [64] proposes an approximate shortest distance algorithm, which balances between accuracy and storage.

A few other approaches focus on speeding up specific analytic queries. Knopp et al. [41] explain how to use highway hierarchies [61] for computing many-to-many

shortest distances. [60] and [69] show how to speed up the K nearest neighbor search by their s-t techniques, respectively. Delling et al. [36] utilize partition-based algorithms developed for s-t queries to handle POI queries. Cho et al. propose UNICONS [31] for continuous nearest neighbor queries, and ALPS [32] for top-k spatial preference search. Our recent paper [52] proposes an integrated architecture that embeds the distance oracles into an RDBMS, and develops many SQL solutions for solving a variety of spatial analytic queries.

4.6 Summary

We presented SPDO, a framework for computing road network distances using Apache Spark and ϵ-approximate distance oracles. We described three algorithms for mapping a distance oracle into a distributed hash structure, which we implemented using Spark's RDD abstraction. Our methods produced at least an order of magnitude higher throughput compared to existing methods that are optimized for latency, and up to 1.5 million distance computations per second in both NYC and US road networks. Using our framework, one can perform millions of distance computations on a road network using just a few machines. We also showed how SPDO can significantly speed up complex spatial analytic queries and discussed two complex use-cases, namely the route directness spectrum and job accessibility.

Chapter 5: CDO: Extremely High-Throughput Road Distance Computations on City Road Networks

5.1 Overview

Some analytic queries on road networks, usually concentrating in a local area spanning several cities, need a high-throughput solution such as performing millions of shortest distance computations per second. However, most existing solutions achieve less than $5,000$ shortest distance computations per second per machine even with multi-threads. We demonstrate a solution, termed *City Distance Oracles* (CDO), using our previously developed ϵ-distance oracle to achieve as many as 7 million shortest distance computations per second per commodity machine on a city road network, i.e., $10K \times 10K$ origin-distance (OD) matrix can be finished in 14 seconds.

Browsing of spatial data is becoming increasingly important [27, 38, 58, 59]. During the spatial analyst's exploration, some types of spatial queries, termed *spatial analytic queries*, can potentially involve thousands to millions of road network distance computations. Examples of such analyses include complex scenarios such as how to assign and deliver $10,000$ packages for UPS in a city, how much traffic

congestion could be reduced if build a new bridge, where to locate the next supermarket among a number of potential locations taking into account a variety of factors like demography, distance to a warehouse, etc., identifying bottlenecks in a road network for evacuation planning, or distance join queries on road networks [63].

Our focus here is on *throughput* which is how to compute a spatial analytic query as quickly as possible. Note that although decreasing the latency time for a single shortest distance query results in reducing the total response time for a spatial analytic query, it is far from enough since these latency methods don't take into account considerations such as multi-users, multi-threads, reused results, and query optimization [52]. Our recent work [54] discussed how to obtain high throughput performance using the ϵ-distance oracle (ϵ-DO) [64, 65] in a distributed key-value store such as Apache Spark for spatial analysis on continental road networks such as the entire USA. However, the reaction of a number of companies that make use of such queries was that typical queries are concentrated in a small local region rather than the whole continental region, termed the *spatial concentration* property. As an example of such a use-case is a delivery company that needs to plan the delivery route for each truck every day, where the route of each truck must be restricted into a local region, i.e., the region near to the package warehouse. In particular, each such warehouse handles $1,000$ to $10,000$ packages per day, and each truck can deliver a maximum of 150 packages per route per day. In order to efficiently assign the packages to trucks and plan routes, the delivery company computes a distance matrix that captures the distance between every pair of destination locations of the packages, This is a common spatial analytic query which makes between 1 million

Figure 5.1: The work flow of demo CDO: First extract any city road network such as New York City from OpenStreetMap [11] and TAREEG [19]; Then precompute the ϵ-DO [64]; Finally load the results in memory and implement multi-thread version to process query workload.

and 100 million distance computations. Here, the *spatial concentration* property means that in the general case, all destination locations of packages must be in proximity to the warehouse, say within 100KM.

Now we demonstrate an extremely efficient solution to solve spatial analytic queries where the spatial concentration property holds. In particular, we will show how one can compute large origin-distance (OD) matrix, say of size $10K \times 10K$ in a few seconds. The main work flow is shown in Figure 5.1. We extend our prior work on ϵ-DO [64, 65], which precomputes and stores approximations of the shortest distances between all pairs of vertices in a road network. The resulting representation takes $O(\frac{n}{\epsilon^2})$ space, where n is the number of vertices in the road network and ϵ is an approximation error bound on the result. Our contributions in this demo are:

1. An efficient implementation of using ϵ-DO in memory instead of in a database [64,

66] with multi-threads and query optimization illustrated in [52]. As a result, we achieve 7 million distance computations per second on the Bay Area road network in latitude/longitude region $[37.173, 38.019] \times [-122.678, -121.571]$ with $755K$ vertices.

2. The design of a new key representation of the ϵ-DO which enables doing a binary search to retrieve the road network distance on the ϵ-DO without requiring any special indices. It greatly speeds up the query time.

3. An application of CDO is illustrated, and an evaluation of time performance is provided for CDO, HLDB [20], and CH [39].

In addition, we set up the CDO demo for the Bay Area and New York City [1] and provide some use cases in our blog site [2].

5.2 Preliminaries and Examples

Recall the two concepts we previously introduced, Morton code in Figure 5.2 and well separate pairs in Figure 5.3. We use the Morton (Z) order space-filling curve [57] that provides a mapping, $\mathbb{Z}^2 \rightarrow \mathbb{Z}$, of a multidimensional object (e.g., a vertex or a quadtree block) in a 2-dimensional embedding space to a positive number. Given an object o, let $mc(o)$ be the mapping function that produces the Morton representation of o by interleaving the binary representations of its coordinate values. Figure 5.3 illustrates the theoretical WSP example and a real WSP on road network.

[1] `http://sametnginx.umiacs.umd.edu/`

[2] `http://roadsindb.com/`

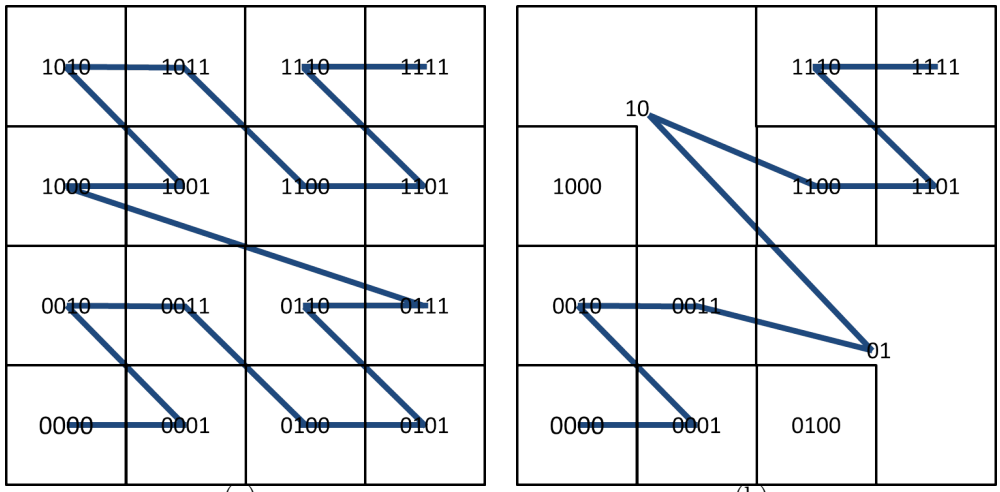


Figure 5.2: (a) Morton code and ordering in a 4×4 space. (b) Example to illustrate the key representation of distance oracle.

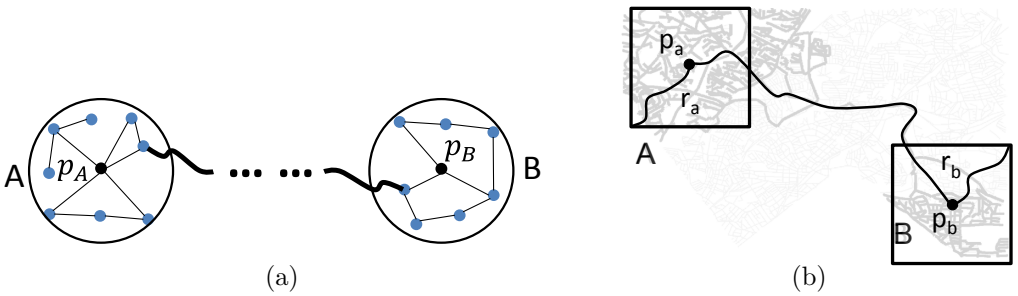


Figure 5.3: A well-separated pair example: (a) A theoretical WSP example. (b) A potential oracle containing blocks A and B in Silver Spring, MD showing representative vertices p_a, p_b and radius r_a and r_b.

5.3 Method

CDO is based on ϵ-DO on a city road network, e.g., the number of vertices n is less than one million. Note that on a commodity machine, immediately loading $O(n^2)$ distance results in memory is infeasible when n is larger than 50 thousand.

As a result, the ϵ-distance oracle method generates $O(\frac{n}{\epsilon^2})$ well-separated pairs, denoted as $(A, B, d_\epsilon(A, B))$. Both A and B are a pair of PR quadtree blocks [57] at the same depth. In order to make a well-separated pair easy to embed in a database as a key-value pair, the ϵ-distance oracle uses the Morton (Z) order space-filling curve [57] to map a quadtree block in a 2-dimensional embedding space to a positive number. Thus, each well-separated pair $(A, B, d_\epsilon(A, B))$ is considered as a key-value pair $(mc(mc(A), mc(B)), d_\epsilon(A, B))$, where the value is the distance and the key is $mc(mc(A), mc(B))$.

5.3.1 Storing and Querying CDO

Here we illustrate how to obtain the network distance in CDO, given a source location $p_1 = (lat_1, lng_1)$ and a destination location $p_2 = (lat_2, lng_2)$. Once ϵ-DO has been computed, CDO loads all well-separated pairs, which the schema is (code, d), in memory as an array sorted by code, where code is a succinct representation of the well-separated pair and d is the approximate network distance. Although such a schema is similar to the one proposed in ϵ-DO [64], our method just uses the default integer comparator instead of redefining the string comparator operators (i.e., $<$ and $=$) while doing binary search. This is important because the default integer

comparator saves much time in contrast to the redefined string comparator.

To illustrate our method for packing the code, we first start with a simpler two-dimensional example (i.e., Z_2) and we then describe how to encode a well-separated pair as a four-dimensional Morton block. Suppose that we have a number of various length Morton codes in two-dimensions, which means that the corresponding quadtree blocks are at different depths. The simpler problem we want to solve is that we are given a point p, and we need to efficiently find a unique quadtree block A containing p. Here we assume that the uniqueness property from the property of WSP [28] is also true in this simpler example. The uniqueness property here means that there is exactly one quadtree block containing p such as in Figure 5.2. This search problem is equivalent to finding the unique $mc(A)$ such that $p \prec A$.

Our approach is to make all the Morton codes have the same length by padding them with enough zeros, so that all Morton codes are always the same length, i.e., $2 \cdot L$ bits long in two-dimensions. For any Morton code $mc(A)$, padding with enough zeros is equivalent to choosing a unit-sized block that is a descendant of A in the quadtree that has the smallest Morton code. This needs to be done carefully as we illustrate with the following example. Suppose that our two-dimensional oracles has ten quadtree blocks as in Figure 5.2 whose Morton codes are 0000, 0001, 0010, 0011, 01, 10, 1100, 1101, 1110, and 1111. Only two Morton codes 01 and 10 are not 4 digits long. Thus, consider the quadtree blocks 01 and 10 in Figure 5.2, which we convert to 0100 and 1000 respectively by padding zeros to the right hand side. The codes of our oracle become: 0000, 0001, 0010, 0011, 0100, 1000, 1100, 1101, 1110, and 1111 in order. Given a query point $p = 0111$ that is contained by a unique

quadtree block A. To find A, we need to find a quadtree block in the B-tree such that it is the largest value that is less than or equal to p, which in this case is 0100 (i.e., quadtree block 01, which is the correct answer).

Algorithm 6: GETDIST(lat_1, lng_1, lat_2, lng_2)

Data: A: Sorted array containing all well separated pairs

Result: *Result*: Network distances between (lat_1, lng_1) and (lat_2, lng_2)

```
1  code ← Z_4^0( Z_2((lat_1, lng_1)), Z_2((lat_2, lng_2)) );
2  left ← 0;
3  right ← A.size();
4  while left ≤ right do
5      mid ← (left + right)/2;
6      if A[mid].code ≤ code then
7          left ← mid + 1;
8      else
9          right ← mid - 1;
10 return A[left − 1].d;
```

Now going back to CDO, we obtain a four dimensional Morton code by interleaving $mc(A)$ and $mc(B)$ two digits at a time. This packing is given by the function $Z_4(A, B)$. Next, we define function $Z_4^0(A, B)$ by padding $Z_4(A, B)$ with zeros to the right side. For example for the blocks in Figure 5.2, Z_4 and Z_4^0 should be

$$Z_4(01, 10) = 0110 \qquad Z_4^0(01, 10) = 01100000 \tag{5.1}$$

$$Z_4(0000, 1111) = Z_4^0(0000, 1111) = 00110011 \tag{5.2}$$

This packing Z_4^0 produces a Morton code of $4 \cdot L$ bits length. This forms the *code* attribute. At this point, given a source location p_1 and a destination location p_2, the approximate network distance query first calculates $key = Z_4^0(mc(p_1), mc(p_2))$ in $O(1)$ time using bitwise operations and then issues the following binary search function, Algorithm 6, to obtain the network distance.

The reason this scheme works is because of the uniqueness property of WSP. For any two points in the domain S, there is exactly one WSP containing them.

5.3.2 Multi-threads

As most memory resources in Algorithm 6 are only processed by read-only operations, parallel processing should increase the throughput a lot. Without loss of generality, we show how to obtain the network distances of a batch of source-target pairs with multi-threads in Algorithm 7.

5.4 Demo scenario

We extracted and prepared a CDO of the Bay Area road network with $781K$ vertices and one for the New York City road network with $407K$ vertices from OpenStreetMap [11]. The demo is set up at `http://sametnginx.umiacs.umd.edu/`.

We implemented three methods: our solution *CDO* with $\epsilon = 0.05$, HLDB [20], and CH [39], in C++. All of them are processing *in memory* with multi-threads, and in the same environment, a Macbook Pro 15-inch, 2.8 GHz Quad-core Intel

Algorithm 7: GetBatchDist(Q)

Data: Q: Query array where each element is a source-target pair; *nthread*: Number of threads

Result: *Result*: Network distances for Q

```
1 len ← Q.size()/nthread;
2 ans ← initial a float array with Q.size() elements;
3 for i ← 0 to nthread do
4     thread[i] ← initial thread i, process Q[j], j ∈ [i · len, (i + 1) · len) by
      Algorithm 6 in thread i, and store distance results in corresponding ans[j]
      ;
5 for i ← 0 to nthread do
6     thread[i].join();
7 return ans;
```

Core i7, 16 GB memory.

Figure 5.4 shows the time performance of the three methods on the Bay Area road network. CDO is the fastest one, which takes 0.16 seconds with 8 threads for 1 million distance computations.

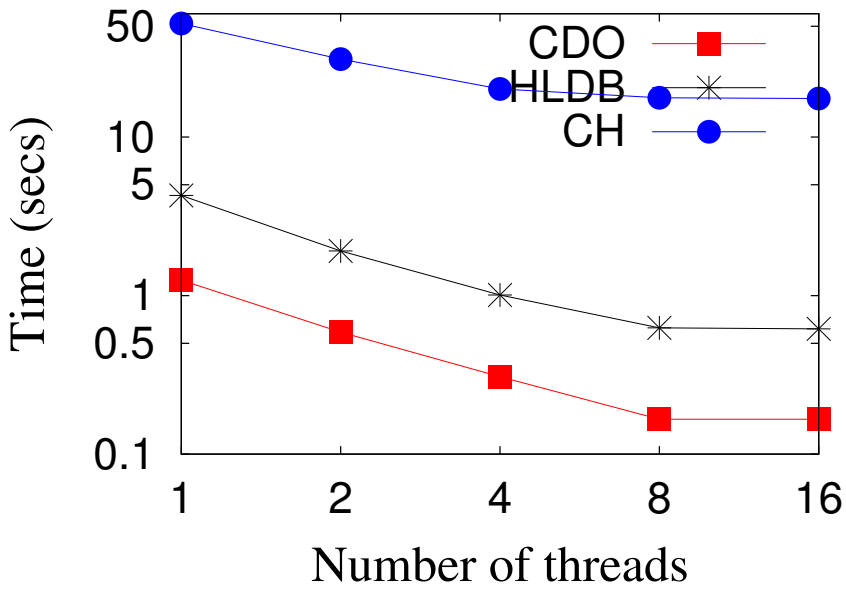


Figure 5.4: Time consumption for 1 million random source-target pairs on the Bay Area road network, varying with number of threads.

In addition, here we provide an application that can be efficiently solved by our demo. It is to measure the accessibility of jobs, i.e., how many job opportunities exist nearby each census block. We use the LEHD dataset [10] to obtain the job locations around the Bay Area. This workload shown in Figure 5.5 makes 120 million distance computations, where CDO only takes 18 seconds. Obviously, based on our solution, many analytic queries could be solved and visualized in a much quicker way, such as showing the influence of building a bridge from two arbitrary Bay Area

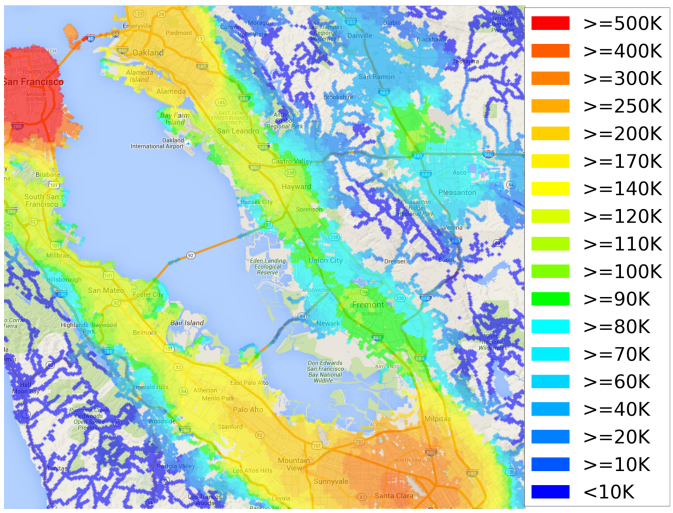


Figure 5.5: Nearby job opportunities (e.g., within 10 kms) for each census block in the Bay Area, requiring 120 million distance computations, where CDO finished it within 18 seconds.

locations. Future work involves investigating the use of distributed data structures in the application (e.g., [71]).

Chapter 6: DOS: A Spatial System Offering Extremely High-Throughput Road Distance Computations

6.1 Overview

Our previous work discussed how to process graph distance computations in a PostgreSQL database on a general road network, e.g., $60K$ distance computations per second per machine; how to "scale out" by using a Spark cluster to achieve $73.8K$ distance computations per second per machine; and how to obtain a extremely high-throughput solution in memory for city-sized road networks, e.g., $6.7M$ distance computations per second. However, there is no solution that could achieve more than $1M$ throughput for general road networks. In an industrial setting, most state-of-the-art solutions yield $5K - 10K$ shortest distance computations per second per machine even with multi-threads. In this chapter, we propose a new distance oracle system (DOS) for general road networks. It can solve most spatial analytic queries, and its throughput achieves $5M$ distance computations per second even on the whole USA road network. In addition, a $10K \times 10K$ origin-distance (OD) matrix can be computed in 20 seconds.

Beyond simple navigation queries, location-based web services like Google

Maps repeatedly pose queries on a road network and utilize the results to serve a user base. For example, the Google Distance Matrix product offers an API that computes the distance matrix between a set of origins and a set of destinations. Other examples include complex scenarios such as how to assign and deliver $10,000$ packages for UPS in a city, how much traffic congestion could be reduced if a new bridge is built, where to locate the next supermarket among a number of potential locations taking into account a variety of factors such as, but not limited to, demography, distance to a warehouse, etc., identifying bottlenecks in a road network for evacuation planning, or distance join queries on road networks [63]. We use the term *spatial analytic queries* to collectively describe such queries. The challenge lies in taking note of the realization that each such instance of a spatial analytic query invariably involves being able to make hundreds to as many as millions of computations of distance along a spatial network rather than as the crow flies.

In the face of a massive amount of spatial analytic queries from internet scale users, for example, Google Maps [8] drastically restricts the number of shortest distance results per query (e.g., a limit of 625 (O-D matrices of size 25×25) shortest distances per query using the Google Distance Matrix API even to their paying customers). Most other existing services such as Yelp just use the Euclidean distance instead of the network distance. Figure 1.3 illustrates the drawback of using the Euclidean distance in Google Maps and Yelp, respectively.

Reviewing previous research work, we find none that are concerned with general spatial analytic queries. Instead, they focus on speeding up one specific type of query, e.g., KNN search queries [32, 36, 47, 50], CNN queries [31], and distance

matrix [41]. However, these algorithms are not easy to extend to include general spatial analytic queries. On the other hand, most state-of-the-art methods such as HL [21], TNR [25], CH [39], etc, focus on decreasing the latency time for a single source-target (s-t) query, which is the basic unit of a spatial analytic query. Although decreasing the latency time for one s-t query results in reducing the total response time for a spatial analytic query, it is far from enough since these methods don't take into account considerations such as cache results, multi-threads, and distributed systems that can be used to speed up a spatial analytic query [52].

Our focus here is on *throughput* which is how to compute a spatial analytic query as quickly as possible. The first attempt uses our fundamental theory work ϵ-distance oracle (ϵ-DO) [64] and PCPD [67] methods. It computes ϵ-approximate network distance based on the *oracle representation* of a road network requiring $O(\frac{n}{\epsilon^2})$ space for n vertices and an ϵ error bound. As noted in [74], this method is not scalable to road networks with more than $80,000$ vertices. It means that ϵ-DO is not available even for a city road network such as New York City with $264,346$ vertices. Our previous work SPDO [54] discussed how to obtain high throughput performance using ϵ-DO in a distributed key-value store such as Apache Spark for spatial analysis on the continental road networks such as the entire USA. Our previous demo work CDO [53] utilized the *spatial concentration* property to obtain an extremely high-throughput, e.g., $6.7M$ distance computations per second for city-sized road networks. Note that the *spatial concentration* property only holds for some specific companies such as delivery companies, but not for general.

In this chapter, we propose a system called DOS (denoting Distance Ora-

cle System) for spatial analytic queries on general road networks. Its throughput achieves $5M$ distance computations per second, which is very close to the performance of CDO [53] on a city-sized road network, but is also available for large road networks. Contributions of DOS are:

1. We developed an infrastructure to precompute the oracle representation for large road networks. Our experimental results show that the preprocessing needed to form the oracle for the entire USA road network can be performed in 5.1 hours when using a modest size cluster of 45 Amazon EC2 machines incurring less than $50 in AWS charges.

2. DOS extends our demo work CDO, which garnered a best demo award at the SIGSPATIAL'16. It is an efficient implementation of using ϵ-DO on disk and cached in memory instead of in a database [64, 66] with multi-threads and query optimization illustrated in [52]. As a result, we achieve 5 million distance computations per second for both city-sized road networks, e.g., the Bay Area road network, and the country-sized road networks, e.g., the USA road network.

3. DOS utilizes FlatBuffers [6] to serialize distance oracles to binary files. It greatly reduces the required space for storage. Then DOS uses *mmap* to load binary files in program. It makes the preloading time instant even for large size of distance oracles.

4. We show how to solve some representative industrial queries in DOS, and provide a detailed execution time evaluation for DOS, CDO [53], DO [64],

HLDB [20], and CH [39].

In addition, all applications mentioned in this chapter are provided in our distance oracle demo [1] and in our blog site [2].

The rest of this chapter is organized as follows. Section 6.4 summarizes related work. Section 6.2 presents our DOS framework and techniques in detail. Section 6.3 contains a detailed experimental evaluation between our DOS and other solutions. Section 6.5 draws concluding remarks.

6.2 Method

Table 6.1 summarize the notations used latter in this chapter. DOS includes two parts, precomputing and querying. The first part is precomputing the $DO(G)$, and the other one is processing spatial analytic queries utilizing $DO(G)$. In this section, we first show the main picture of DOS. Next, explain how we speed up precomputing the $DO(G)$. Finally, discuss each component of the querying part of DOS, and how we process some spatial queries.

6.2.1 DOS framework

Our previous chapters discussed how to utilize $DO(G)$ in different queries and settings. In SPDO [54], we proposed a general distributed framework to achieve a high throughput relying on the distance oracle of the whole USA road network. In contrast, when presenting the CDO [53], we claimed that the reaction of some com-

[1] `http://sametnginx.umiacs.umd.edu/`

[2] `http://roadsindb.com/`

Table 6.1: Notation Summary of DOS

Symbol	Meaning
n	the number of vertices in the graph
N	the number of s-t queries
ϵ	the error bound of the ϵ-DO
$mc()$	Morton code function
$d_G(s,t)$	the shortest distance/time from s to t
$d_E(s,t)$	the Euclidean/geodesic distance from s to t
$d_\epsilon(s,t)$	the approximate distance/time from s to t bounded by ϵ
$DO(G)$	the distance oracle representation of a road network G

panies, such as UPS, Uber, PlaceIQ, etc, was that typical queries are concentrated in a small local region rather than the whole continental region, termed the *spatial concentration* property. Such applications with the spatial concentration property require an extremely high-throughput solution such as CDO, which optimizes our distance oracle technique on a small road network (city-sized road network) with limited computing resources (a commodity machine) to achieve millions of distance computations per second.

Speeding up precomputing $DO(G)$ and extending the CDO solution, we propose the DOS system, illustrated in Figure 6.1, for general road networks, not limited to a city. In the preparation stage, which was described in Section 2.2 in detail, we

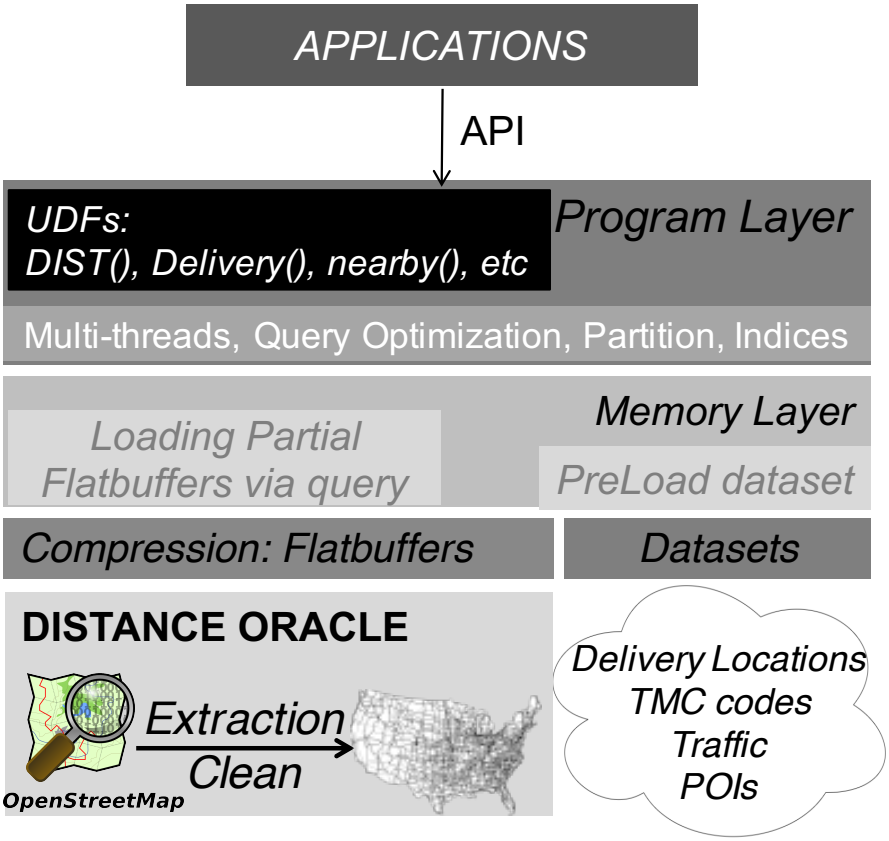


Figure 6.1: The DOS framework

first extract the road network for any given region such as the whole USA road network from OpenStreetMap [11] and TAREEG [19], and then precompute the $DO(G)$. The distance oracle result is ordered and partitioned by the Z_4 code and stored in several text files. The number of text files depends on the size of the road network. In particular, this partition is the same as the WP method in SPDO [54], which puts the nearby WSPs in the same text file to preserve the locality of WSPs. Each text file stores around 100 million WSPs because of the limit of Flatbuffers [6] (explained in detail in Section 6.2.2). In this way, given a pair of two locations (p_1, p_2) and its $Z_4(p_1, p_2)$ code, we can quickly know which text file contains the

WSP for $Z_4(p_1, p_2)$. Next, each text file is compressed and serialized by Flatbuffers to a binary file. The Flatbuffers binary file reduces the disk size significantly, and enables the program to flexibly access WSPs during the querying stage. The details of using Flatbuffers are given in 6.2.2.

In the querying stage, our program preloads the required dataset other than distance oracles in memory, e.g., all delivery locations for delivery tasks, restaurant positions for nearby search, or home and work places for traffic analysis. In order to use distance oracles, we formalize queries as hundreds to millions one-to-one distance queries [52]. Each one-to-one distance query could be answered by binary search in time $O(\log \frac{n}{\epsilon^2})$ [54], which means that each one-to-one distance query would visit at most $O(\log \frac{n}{\epsilon^2})$ WSPs. Thus, at the beginning, all WSPs are in the Flatbuffers binary file on disk, and our program only has the iterator pointers of the binary files. While querying, our program caches the visited WSPs in memory to speed up the binary search of further queries. Our experiments in Section 6.3 prove that using Flatbuffers is an efficient way for both storage size and querying time. Furthermore, our experiments show how powerful our system is while utilizing multi-threads, efficient query plans, and indices.

6.2.2 Flatbuffers Binary Representation

Recall back to CDO, we obtain a four dimensional Morton code by interleaving $mc(A)$ and $mc(B)$ two digits at a time. This packing is given by the function $Z_4(A, B)$. Next, we define function $Z_4^0(A, B)$ by padding $Z_4(A, B)$ with zeros to the

right side. For example for the blocks in Figure 5.2, Z_4 and Z_4^0 should be

$$Z_4(01, 10) = 0110 \qquad Z_4^0(01, 10) = 01100000 \tag{6.1}$$

$$Z_4(0000, 1111) = Z_4^0(0000, 1111) = 00110011 \tag{6.2}$$

This packing Z_4^0 produces a Morton code of $4 \cdot L$ bits length. This forms the *code* attribute. At this point, given a source location p_1 and a destination location p_2, the approximate network distance query first calculates $key = Z_4^0(mc(p_1), mc(p_2))$ in $O(1)$ time using bitwise operations and then issues a binary search call to obtain the network distance.

Algorithm 8: Oracle.fbs for Flatbuffers in C++

```
1 namespace MyOracle.Sample;
2 struct Wsp {
3     code:long;
4     d:float;
5 }
6 table Oracle {
7     wsps: [Wsp];
8 }
9 root_type Oracle;
```

Obviously, each WSP is a bigint for the Z_4^0 value and a float for the network distance value, which should amount to 12 bytes. The CDO [53] stores all WSPs as an array in memory. However, the size of distance oracles of some road networks is

too large to fit in the memory of a commodity machine. For example, the distance oracles for the USA road network contains 4.6 billion WSPs with $\epsilon = 0.25$. If each WSP requires 12 bytes, it is expected at least a storage of 55.2GB. To solve it, our previous work [52] stored such big distance oracles in PostgreSQL, and SPDO [54] enables it to work in a distributed memory system. Both these methods have a heavy overhead in storage due to the extra bytes of the data header, plus the space for index. In fact, these two methods need more than 300GB to store 4.6 billion WSPs, and the throughput performance is less than 100K distance computations per second per machine. Although these methods are faster than all other state-of-the-art methods, they are much lower than 6.7M per second in CDO because of the heavy I/O cost,

In order to enable distance oracles for large road networks to also fit in a commodity machine without sacrificing time performance, we need to find a way to serialize and compress distance oracles on disk and cache the WSPs during querying. After investigation, we use FlatBuffers to serialize distance oracles. FlatBuffers is an efficient cross platform serialization library for performance-critical applications [6]. Although Protocol Buffers [15] is indeed relatively similar to FlatBuffers, since FlatBuffers does not need a parsing/ unpacking step before we access data, FlatBuffers is faster than Protocol Buffers in our setting.

The way of using FlatBuffers is as follows: (1) Take the oracle representation of a road network and represent each WSP as a proto; (2) Use FlatBuffers to compress the representation; (3) Load FlatBuffers into through mmap; (4) Use their support for binary search.

Algorithm 9: oracle_serialize.cpp for serialization preparation

```
1  #include "oracle_generated.h"
2  using namespace MyOracle::Sample;
3
4  std::vector¡Wsp¿ wsp_vector;
5  Void PrepareSerialization{
6    flatbuffers::FlatBufferBuilder builder;
7    foreach text file f storing WSPs do
8      wsp_vector.clear();
9      wsp_vector load all WSPs from file f;
10     sort(wsp_vector), order by code;
11     auto wsps = builder.CreateVectorOfStructs(wsp_vector);
12     auto oracle = CreateOracle(builder, wsps);
13     builder.Finish(oracle);
14     std::ofstream filew("f_oracle_flatbuffer.out");
15     filew.write(builder.GetBufferPointer(), builder.GetSize());
16     filew.close();
17 }
```

Algorithm 10: oracle_run.cpp for the querying stage

```
1  #include ¡thread¿
2  #include ¡sys/mman.h¿
3  #include "oracle_generated.h"
4  using namespace std;
5  namespace ofb = MyOracle::Sample;
6  const int N = number of binary files;
7  const flatbuffers::Vector¡const ofb::Wsp *¿ *wspsdata[N];
8  Void Process{
9    foreach binary FlatBuffers file f do
10     long long filesize = getFileSize(f);
11     int fd = open(f, O_RDONLY, 0);
12     void* mmappedData = mmap(NULL, filesize, PROT_READ,
       MAP_PRIVATE , fd, 0);
13     auto result = ofb::GetOracle(mmappedData);
14     wspsdata[f] = result→wsps();
15   Load all one-to-one distance queries;
16   Partition queries to m threads;
17   for i ← 0 to mthread do
18     thread[i] ← initial thread i, process distance queries using function
       GetDist(p1, p2) in Algorithm 11;
19   for i ← 0 to mthread do
20     thread[i].join();
21 }
```

Algorithm 11: GETDIST(lat_1, lng_1, lat_2, lng_2)

```
1  code ← Z_4^0( Z_2((lat_1, lng_1)), Z_2((lat_2, lng_2)) );
2  f ← which file would have the WSP of code;
3  left ← 0;
4  right ← wspsdata[f]→size();
5  while left ≤ right do
6      mid ← (left + right)/2;
7      if wspsdata[f]→Get(mid)→code() ≤ code then
8          left ← mid + 1;
9      else
10         right ← mid - 1;
11 return wspsdata[f]→Get(l − 1)→d();
```

We use FlatBuffers in $C++$. To generate our distance oracle $C++$ header called *oracle_generated.h*, we define a schema, say *Oracle.fbs* in Algorithm 8, and use the compiler (e.g. flatc -c Oracle.fbs) to generate *oracle_generated.h*. After that, we use Algorithm 9 to serialize each distance oracle text file to a binary Flatbuffers file.

In the querying stage illustrated in Algorithm 10, we use *mmap* from *sys/m-man.h* to virtually load all binary files in program, and *thread* library to enable multi-thread processing similar to CDO [53].

Recall that each distance oracle text file stores around 100 million WSPs. This is because Algorithm 9 needs to load all WSPs in memory for one text file and then serialize it to a binary file. Storing 100 million WSPs usually takes 7.4GB in a plain text file. After using FlatBuffer, each WSP is represented as 12 bytes, and the total size of the binary file on disk is 1.6GB. Note that it is not 12 bytes for each WSP because of the extra header. In this way, for the whole USA road network, the 4.6 billion WSPs are separated into 46 binary files, and each binary file occupies around 1.6GB on disk.

6.2.3 Querying and Applications

Algorithm 11 explains how to process the *basic* query that retrieves the network distance for a pair of vertices. Our previous work [52,54] illustrated how to process a batch of source-target queries, distance matrix query, and trajectory query based on the *basic* query and user-defined functions. In this section, we describe two common

industrial use-cases using our DOS. One is the delivery query, which is demoed in `http://sametnginx.umiacs.umd.edu/maryland` , and the other one is the KNN query. More examples are provided online in `http://roadsindb.com/`.

6.2.3.1 Delivery

As we discussed in Section 6.1, delivery services frequently need to compute a distance matrix and a relatively optimal route for each truck. A distance matrix is easy to compute by partitioning into a batch of *basic* query, and the relatively optimal route could be achieved by many approximate algorithms as it is an NP-complete problem. However, the reality is often more complex than the theory. In delivery services, there are two problems that must be considered: 1) Trucks take a lot of time to turn left or turn back, but are not allowed in some turns; 2) The delivery locations lie on road segments other than road vertices;

For the first problem, we construct a graph G' for G with the penalty of turns, then precompute the distance oracles for G'. In particular, if the road edge (a, b) has a big turn penalty to edge (b, c), then G' would add a virtual vertex b' for b, and vertex b can only go to vertex c through b'. The weight of edge (b, b') is equivalent to the turn penalty. Certainly, we need to add some edges to connect b' to other adjacent vertices.

For the second problem, we have an extra table for the information of delivery locations. Without loss of generality, each delivery location is represented as follows.

$$(id,\ lat,\ lon,\ edgeid,\ vid_1,\ vid_2,\ d_1,\ d_2) \tag{6.3}$$

vid_1 and vid_2 are the endpoints of the edge, and the d_1 and d_2 are the distances from the delivery location to vid_1 and vid_2 respectively. Note that, sometimes, one delivery location may have more than two nearby road vertices such as in an apartment with several entrances. In this case, we can extend the two vertices to a list of nearby vertices, termed $vid(\cdot)$. The delivery distance should be

$$dist(id_1, id_2) = \min_{a \in vid(id_1), b \in vid(id_2)} GETDIST(a, b) + d_a + d_b \tag{6.4}$$

6.2.3.2 KNN

In the KNN example, we have two location tables, named *University* and *Restaurant* that each record is $(id, Z_2\ code, lat, lon)$. The KNN task is to find the top K nearest restaurants for each university in terms of network distance. In order to avoid computing all pairs of *University* and *Restaurant*, we need to compute a candidate set of restaurants that have the potential to be the K nearest neighbors for each university. Thus, we decompose the task into two steps.

The first step is to compute the candidate restaurants for each university. In particular, for each university u, compute the K neighbors in the Euclidean distance using the GiST index or k-d tree, and then compute the maximum network distance $d_G(u)$ among the K neighbors using DOS. Now the candidate set of a university u is the restaurants what the Euclidean distance is within $d_G(u)$.

In the second step, obtain the IDs of all candidate restaurants using the GiST index or k-d tree for each university u whose Euclidean distance is less than or equal to $d_G(u)$, and then use DOS to compute their corresponding network distances and

retain the K closest ones.

This method is correct because the Euclidean distance is a lower bound on the network distance. The lower bound property guarantees that we find the KNN within the candidate set.

6.3 Experiments

From OpenStreetMap [11], we extracted the Bay Area road network with $758K$ vertices, the New York City road network with $407K$ vertices, and the whole USA road network with $2.4M$ vertices ignoring the vertices not bidirectionally connected to the main graph. In addition, we add the Florida road network from the 9^{th} DIMACS Implementation Challenge [3]. A demo is set up at `http://sametnginx.umiacs.umd.edu/`. Table 6.2 provides the characteristics of the road network datasets used in our evaluation.

Table 6.2: Dataset Characteristics in DOS

Name	NY	Bay	FL	US
Region	NYC	Bay Area	Florida	USA
# of Nodes	407,582	758,104	1,070,376	23,947,347
# of Arcs	977,106	1,663,662	2,712,798	58,333,344
# of WSPs with $\epsilon = 0.25$	84.6M	146M	199M	4.6B
# of WSPs with $\epsilon = 0.1$	431M	765M	929M	-

From our past experiments and previous theoretical work, we conclude that

the value of ϵ greatly influences the size of the distance oracles, but it does not have much of an influence on the querying time as the time complexity is just $O(\log \frac{n}{\epsilon^2})$. Thus, for the small road network, we set $\epsilon = 0.1$, but let $\epsilon = 0.25$ for the whole USA road network. We implemented four methods as follows for distance computations. All of them are implemented in C++, and are processed with multi-threads, and in the same environment consisting of a Macbook Pro 15-inch, 2.8 GHz Quad-core Intel Core i7, 16 GB memory.

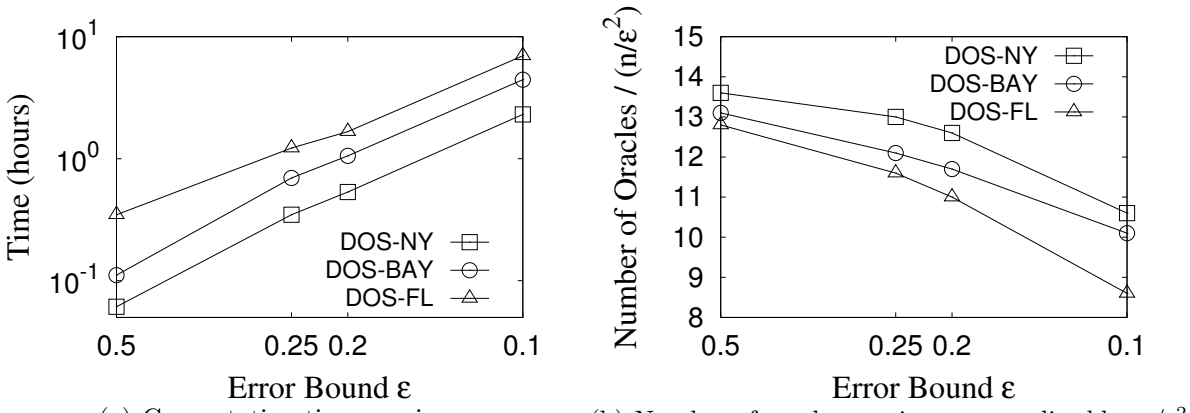
1. *DOS.* We implement our DOS framework with FlatBuffers. The representation of WSPs in FlatBuffers is in binary files on disk. Loading binary files through *mmap* is instant as DOS does not actually load binary files in memory, but still on disk. During querying, DOS caches the block containing visited WSPs to reduce further I/O cost.

2. *CDO.* We implement the CDO solution from [53] in memory. But it is only for small road networks. The time of preloading WSPs in memory is not counted in any querying time.

3. *DO.* We compare against the distance oracle *DO* method of [64] In this case, we load distance oracles as a relational table in PostgreSQL and index it using a B-tree.

4. *CH.* We implement the CH method proposed in [39] as a representative of methods that optimize the execution of single source shortest paths.

5. *HLDB.* We implement HLDB [20] in memory for small road networks same as



(a) Computation time varying ϵ

(b) Number of oracles varying ϵ, normalized by n/ϵ^2

Figure 6.2: Precomputation performance varying ϵ

in the CDO demo [53], but in PostgreSQL for the USA road network.

6.3.1 Precomputing $DO(G)$

First of all, we show the performance of precomputing the $DO(G)$ for different road networks and ϵ settings.

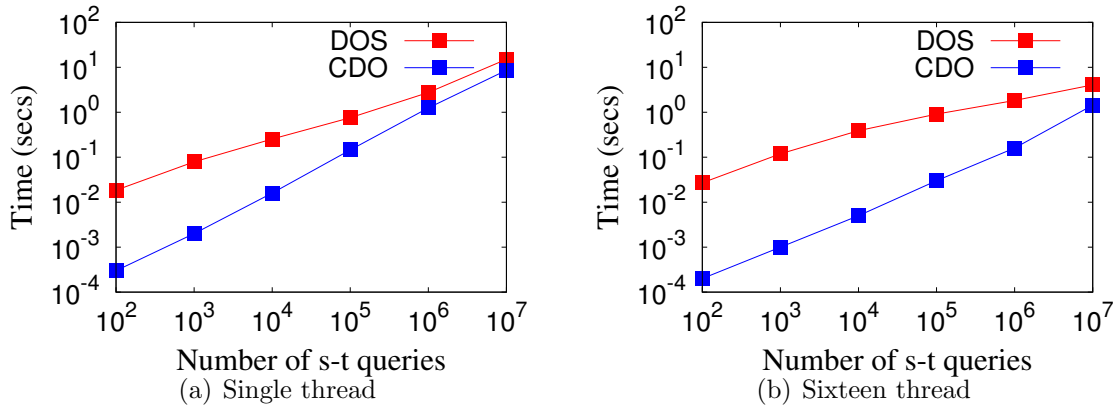
Figure 6.2(a) shows the time to compute the $DO(G)$ using a single machine. $DO(G)$ for the NY, BAY, and FL datasets can be computed in less than an hour for a fairly useful ϵ value of 0.25 and in a little over 7 hours for $\epsilon = 0.1$. Note that these are large datasets comprising road networks of states in US and being able to compute them within a few hours on a single machine means that computing the oracles is a practical proposition. Furthermore, we later show for the US dataset that by adding more machines to the computing infrastructure, we can significantly speed up this process. Next, Figure 6.2(b) plots the ratio of the number of oracles and $\frac{n}{\epsilon^2}$ versus ϵ. Here we let ϵ vary taking on the values 0.5, 0.25, 0.2, and 0.1. As all values are between $8-15$ in Figure 6.2(b), it confirms that the number of oracles

conforms to $C \cdot (\frac{n}{\epsilon^2})$ for the NY, BAY and FL datasets, where the value of C ranges between 8 and 15. Moreover, Figure 6.2(b) shows that C decreases as ϵ decreases, which is a good news for applications that require higher accuracy.

Table 6.3: Precomputation for $DO(G)$ on US Dataset

Performance	DOS-0.25
Cluster Size	Time
4	1.8 days
45	5.1 hours
Speed-up	8.47
Number of oracles	4.6 billion
Number of oracles / (n/ϵ^2)	11.9

Next we show scalability results in Table 6.3 for the $DO(G)$ of the US dataset with ϵ of 0.25. For this experiment, we used two clusters running on the distributed framework in Figure 2.5: an in-house cluster of 4 machines and an Amazon EC2 cluster with 45 *m4.xlarge* machines. From the table, it can be seen that precomputing $DO(G)$ can reduce the time needed from 1.8 days when using 4 machines to a little over 5 hours when using 45 machines. This constitutes a speed of 8.47. Note that by going from 4 machines to 45 machines, we have roughly scaled the computing cluster by a factor of 11. The speedup 8.47 we obtained is very close to 11 which indicates that our algorithm provides a linear speedup in the number of machines, which is a desirable property that one expects from parallel algorithms.



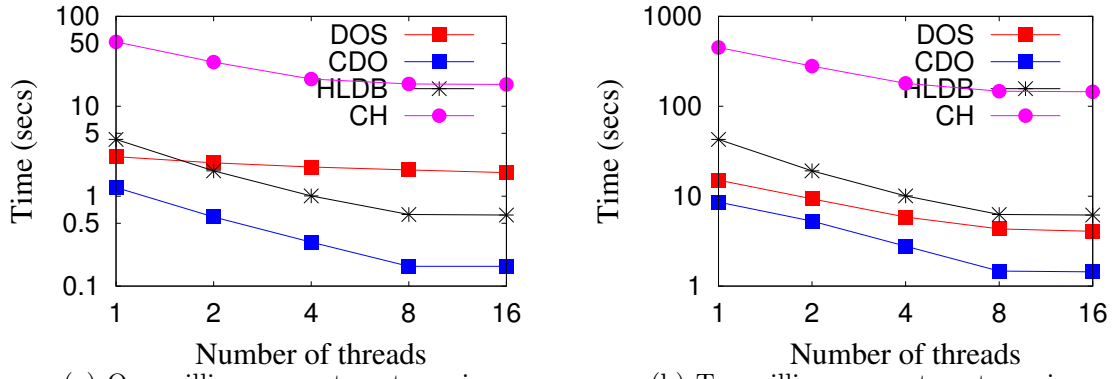
(a) Single thread (b) Sixteen thread

Figure 6.3: Time performance by varying the number of a batch of queries : (a) results for a single thread; (b) results for sixteen threads. As DOS is cold-start and caches WSPs during querying, the performance of DOS is close to CDO as the number of queries increases.

Moreover, the cluster compute and storage cost for precomputing the oracle is less than $50 based on AWS August 2017 prices. These results provide powerful support for our claim vis-a-vis the feasibility of our method since it is cheap to precompute and can be further sped up by simply adding more machines. In fact, if need be, the oracle can simply be recomputed rather quickly if there are large scale changes in the road network such as major road closures etc.

6.3.2 Querying Performance

There are two groups of comparison methods. One is DO and HLDB for large road networks, which their variant representations of the graph for distance computations are stored in a PostgreSQL database. The other group is CDO, CH, and HLDB for small road networks, for which their graph representations are preloaded



(a) One million source-target queries

(b) Ten million source-target queries

Figure 6.4: Time performance for processing a batch of source-target queries by varying the number of threads: (a) results for one million; (b) results for ten million source-target queries. Remember that only DOS is cold-start as it loads WSPs in the FlatBuffers format from disk. All of the other methods are after preloading the variant representation of the graph in memory, and the time of preloading is not counted in the query time.

in memory. Among this group, CDO and HLDB in memory are only available for small road networks, e.g., the number of vertices is less than one million, and CH in memory is applicable for large road networks as well.

Figures 6.3 and 6.4 show the time performance of DOS in a small road network, e.g., the NY road network, and compare it with CDO, CH, and HLDB in memory. The memory and cache were cleaned every time before running the code. Figure 6.3 shows the influence of the number of queries for DOS and CDO. Note that our DOS is an extension of CDO, and is available for large road networks as well. When the number of queries is small, DOS is much slower than CDO. This is because DOS needs to search WSPs from disk, but CDO does the binary search in memory

directly without taking tens of seconds to preload WSPs. As the number of queries gets larger, DOS cached more blocks containing visited WSPs, so that the query performance of DOS gets closer to CDO. Especially for the first $10M$ queries, DOS takes 4.07 seconds with 16 threads, and the throughput of DOS could be $2.45M$ distance computations per second. Moreover, the throughput of DOS for the next $10M$ queries increases to $5.01M$ per second. Thus, after the cold-start stage, DOS would be very close to CDO, e.g., $6.7M$ per second.

Figure 6.4 describes the time performance varying with the number of concurrent threads for DOS, CDO, HLDB, and CH. Figure 6.4(a) is under $1M$ distance computations, and Figure 6.4(b) is under $10M$ distance computations. From Figure 6.3, we know that DOS is under cold-start stage during the first $1M$ distance computations, e.g., in Figure 6.4(a). This is the reason that DOS is even slower than HLDB in Figure 6.4(a), and its time performance does not significantly decrease when using more threads. On the other hand, in Figure 6.4(b), DOS performs better than HLDB and gets closer to CDO as it passes the cold-start for last several million distance computations.

Figure 6.5 shows the time performance for the methods that are available for large road networks, e.g., the US. Note that our previous work CDO is not available here. To generate the source-target queries, as the source-target queries do not distribute uniformly in the whole domain, we randomly pick one road vertex and randomly select the other road vertex within 200km. From Figure 6.5, DOS performs much better than DO, HLDB, and CH, especially after the cold-start stage.

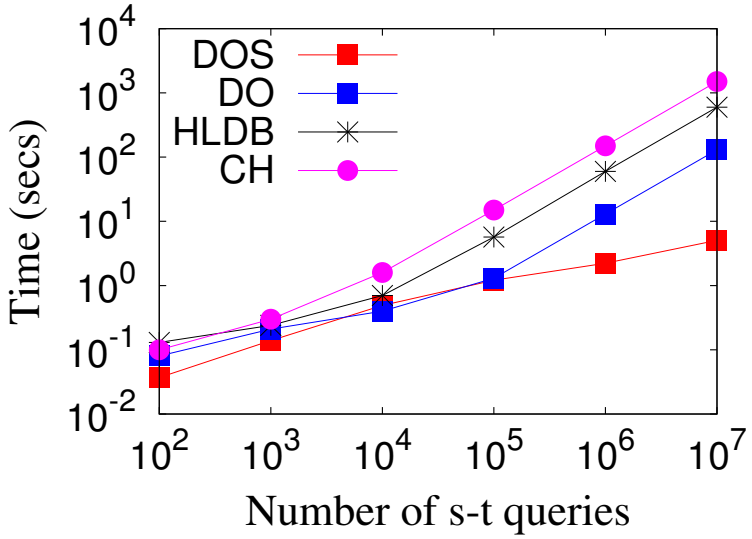


Figure 6.5: Time performance for DOS, DO, HLDB, and CH for the whole USA road network: DOS and CH is running with 16 threads, while DO and HLDB are running in PostgreSQL. In order to make the queries reasonable, each source-target query is generated by randomly picking one road vertex and the other road vertex within 200km.

6.3.3 Spatial Analytic Queries

DOS can efficiently process analytic queries as it can perform a large number of network distance queries. The querying performance of DOS in the following experiments is after the cold-start stage. Figure 6.6 compares the time performance of Dijkstra's algorithm, the CH algorithm, the HLDB in PostgreSQL, DO in PostgreSQL, and our DOS for a common spatial query, KNN, which is described in Section 6.2.3. One location list contains 6,070 locations of universities from [7], and another location list contains 49,573 locations of fast food restaurants from [5]. The format of each record in both lists is *(id, Z_2 code, latitude, longitude)*, where we

precomputed the Z_2 code.

The *KNN* query with which we experimented obtains the K nearest restaurants for each university in Figure 6.6. DOS first used a k-d tree index to retrieve a candidate set of restaurants that have the potential to be the K nearest neighbors for each university (or restaurant). Next, it computed the network distances for each university-restaurant pair (or restaurant-restaurant). Dijkstra's algorithm is implemented using a heap to speed it up. It starts at each university to search, and stops if the search for this university has visited K restaurants. The CH algorithm finds the restaurant candidate set using a k-d tree as well, then computes the distances between the pairs, and finally sorts the result to get the top K restaurants for each university. Both HLDB and DO used the GiST index in PostgreSQL to find the restaurant candidate set.

From Figure 6.6, we can see that DOS is much faster than Dijkstra's algorithm, CH, HLDB, and DO. Although Dijkstra's algorithm is considered efficient for the KNN query, it is not faster than DOS yet even when K is very small, e.g, 5.

In addition, here we provide an application that can be efficiently solved by DOS. It is to measure the accessibility of jobs, i.e., how many job opportunities exist nearby each census block. We use the LEHD dataset [10] to obtain the job locations around the Bay Area. This workload, shown in Figure 6.7, contains 120 million distance computations, where DOS takes 22 seconds for only the Bay Area road network, while CDO needs 18 seconds. In a general setting, i.e., DOS for the whole USA road network, this task can also be finished in 25 seconds. This is because of the spatial concentration property that make most WSPs in the Bay Area be usually

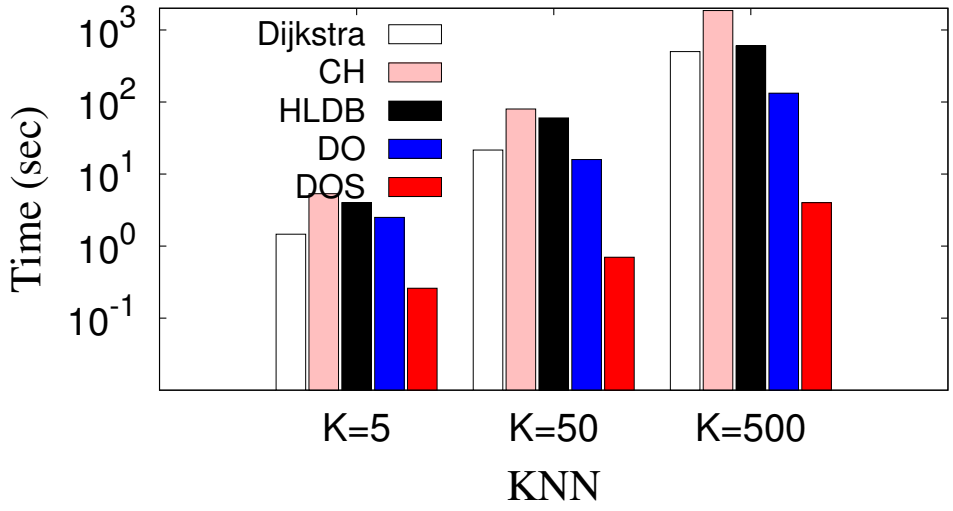


Figure 6.6: Response time for the KNN query including 6,070 university sources and 49,573 restaurant destinations.

in one or two FlatBuffers binary files. Then after the cold-start stage, these WSPs in the Bay Area would all be in memory, so that the hit rate of cache in DOS is much higher than random queries. It makes the performance of DOS more similar to CDO for the small road networks.

Obviously, using DOS, many analytic queries could be solved and visualized in a much quicker way. All applications implemented in our previous work [52–54] and in our blog site [3] could be set up in DOS as well to obtain better performance. Moreover, DOS could be also sped up by using multi-machines as it is easy to copy one set-up machine to many.

[3] `http://roadsindb.com/`

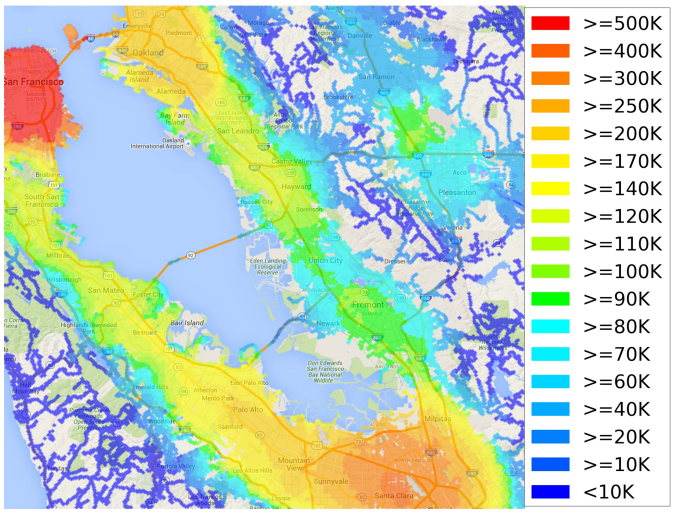


Figure 6.7: Nearby job opportunities (e.g., within 10 kms) for each census block in the Bay Area, requiring 120 million distance computations, which DOS finished in 22 seconds for just the Bay Area road network, and in 25 seconds for the whole USA road network.

6.4 Related Work

Figure 6.8 illustrates the problem domain in three dimensions: query time complexity, space complexity, and result accuracy. Reviewing previous research work, most focus on the trade-off between time complexity and space complexity with exact shortest distance/time results. However, most spatial analytic queries in industrial settings allow approximate results such as when using the Euclidean distance.

The state-of-the-art methods for computing shortest distances fall into two main categories: *latency* methods and *throughput* methods. However, there is no method that could achieve more than $1M$ throughput for general spatial road net-

works.

Latency approaches are designed to answer a single or a small number of shortest path or network distance queries on road networks. The original road network, or a processed representation of it, is stored in memory and queries perform operations on this in-memory representation. The most common latency approach is Dijkstra's algorithm [37]. Other methods [21,22,25,30,34,39,40,43,55,61,76] operate on the observation that some vertices in a spatial network are more important than others in answering shortest path queries. These methods offer different trade-offs between preprocessing time, storage, and query time. RE [40] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. HL [21] and m-hop [30] find hub nodes or distance labels such that the network distance between any two vertices can be computed by just checking their hub nodes or distance labels. DisLand [43] and LLS [55] find landmarks among network vertices to speed up network distance queries. [22, 25, 34, 39, 61, 76] build an explicit hierarchy graph to overcome the drawbacks of Dijkstra's algorithm.

Precomputing distance oracles requires knowledge of the network distances between some of the vertices in the road network. To do this, we use the CH method [39], which is one of the fastest available in-memory methods. CH also has a pre-processing stage where it computes an importance score for each vertex and then replaces some of the original edges by shortcuts. For a spatial network with n vertices, [39, 74] show that CH takes $O(n)$ additional space to store this auxiliary information. For the full USA road network data, we found that CH's pre-processing stage takes about one hour and generates 24.5 million shortcuts. The response time

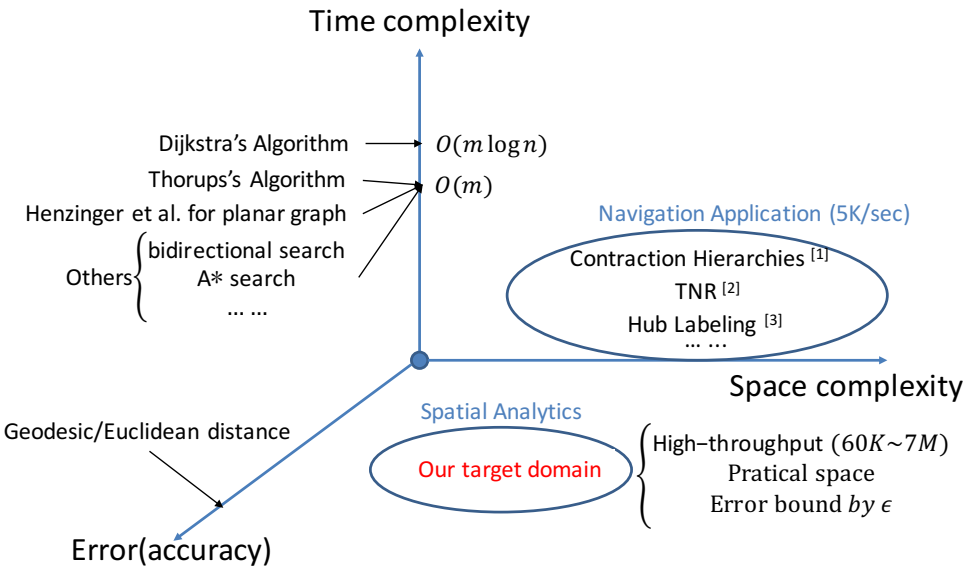


Figure 6.8: The target problem domain we focus on is spatial analytic queries. To achieve a high throughput performance and meaningful analytic results, it requires a trade-off among query time complexity, space complexity of storage, and result accuracy.

for a single path and distance query is in the 0.1–10 ms range.

Other latency methods such as [69, 72] take advantage of the spatial information associated with the vertices and edges of a road network and use geometric techniques. *Road Network Embedding* (RNE) [69] applies a Lipschitz embedding [42] to a road network, such that vertices of the spatial network become points in a high-dimensional vector space. In this method, all operations on the road network occur in the high-dimensional space. [72] takes advantage of the fact that the shortest paths from vertex u to all other vertices can be decomposed into subsets based on the first edges on the shortest paths from u to them. This property is referred to as *spatial coherence* in ϵ-DO [64] and is used by [72] to speed up Dijkstra's algorithm.

In particular, [72] stores a few geometric objects for each vertex in the road network that can prune searches during run-time. The spatial coherence of the destination vertices is also used to find nearest neighbors in a road network [60, 62].

A characteristic of *throughput* methods is that the shortest paths and distances are precomputed so that the query process only requires a lookup as opposed to any real computation on the fly. The resulting precomputed representation is large so that it is stored on disk thereby affecting latency. On the other hand, these methods are good for obtaining a high throughput since multiple lookups can be batched up at the same time thereby increasing the number of queries that can be answered at the same time, although each query may take a bit more time.

Among the throughput methods, [64, 65, 67] exploit the spatial coherence of both sources and destinations in the sense that if a set of vertices are sufficiently far away, then distances between pairs of points in different clusters are similar. The *Path-Coherent Pairs Decomposition* (PCPD) [67] gives one exact shortest path algorithm, while the ϵ-DO [64, 65] provides an approximate network distance with ϵ-error guarantees and $O(\frac{n}{\epsilon^2})$ space. [52] examines the task of computing spatial analytic queries and experimentally compares their performance using the ϵ-DO architecture, where query processing is completely handled by an RDBMS, and using a hybrid architecture, where there are separate modules for the database, the road network, and a query analysis tool. SPDO [54] is another method of using ϵ-DO inside a distributed key value store such as Apache Spark. Another database-centric method is HLDB [20] which can answer exact network distance queries and full shortest-path even for an area as large as Europe containing 18 million vertices

with complex SQL queries.

In HLDB [20], the authors mention that most of the memory-based latency approaches surveyed in [35] are difficult to embed into a database system and to query using SQL queries. This is because most methods rely on complicated data structures such as graphs and priority queues, which cannot be incorporated into a database system where the fundamental building blocks are relational operators. The main contribution of HLDB is the embedding of the memory-based hub labels (HL) [21] method into a database. The HL method precomputes the hub nodes for each vertex such that the distance between any two vertices s and t can be obtained given only their hub nodes. However, compared with the best previous database-centric oracle methods ϵ-DO [64, 65] and PCPD [67], HLDB has two drawbacks: 1) HLDB is not efficient if the average number of hub nodes per vertex is large; 2) Each spatial query in HLDB is a complex SQL statement that must perform a join operator on "forward" and "backward" tables, so that HLDB cannot guarantee query responses within a time bound.

Wu et al. [74] evaluate several state-of-the-art methods (i.e., [25, 39, 64, 67]) for computing road network distance in the same environment. Even though they do not make the distinction between latency and throughput methods, and only compare all the methods from a latency perspective, there are some valuable lessons to be learned from this work. Wu et al. [74] show that TNR [25] and CH [39] have fast preprocessing, low space overhead, support for real time queries, and the ability to easily handle continental road networks with tens of millions of vertices. This inspired our decision to use CH [39] for precomputing ϵ-DO. They also point out that

although ϵ-DO and PCPD are better for answering queries, they are not practical because they are too expensive to precompute. This chapter remedies this perceived deficiency of ϵ-DO and enables it to scale to handle continental road networks such as the entire USA.

6.5 Concluding Remarks

DOS is a practical system that utilizes all of our previous distance oracle techniques such as ϵ-DO [64, 65], PCPD [67], SPDO [54], and CDO [53] . DOS is the first system that achieves $5M$ distance computations per second per machine for general spatial analytic queries on any-sized road networks. Although the shortest distance result is approximate, it is bounded by ϵ. Our previous work has shown that $\epsilon = 0.25$ is enough for most use-cases. As DOS uses *mmap* to virtually load distance oracles from disk, it accepts larger sizes of distance oracles, or with smaller ϵ, e.g., $\epsilon = 0.1$ or 0.05, where the only limit is the disk storage. In addition, DOS has a cold-start stage for querying. It starts at a throughput of $10K$ per second without caching any WSP in memory, but achieves a throughput of $5M$ per second after ten seconds.

Future work includes incorporating traffic information and changes in the road networks such as road closures. This requires devising ways for computing the oracle in piecemeal-fashion so as to avoid doing it from scratch.

Chapter 7: Conclusion Remarks and Open Problems

7.1 Summary

In this thesis, we investigated in a set of implementations and applications of the distance oracle representation and spatial analytic queries. We defined the *throughput* metric to measure the performance of a spatial system for spatial queries. We first proposed the ASDO distributed architecture for computing the distance oracle representation to make it affordable for large road networks. Then in our detailed evaluation, using the distance oracle representation in a traditional database would obtain a much better throughput performance, e.g., $60K$ road distance operations per second per database server, for a variety of spatial analytic queries in common use such as KNN, distance matrix, and trajectory queries, comparing with a widely used hybrid architecture in industry. Next, SPDO enabled us to embed our distance oracle representation in a large distributed key-value clusters such as Apache Spark. It improved the throughput by at least two orders of magnitude, and made it easier for users to program spatial analytic queries for a mass of spatial data in a distributed storage. Finally, after communicated with tens of related spatial companies, we first implemented the CDO demo for a city-sized road network , and then extended it to DOS for general road networks. This system achieved 5 million

distance computations per second per commodity machine even on the whole USA road network.

Although the distance result is approximate, which is not accurate enough for navigation, its accuracy is far enough for spatial analytic queries. The benefits of these systems are: 1) saving a huge amount of processing time for spatial analytic queries, e.g., from one week to ten minutes; 2) significantly reducing the hardware cost of the computing center for spatial analysis; 3) making some spatial applications or queries possible to interact with in real time.

7.2 Open Problems

In Chapters 1–6, we have described our work in progress on high-throughput distance computations on large spatial networks and the subsequent applications. Finally, we discuss a few interesting related problems to our work on scalable query processing on spatial networks.

1. **High-throughput Solution for In-Path Queries**

   In-Path queries are another group of popular spatial analytic queries needing shortest path information instead of only distance information. It is not easy to obtain a high-throughput solution for them because in almost all state-of-the-art methods, the time needed to perform the shortest path retrieval is much longer than the time needed to perform the shortest distance retrieval. In particular, In-Path queries involve finding points from one set that lie on the shortest paths (allowing for short detours) between points from another

set. The problem formulation of the In-Path query involves two sets (a) A set of landmarks(e.g.,fast food restaurants); (b) A set of mobile users denoted by starting (sources) and ending (destinations) locations. Then, we have two basic In-Path queries:

- *Query 1*: Given a mobile user and a path with a source and destination, determine what landmarks lie on the shortest path between source and destination?
- *Query 2*: Given a landmark, determine which recent mobile users will pass through it on some shortest path between their corresponding sources and destinations?

In real applications such as local awareness advertisement, the following assumptions are acceptable: (a) The number of landmarks is much smaller than the number of mobile users; (b) The identities of mobile users are not known a priori and they arrive at a high rate; (c) The Landmarks are known a priori although new landmarks can be added and/or deleted from time to time.

Based on this formulation, we can devise some variant or extension settings which will make the problem more difficult. Also, we can propose some interesting analytic In-Path queries, which are open problems. We illustrate some variants of In-Path queries as follows.

- *Query 3*: Given a mobile user and a path with a source and destination, determine what landmarks are within r, e.g., 500 meters, for the shortest

path between source and destination? There are two options for the queries. One is for radius r to be fixed, or can vary. The other one is that the distance r can be the network distance or the geodesic distance.

- *Query 4*: Same as *Query 2*, but seek the users that pass through the circular region of the landmark with radius r meters.
- *Query 5*: Given k users and their paths, find the landmark through which the most users pass. This query takes place when finding sites for a bus stop or determining a route for an employee shuttle bus.
- *Query 6*: Given k users and their paths, partition the k users into the minimum S groups, where each group's users pass through the same landmark. This query can be used for ride-sharing applications, in which case at times the maximum size of each group is limited to 4.
- *Query 7*: Given the history paths and current landmark locations,where to build a new landmark that satisfies the largest number of uncovered users' paths. Alternatively, we can analyze which landmark can be removed as fewest users pass through it.

The brute-force approach to in-path queries usually requests n shortest path computations for n mobile users, which is way too much work, especially n is more than one million. One approximate solution is that we precompute the landmarks lying on each distance oracle such as index, which we term the *landmark-DO* index and denote as *INDEX-1*.

$$INDEX-1: \quad (code_A,\ code_B,\ \ d_\epsilon) \Rightarrow \{lm_1, lm_2, lm_3, ...\} \tag{7.1}$$

where lm_i are landmarks lying on the shortest path between representative vertices p_A and p_B. Next, build the inverted index denoted as *INDEX-2* on the path oracles that pass through each landmark such as:

$$INDEX-2: \quad landmark_i \Rightarrow \{oracle_1, oracle_2, oracle_3, ...\} \tag{7.2}$$

Using these indexes, for *Query 1*, given a source p_A and destination p_B, we find the corresponding distance oracle first $(code_{p_A}, code_{p_B}, d_\epsilon)$, and then use *INDEX-1* to retrieve the landmarks for the found distance oracle. Here we just need two lookups to retrieve the landmarks that are lying on the approximate shortest path from p_A to p_B. Also since we only have the approximate shortest path, we may miss some landmarks, or return some extra landmarks that are close to the shortest path but not exactly lying on the path.

For *Query 2*, we use *INDEX-2* to retrieve the distance oracle candidates that pass through or near the given landmarks. Obviously, we can compute one distance oracle for each user or path. Thus, we just check the distance oracle of each candidate's source and destination to see if it is in the distance oracle candidates of the landmark or not.

Queries 3-7 are more complex. For *Queries 3-4* with a fixed radius r, we can precompute and build the *INDEX-1* and *INDEX-2* while taking account the fixed radius r. In particular, the landmarks in *INDEX-1* are the landmarks within r distance of the shortest path. It is not easy to deal with an adjustable radius r. One possible way is to precompute *INDEX-1* with a well-designed radius set $R = \{r_1, r_2, r_3, ..., r_w\}$ such as $\{r_1 = 10m, r_2 = 100m, r_3 = 1km, r_4 =$

$10km\}$. When a client submits an adjustable radius $r = 400m$, we can reduce the query to $r_3 = 1km$ and find the landmarks, and then filter out the landmarks that are outside of $r = 400m$. Moreover, *Queries 5-7* can be solved by leveraging *Query 1* to quickly retrieve the landmarks of each path. Next, *Query 5* can be solved by counting, *Query 6* can be reduced to a matching problem, and *Query 7* can be solved by enumerating the candidate landmarks and computing the cost function.

2. **Dynamic Updates for Time Oracles**

Many GIS applications care about travel time more than travel distance. Changing distance weights to time weights for all edges is not enough for practical time oracles as the time weights are updated in a high rate due to factors such as road closures, traffic obstruction times, accidents, etc. A straightforward solution for time oracles is to generate a time oracle for each time period, e.g., an hour, based on historical traffic data. It is enough for most analytic queries. Another solution for time oracles is to allow efficient updates of vertices and edges on ASDO. This requires devising ways of computing the oracle in piecemeal-fashion so as to avoid doing it from scratch as we currently do. A similar problem has been proposed recently [51], but it only considers the Euclidean distance metric. To deal with updates in ASDO, an important observation is that the significant influence of one update of an edge is limited in a local region rather than the whole road network. For example, the travel time from Washington, D.C. to New York City would be slightly

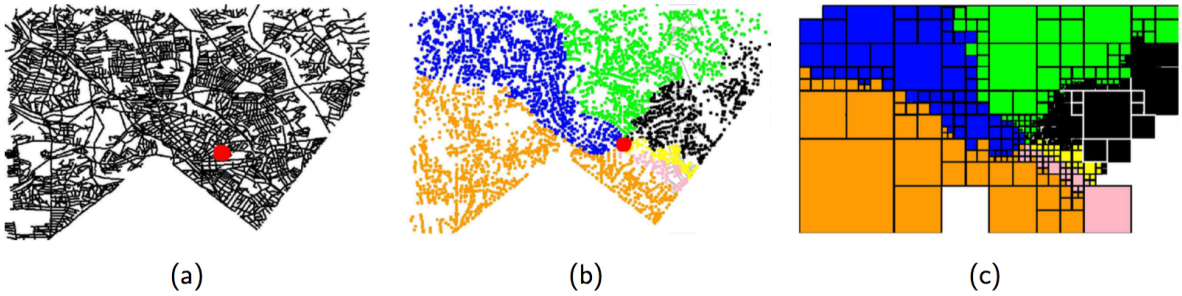


Figure 7.1: Example illustrates the coloring process of vertices for Silver Spring, MD. a) Sample vertex u having six outgoing vertices, b) The remaining vertices are assigned colors based on their shortest path to u through one of the six adjacent vertices of u. c) Morton blocks in a PR-Quadtree corresponding to the colored regions in (b).

changed by an accident near to Philadelphia. Ignoring these slightly changes is acceptable in almost all spatial analytic applications. Addressing this problem would make the underlying spatial database systems more powerful.

3. **SILC-Oracle: Auto Adjusting from ASDO to SILC for a Unified Framework**

   SILC [60, 62] is an efficient and unified framework to store a spatial network with n vertices in $O(n\sqrt{n})$ space for answering shortest path and distance queries, but requiring precomputing n^2 distance pairs to build the individual shortest path quadtrees. A possible distance oracle of size $O(\frac{n \log n}{\epsilon^2})$ *one-to-many* well-separated pairs (OM-WSP) is proposed in [64] that can approximately answer shortest distance queries within an error bound ϵ. These two frameworks have a similar external structure but evolved from different the-

oretical work. We propose a way to bridge the gap between the $O(\frac{n \log n}{\epsilon^2})$ distance oracle and SILC.

Figure 7.1 [62] introduces the main idea of SILC. For each vertex u, SILC generates $O(\sqrt{n})$ quadtree blocks that the first edges of the shortest paths from u to all vertices in the same quadtree block are the same. It is not straightforward to generate the SILC framework from $O(n \log n)$ distance oracles, although they have similar *one-to-many* structure. First, we can use the similar algorithm and distributed framework in ASDO to produce $O(n \log n)$ distance oracles with an ϵ error bound. For each distance oracle $(u,\ code,\ d_\epsilon)$, we know the shortest distance from u to any vertex in the block with Morton code *code* is bounded by $d_\epsilon) \cdot (1 + \epsilon)$. Next, two attributes, the first edge (FE) and the representative vertex (RV), are added to the schema of $O(n \log n)$ distance oracles. The schema looks the same as the SILC except for the additional RV attribute. Note that the FE is the first edge in the shortest path from u to the corresponding RV. So the schema of the new $O(n \log n)$ distance oracles is

$$(vertex\ id,\quad code,\quad d_\epsilon,\quad FE,\quad RV) \tag{7.3}$$

This schema can answer all the distance queries or analytic queries same as ASDO. But one main challenge exists which is how to use this structure to answer path queries for navigation? Figure 7.2 shows a general situation for a path query from s_1 to t. A is the block containing t in the distance oracle, p_a is the RV, and the FE is (s_1, s_2). In order to generate a path, we get the

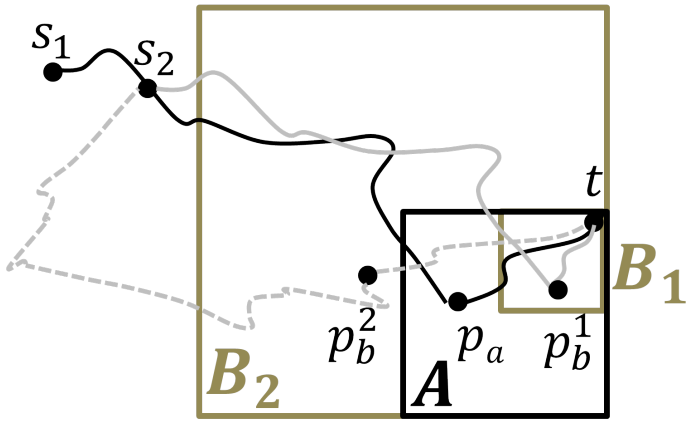


Figure 7.2: The challenge we meet. The query is the shortest path from s_1 to t. Starting at s_1, we know the representative vertex for the block A containing t is p_a and the next vertex in the shortest path from s_1 to p_a is s_2. Then at s_2, the block B containing t may be p_b^1 or p_b^2.

path from s_1 to p_a, then from p_a to t. In this case, the path is ϵ-approximate.

However, even for an ϵ-approximate path, it is not easy to get. Starting at s_1, we know the next vertex is s_2. After reaching s_2, the new distance oracle retrieved has three cases in Figure 7.2: 1) the original block A; 2) the smaller block B_1; 3) the larger block B_2. The first case is the easiest one. We just continue the retrieval process.

As we are closing to t, the second case most likely occurs. In this case, the representative vertex p_b^1 is different from p_a. Then we do not know how to reach p_a any more at s_2 since the FE is the edge for the shortest path from s_2 to p_b^1, not from s_2 to p_a. The third case rarely occurs, but it can happen. This case is caused by the shortest distance from s_2 to p_b^2 becoming larger than the

distance from s_1 to p_b^2. So this would make the following criteria become true.

$$\frac{r_{b_2}}{d_G(s_2, p_b^2)} \leq \epsilon \tag{7.4}$$

Both the second and the third cases propose the *representative vertex jumping* problem. If we continue to go the FE of s_2, it is possible that the FE of s_2 is (s_2, s_1). Then it causes a infinite-loop.

The good news here is that when this event happens, we know that the distance oracles of s_1 or s_2 can be have split further to approximate to SILC. SILC has the property that all vertices in A have the same first edge starting at s_1. For example for the second case, if we hold the SILC property for both (s_1, A) and (s_2, B_1), we do not need to care about the representative vertex jumping since no matter how the representative jumps, the first edge we selected is equivalent to the first edge from $s_1(s_2)$ to t. So that the infinite-loop problem would not happen as well.

An additional question is that obviously we can obtain a more accurate path than the ϵ-approximate path. In the previous strategy, we will reach p_a first, then from p_a to t. If t is at the boundary of the block A, the retrieved path almost touches the ϵ bound. A smarter strategy is that when we get closer to t, we automatically reach the better representative vertex in a smaller block instead of p_a. For example in the second case, at s_2, it is better for us to go to t through p_b^1 directly than through p_a.

In summary, we list the challenges here to automatically adjust ASDO to SILC.

(a) How to answer path queries, even for approximate path result, avoiding infinite-loop?

(b) How to detect which blocks need to be split further to achieve SILC?

(c) How to improve accuracy?

Bibliography

[1] Cabspotting. http://cabspotting.org/.

[2] CRAWDAD. http://crawdad.cs.dartmouth.edu/~crawdad/epfl/mobility/.

[3] DIMACS. http://www.dis.uniroma1.it/challenge9.

[4] ESRI. http://www.esri.com/.

[5] Fast food maps. http://www.fastfoodmaps.com/.

[6] Flatbuffers. https://google.github.io/flatbuffers/.

[7] GeoNames. http://www.geonames.org/.

[8] Google Maps API. https://developers.google.com/maps/.

[9] IndexedRDD. https://github.com/amplab/spark-indexedrdd/.

[10] LODES. http://lehd.ces.census.gov/data/.

[11] OpenStreetMap. http://www.openstreetmap.org/.

[12] PART. https://github.com/ankurdave/part/.

[13] pgRouting. http://pgrouting.org/.

[14] PostgreSQL. https://wiki.postgresql.org/wiki/FAQ/.

[15] Protocol Buffers. https://github.com/google/protobuf/.

[16] Redis. http://redis.io/.

[17] SQL Examples of Distance Oracles. http://roadsindb.com/.

[18] SSTI. http://www.ssti.us/events/.

[19] TAREEG. http://tareeg.org/.

[20] I. Abraham, D. Delling, A. Fiat, A. Goldberg, and R. Werneck. HLDB: Location-based services in databases. In *ACM GIS*, pages 339–348, Redondo Beach, CA, Nov. 2012.

[21] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, Kolimpari Chania, Greece, May 2011.

[22] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, Ljubljana, Slovenia, Sep 2012.

[23] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient algorithms and regular data structures for dilation, location and proximity problems. *Algorithmica*, 30(2):164–187, 2001.

[24] C.-H. Ang, H. Samet, and C. A. Shaffer. A new region expansion for quadtrees. *TPAMI*, 12(7):682–686, July 1990.

[25] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, pages 46–59, New Orleans, LA, Jan 2007.

[26] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDBJ*, 8(2):101–119, 1999.

[27] F. Brabec and H. Samet. Client-based spatial browsing on the world wide web. *IEEE Internet Computing*, 11(1):52–59, Jan/Feb 2007.

[28] P. B. Callahan. *Dealing with Higher Dimensions: The Well-separated Pair Decomposition and its Applications*. PhD thesis, The Johns Hopkins University, Baltimore, MD, Sep 1995.

[29] D. Carstoiu, E. Lepadatu, and M. Gaspar. HBase: Non-SQL database performances evaluation. *IJACT*, 2(5):42–52, 2010.

[30] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB J.*, 21(6):869–888, 2012.

[31] H. Cho and C. Chung. An efficient and scalable approach to CNN queries in a road network. In *PVLDB*, pages 865–876, Trondheim, Norway, Aug 2005.

[32] H. Cho, S. J. Kwon, and T. Chung. ALPS: an efficient algorithm for top-k spatial preference search in road networks. *KAIS*, 42(3):599–631, Mar 2015.

[33] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *IPDPS*, pages 1–8, Long Beach, CA, Mar 2007.

[34] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *SEA*, pages 376–387, Kolimpari Chania, Greece, May 2011.

[35] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. 2009.

[36] D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. *TKDE*, 27(3):686–698, Mar 2015.

[37] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[38] C. Esperança and H. Samet. Experience with SAND/Tcl: a scripting tool for spatial databases. *JVLC*, 13(2):229–255, Apr. 2002.

[39] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, Cape Cod, MA, May 2008.

[40] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, Miami, FL, Jan 2006.

[41] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing many-to-many shortest paths using highway hierarchies. In *ALENEX*, New Orleans, LA, Jan 2007.

[42] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995.

[43] S. Ma, K. Feng, H. Wang, J. Li, and J. Huai. Distance landmarks revisited for road graphs. *CoRR*, abs/1401.2690, 2014.

[44] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, pages 393–404, Venice, Italy, Aug 1998.

[45] A. Mortensen, D. Kostelec, B. Turley, and A. Parast. Evaluating connectivity projects: Using point-to-point gis routing to measure the benefits of new transportation connections. In *Transportation Research Board 90th Annual Meeting*, number 11-2288, 2011.

[46] S. Nutanong, E. H. Jacox, and H. Samet. An incremental Hausdorff distance calculation algorithm. *PVLDB*, 4(8):506–517, Aug 2011.

[47] S. Nutanong and H. Samet. Memory-efficient algorithms for spatial network queries. In *ICDE*, pages 649–660, Brisbane, Australia, Apr 2013.

[48] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX*, pages 183–191, Monterey, CA, June 1999.

[49] Oracle Corporation. Oracle spatial and graph network data model white paper. Technical report, Mar 2015.

[50] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, Berlin, Germany, Sep 2003.

[51] E. Park and D. M. Mount. Output-sensitive well-separated pair decompositions for dynamic point sets. In *ACM GIS*, pages 344–353, Orlando, FL, Nov. 2013.

[52] S. Peng and H. Samet. Analytical queries on road networks: An experimental evaluation of two system architectures. In *ACM GIS*, pages 1:1–1:10, Seattle, WA, Nov 2015.

[53] S. Peng and H. Samet. CDO: Extremely high-throughput road distance computations on city road networks. In *ACM GIS*, pages 84:1–84:4, Burlingame, CA, Nov 2016.

[54] S. Peng, J. Sankaranarayanan, and H. Samet. SPDO: High-throughput road distance computations on spark using distance oracles. In *ICDE*, pages 1239–1250, Helsinki, Finland, May 2016.

[55] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *TKDE*, 26(1):55–68, 2014.

[56] H. Samet. Distance transform for images represented by quadtrees. *IEEE TPAMI*, 4(3):298–303, May 1982.

[57] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2006.

[58] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *Commun. ACM*, 46(1):63–66, Jan. 2003.

[59] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, Nov 1984.

[60] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, Vancouver, Canada, June 2008.

[61] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA*, pages 804–816, Zurich, Switzerland, Sep 2006.

[62] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *ACM GIS*, pages 200–209, Bremen, Germany, Nov. 2005.

[63] J. Sankaranarayanan, H. Alborzi, and H. Samet. Distance join queries on spatial networks. In *ACM GIS*, pages 211–218, Arlington, VA, Nov 2006.

[64] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, Shanghai, China, Apr. 2009.

[65] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *TKDE*, 22(8):1158–1175, Aug 2010.

[66] J. Sankaranarayanan and H. Samet. Roads belong in databases. *Data Engineering Bulletin*, 33(2):4–11, Jun 2010.

[67] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[68] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, Apr–Jun 1990.

[69] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, Sep 2003.

[70] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, Trondheim, Norway, Aug 2005.

[71] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *ICDE*, pages 254–255, Tokyo, Japan, Apr 2005.

[72] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA*, LNCN 2832, pages 776–787, Budapest, Hungary, Sep 2003.

[73] S. Wasserman and K. Faust. *Social network analysis: methods and applications*. Cambridge University Press, 1994.

[74] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.

[75] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, Boston, MA, Jun 2010.

[76] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *SIGMOD*, pages 857–868, New York, NY, Jun 2013.