DOS: A Spatial System Offering Extremely High-Throughput Road Distance Computations [*]

Shangfu Peng
UMIACS, Department of Computer Science, University of Maryland
College Park, Maryland
shangfu@cs.umd.edu

Jagan Sankaranarayanan[†]
UMIACS, Department of Computer Science, University of Maryland
College Park, Maryland
jagan@umiacs.umd.edu

Hanan Samet
UMIACS, Department of Computer Science, University of Maryland
College Park, Maryland
hjs@cs.umd.edu

ABSTRACT

Large analytic applications on road networks including simulations, logistics, location-based advertisement, and transportation planning require shortest distance/time methods that provide high throughput (i.e., distance/time computations per second). Our previous work discussed how to process graph distance computations in a PostgreSQL database on a large 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 large 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 paper, we propose a new distance oracle system (DOS) for large 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. For example, a $10K \times 10K$ origin-distance (OD) matrix can be computed in 20 seconds.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**; **Database query processing**;

KEYWORDS

high-throughput, spatial analytic query, distance oracle

ACM Reference Format:
Shangfu Peng, Jagan Sankaranarayanan, and Hanan Samet. 2018. DOS: A Spatial System Offering Extremely High-Throughput Road Distance Computations . In *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18), November 6–9, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274895.3274898

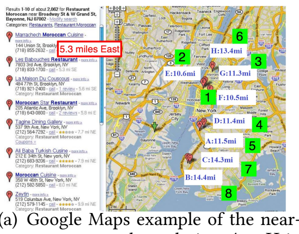
[*]This work was supported in part by the AWS research grant and the NSF under Grants IIS-13-20791 and IIS-18-16889.
[†]Now at Google Inc

1 INTRODUCTION

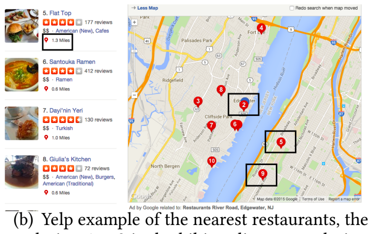
Beyond simple navigation queries, location-based web services like Google Maps repeatedly pose queries on a road network and utilize the results to serve a user base, e.g., the Google Distance Matrix product. Other examples include complex scenarios such as how to assign and deliver $10,000$ packages 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 [34]. We use the term *spatial analytic queries* to collectively describe such queries. The challenge lies that each such instance of a spatial analytic query invariably involves being able to make hundreds to as many as millions of computations of the shortest distance along a spatial network rather than as the crow flies or the Hausdorff distance (e.g., [26]).

In the face of a massive amount of spatial analytic queries from internet scale users, for example, Google Maps [5] drastically restricts the number of shortest distance results per query (e.g., a limit of 625 (25×25 O-D 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 the Euclidean distance instead of the network distance. Figure 1 illustrates the drawback of using the Euclidean distance in Google Maps and Yelp. Clearly, using the 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 2, termed the *route directness spectrum* (RDS) [29]. It shows the distortion in approximating network distance with geodesic distance. In particular, a query that is of immense interest to transportation planners is a measure called *route directness index* (RDI) [9]. The RDI of any two locations in the road network is the ratio between the shortest network distance to the geodesic distance.

Figure 2 shows the route directness spectrum of New York City (NYC), the Bay Area (Bay), and Salt Lake City (SLC) road networks, respectively, and 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 is what we expect as we know that most streets in NYC are laid out on a grid. 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. Considering the ordering examples in Figure 1 and the results of the route directness spectrum in Figure 2, we conclude that more



(a) Google Maps example of the nearest restaurants, 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 blue values) ordering we computed.



(b) Yelp example of the nearest restaurants, 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.

Figure 1: Both Google Maps and Yelp uses 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; (b) Yelp results for the query: find the restaurants around River Road, Edgewater, NJ within a 2 mile biking distance.

accurate network distances are well worth computing in spatial analytic applications.

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, CNN, and distance matrix. 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 [12], TNR [14], CH [22], 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 [27].

Our focus here is on *throughput* which is how to compute a spatial analytic query as quickly as possible. This proposed work is an extension of our demo work in CDO [28], and provides more details on the end-to-end system, precomputation process, and querying stack for very large road networks. This paper differs from our prior work as follow: (1) The first attempt is our fundamental theory work ϵ-distance oracle (ϵ-DO) [35] and PCPD [38] methods. These two papers introduce the distance oracle, but is not scalable to road networks with more than $80,000$ vertices. (2) Our previous work SPDO [29] 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. (3) Our previous demo work CDO [28] claimed that the reaction of some companies, 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. CDO 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 is only true for the mentioned specific companies, not for general applications.

In this work, we propose a system called DOS (denoting Distance Oracle System) for spatial analytic queries on large road networks. Its throughput achieves $5M$ distance computations per second, which is very close to the performance of CDO [28] on a city-sized road network, but is also available for large road networks. Contributions of DOS are:

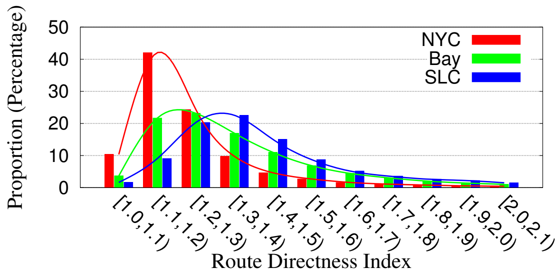


Figure 2: Route directness spectrum (RDS) of New York City (NYC), the Bay Area (Bay), and Salt Lake City (SLC). In contrast, the maximum RDI, corresponding to the maximum ratio of network distance to geodesic distance, is 10.6 **in NYC,** 30.4 **in Bay, and** 26.3 **in SLC; the average RDI is** 1.213 **in NYC,** 1.384 **in Bay Area, abd** 1.475 **in SLC.**

(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 [35] with multi-threads and query optimization illustrated in [27]. 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 [3] to serialize distance oracles to binary files. It reduces the storage space a lot. 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 [28], DO [35], HLDB [11], and CH [22].

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

2 RELATED WORK

Figure 3 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 industry 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 networks.

[1]http://sametnginx.umiacs.umd.edu/
[2]http://roadsindb.com/

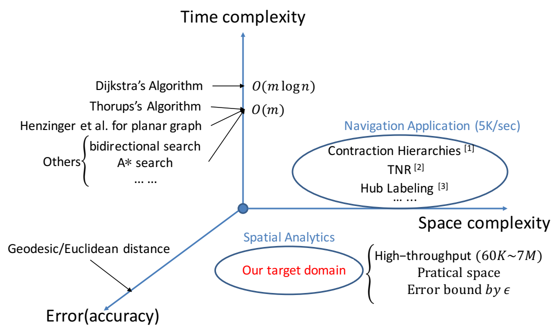


Figure 3: The target problem domain we focus on is spatial analytic queries. To achieve a high throughput performance and meaningful analytic results, it requires a trade-off among query time complexity, space complexity of storage, and result accuracy.

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 [20]. Other methods [12–14, 17, 18, 22, 23, 25, 30, 33, 43] 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.

Other latency methods such as [39, 41] take advantage of the spatial information associated with the vertices and edges of a road network and use geometric techniques. *Road Network Embedding* (RNE) [39] applies a Lipschitz embedding [24] to a road network, such that vertices of the spatial network become points in a high-dimensional vector space.

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. Among the throughput methods, [35–38] 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. Details would be explained in Section 3.2. SPDO [29] is another method of using ϵ-DO inside a distributed key value store, e.g., Apache Spark. Another database-centric method is HLDB [11] 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 [11], the authors mention that most of the memory-based latency approaches surveyed in [19] are difficult to embed into a database system and to query using SQL queries because of complicated data structures, e.g., graphs and priority queues. The main contribution of HLDB is the embedding of the memory-based hub labels (HL) [12] method into a database. HL precomputes the hub nodes for each vertex such that the distance between any two vertices can be obtained given only their hub nodes.

Wu et al. [42] evaluate several state-of-the-art methods (i.e., [14, 22, 35, 38]) for computing road network distance in the same environment. Wu et al. [42] show that TNR [14] and CH [22] 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 [22] 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 paper remedies this perceived deficiency of ϵ-DO and enables it to scale to handle continental road networks such as the entire USA.

3 PRELIMINARIES

3.1 Notations and Morton code

Table 1: Notation Summary

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
$Z_2(p)$	Morton code for a quadtree node
$Z_4(p_1, p_2)$	4D Morton code for a pair of quadtree nodes
$d_G(s,t)$	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

To make the discussion more general, we introduce some basic concepts about *spatial network* and *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 proposal, all discussion is under a 2-dimensional space S, but note that it is straightforward to extend our results to d-dimensional space. 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.

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

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.

We use the Morton (Z) order space-filling curve [31] that provides a mapping, $\mathbb{Z}^2 \to \mathbb{Z}$, 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 level. Each unit block i is referenced by a unique Morton code $mc(i)$. Figure 4(a) shows how a Morton order of quadtree blocks in a two dimensional space with $L = 2$. A spatial graph $G(V, E, w, p)$ on the domain S can also be divided into $2^L \times 2^L$ unit blocks. Given vertex v in the unit block i, v is assigned a Morton

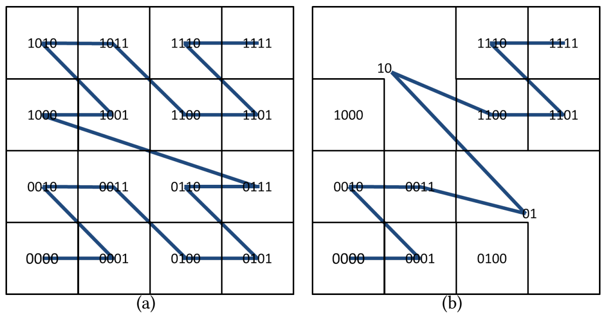


Figure 4: (a) Morton code and ordering in a 4×4 space. (b) Example illustrating key representation of distance oracle.

code $mc(v)$ as $mc(i)$,. All vertices located in the same block have the same Morton code. Besides the unit blocks, every 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 top left quadrant (1000, 1001, 1010, 1011) is 10. In this paper, 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 $mc(A)$. Once the data are sorted using this ordering, the resulting blocks can be stored using any one-dimensional data structure such as, but not limited to, a B-tree.

3.2 Distance Oracles

The ϵ-DO [35, 36] is 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, ϵ-DO [35, 36] describes a well-separated pair decomposition (WSPD) [15, 16] of a road network in order to produce well-separated pairs (e.g., (A, B) with particular network distance properties). Two sets of vertices A and B are said to be well-separated as in Figure 5 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), which satisfies the property that for any pair of vertices (s, t), $s \in A$ and $t \in B$, we can find the approximate distance $d_\epsilon(A, B)$, where $\epsilon = \frac{2}{s}$, providing an approximate network distance such that it satisfies the condition

$$(1 - \epsilon) \cdot d_\epsilon(A, B) \leq d_G(s, t) \leq (1 + \epsilon) \cdot d_\epsilon(A, B) \qquad (1)$$

As a result, ϵ-DO generates $O(\frac{n}{\epsilon^2})$ well-separated pairs, denoted as $(A, B, d_\epsilon(A, B))$. We denote the set of all WSPs as the distance oracle representation, i.e., $DO(G)$. Both A and B are a pair of PR quadtree blocks [31] at the same depth. In order to make a well-separated pair easy to embed in a database as a key-value pair, ϵ-DO uses the Morton (Z) order space-filling curve [31] to map a quadtree block in a 2-dimensional embedding space to a positive number.

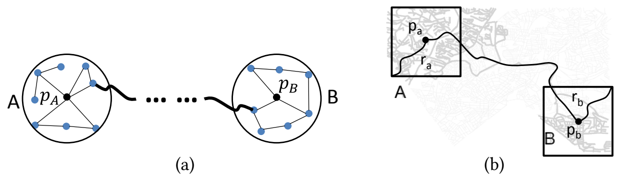


Figure 5: A well-separated pair example: (a) A theoretical WSP example. (b) A potential oracle containing blocks A and B in Silver Spring, MD showing representative vertices p_a, p_b and radii r_a and r_b.

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 $mc(mc(A), mc(B))$.

4 SYSTEM

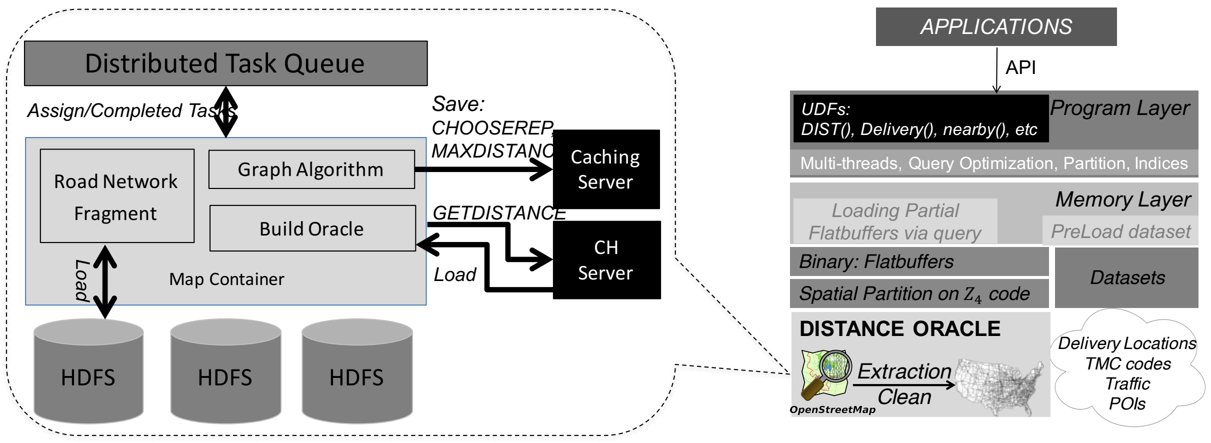
DOS includes two stages, precomputing and querying. The first stage is precomputing the $DO(G)$, and the other one is processing spatial analytic queries utilizing $DO(G)$. In this section, we first show the main picture of DOS, then explain how we speed up precomputing the $DO(G)$, and finally discuss each component of the querying stage of DOS, and how we process some spatial queries.

4.1 DOS framework

Our previous work [27–29] discussed how to utilize $DO(G)$ in different queries and settings. In SPDO [29], we proposed a general distributed framework to achieve a high throughput relying on the distance oracle of the whole USA road network. In contrast, the CDO demo paper [28] claimed that such applications with the *spatial concentration property* request a extremely high-throughput solution such as CDO, which optimizes our distance oracle technique on a small road network (city-sized road network) and 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, for large road networks, not limited to a city. In the preparation stage, which would be described in Section 4.2 in detail, we first extract the road network for any given region such as the whole USA road network from OpenStreetMap [7] and TAREEG [10], and then precompute the $DO(G)$. The distance oracle result is ordered and partitioned by 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 [29], 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 [3] (explained in detail in Section 4.4). In this way, given a pair of two location (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 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 detail of using Flatbuffers is explained in 4.4.

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,



(a) Distributed architecture for precomputing $DO(G)$ **(b) The DOS framework**

Figure 6: The DOS framework: (a) Introduce the distributed architecture for precomputing $DO(G)$, which is also the distance oracle workflow in (b); (b) Introduce the whole DOS framework that processes spatial analytic queries utilizing $DO(G)$.

we formalize queries as hundreds to millions one-to-one distance queries [27]. Each one-to-one distance query could be answered by binary search in time $O(\log \frac{n}{\epsilon^2})$ [29], 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 5 proved 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.

4.2 Task Partition and Parallelism

ϵ-DO [35] describes how we precompute $DO(G)$ for a city-sized road network. We first build a PR quadtree [31] on V based on the spatial position of the vertices. Then the $DO(G)$ construction algorithm is a top-down approach that starts with the WSP decomposition [15] of the block pair (S, S), which is the largest *potential oracle*. Note that S is the "whole spatial domain", which is defined in Section 3. A potential oracle is a pair of blocks that has 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 4(b), where "01" denotes the bottom-right 2×2 unit blocks, "10" denotes the upper-left 2×2 unit blocks. Starting at (S, S) as the first entry, a queue Q holds all current potential oracles. The algorithm pops a potential oracle (A, B) from the head of the queue. Figure 5(b) 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)* [35], 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 $DO(G)$. 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. From ϵ-DO [35], CheckOracle*(A,B)* includes three components: choose representative vertex p_A of a quadtree block A by invoking ChooseRep(A), compute the radius of the block using MaxDistance(A), and obtain the network distance d between p_A and p_B (i.e., $d = d_G(p_A, p_B)$) denoted as GetDistance(p_A, p_B) .

However, as the size of the road network becomes larger, we need a distributed architecture to compute $DO(G)$ since it takes a long time with a single machine. The quadtree structure that we use to represent the oracles lends itself to partitioning the workload. We can observe that the task of examining each potential oracle is essentially a data independent task. Based on this observation, we design and implement a distributed architecture for $DO(G)$ precomputation showed in Figure 6(a).

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 [22] (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 stage 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 use later. 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 7, we decompose the road network into 16 quadtree blocks in the Morton order such that

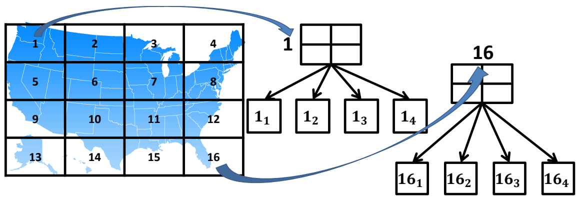


Figure 7: Initial decomposition of the US dataset to partition the task of precomputing $DO(G)$

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 or apply the graph center algorithm to compute the center. Once we have obtained the representative, we compute the radius by applying Dijkstra's algorithm. We save the representative vertex for each quadtree block and its radius in the caching server, then subdivide the quadtree block and continue processing until we reach the leaf blocks. These algorithms are implemented in the graph algorithm module in Figure 6(a). Now, for every block in the quadtree, we have stored a representative vertex and its radius.

In the second stage, 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 7, 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 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. Then the checking process invokes 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, $DO(G)$ is ready and can be loaded into DOS later.

4.3 Binary Search Querying

Our previous demo work CDO [28] illustrated an efficient way to obtain the network distance from distance oracles, given a source location $p_1 = (lat_1, lng_1)$ and a destination location $p_2 = (lat_2, lng_2)$. Once $DO(G)$ has been computed, CDO loads all well-separated pairs, for 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 [35], our method uses the default integer comparator instead of redefining the string comparator operators (i.e., $<$ and $=$) while doing binary search. This is important since 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). 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 [15] 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 4. 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 4 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 4, 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 DOS, 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 4, Z_4 and Z_4^0 should be

$$Z_4(01, 10) = 0110 \qquad Z_4^0(01, 10) = 01100000$$
$$Z_4(0000, 1111) = Z_4^0(0000, 1111) = 00110011$$

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.

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.

4.4 Flatbuffers Binary Representation

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. In the CDO demo [28], we stored 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 [27] stored such big distance oracles in PostgreSQL, and SPDO [29] 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

Algorithm 1: Oracle.fbs structure for Flatbuffers in C++

```
namespace MyOracle.Sample;
struct Wsp {
  code:long;
  d:float;
}
table Oracle {
  wsps: [Wsp];
}
root_type Oracle;
```

Algorithm 2: oracle_serialize.cpp for serialization preparation

```
#include "oracle_generated.h"
using namespace MyOracle::Sample;
std::vector<Wsp> wsp_vector;
Void PrepareSerialization{
  flatbuffers::FlatBufferBuilder builder;
  foreach text file f storing WSPs do
    wsp_vector.clear();
    wsp_vector load all WSPs from file f;
    sort(wsp_vector), order by code;
    auto wsps = builder.CreateVectorOfStructs(wsp_vector);
    auto oracle = CreateOracle(builder, wsps);
    builder.Finish(oracle);
    std::ofstream filew("f_oracle_flatbuffer.out");
    filew.write(builder.GetBufferPointer(), builder.GetSize());
    filew.close();
}
```

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 performance, we need a way to serialize 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 [3]. Although Protocol Buffers [8] is 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 serialize the representation; 3) Load FlatBuffers into through mmap; 4) Use their support for binary search. 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 1, and use the compiler (e.g. flatc -c Oracle.fbs) to generate *oracle_generated.h.* After that, we use Algorithm 2 to serialize each distance oracle text file to a binary Flatbuffers file.

In the querying stage illustrated in Algorithm 3, we use *mmap* from *sys/mman.h* to virtually load all binary files in program, and *thread* library to enable multi-thread processing similar to CDO [28].

Recall that each distance oracle text file stores around 100 million WSPs. This is because Algorithm 2 needs to load all WSPs

Algorithm 3: oracle_run.cpp for the querying stage

```
#include <thread>
#include <sys/mman.h>
#include "oracle_generated.h"
using namespace std;
namespace ofb = MyOracle::Sample;
const int N = number of binary files;
const flatbuffers::Vector<const ofb::Wsp *> *wspsdata[N];
Void Process{
  foreach binary FlatBuffers file f do
    long long filesize = getFileSize(f);
    int fd = open(f, O_RDONLY, 0);
    void* mmappedData = mmap(NULL, filesize, PROT_READ,
      MAP_PRIVATE , fd, 0);
    auto result = ofb::GetOracle(mmappedData);
    wspsdata[f] = result→wsps();
  Load all one-to-one distance queries;
  Partition queries to m threads;
  for i ← 0 to mthread do
    thread[i] ← initial thread i, process distance queries using
      function GetDist(p_1, p_2) in Algorithm 4;
  for i ← 0 to mthread do
    thread[i].join();
}
```

Algorithm 4: GetDist(lat_1, lng_1, lat_2, lng_2)

```
$code \leftarrow Z_4^0( Z_2((lat_1, lng_1)), Z_2((lat_2, lng_2)) )$;
$f \leftarrow$ which file would have the WSP of $code$;
$left \leftarrow 0$;
$right \leftarrow$ wspsdata[$f$]→size();
$p \leftarrow$ compute the index that wspsdata[$f$]→Get($p$)→code() is
  the maximal code value and $\leq code$ using a binary search;
return wspsdata[f]→Get(p)→d();
```

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 USA road network, the 4.6 billion WSPs are separated into 46 binary files, and each binary file occupies around 1.6GB on disk.

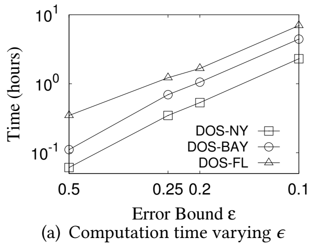
5 EXPERIMENTS

From OpenStreetMap [7], 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 $24M$ vertices ignoring the vertices not bidirectionally connected to the main graph. In addition, we add the Florida road network from the 9th DIMACS Implementation Challenge [1]. A demo is set up at http://sametnginx.umiacs.umd.edu/. Table 2 provides the characteristics of the road network datasets used in our evaluation.

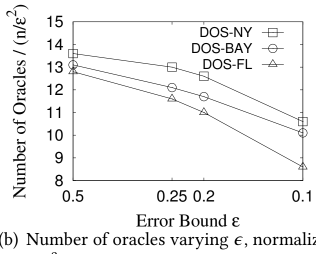
From our past experiments and previous theoretical work, we conclude that the value of ϵ greatly influences the size of the distance oracles, but it does not have much of an influence on the

Table 2: Dataset Characteristics

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	-



(a) Computation time varying ϵ



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

Figure 8: Precomputation performance varying ϵ

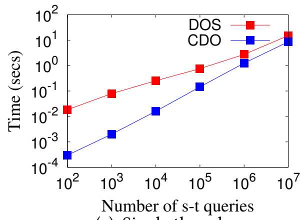
querying time as the time complexity is just $O(\log \frac{n}{\epsilon^2})$. Thus, for the small road network, we set $\epsilon = 0.1$, but let $\epsilon = 0.25$ for the whole USA road network. We implemented four methods as follows for distance computations. All of them are implemented in C++, and are 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 [28] 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 [35] 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 [22] as a representative of methods that optimize the execution of single source shortest paths.
(5) *HLDB.* We implement HLDB [11] in memory for small road networks same as in the CDO demo [28], but in PostgreSQL for the USA road network.

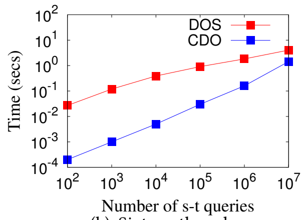
5.1 Precomputing $DO(G)$

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

Figure 8(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,



(a) Single thread



(b) Sixteen thread

Figure 9: 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.

Figure 8(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 8(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 8(b) shows that C decreases as ϵ decreases, which is a good news for applications that require higher accuracy.

Table 3: Precomputation for $DO(G)$ on US Dataset

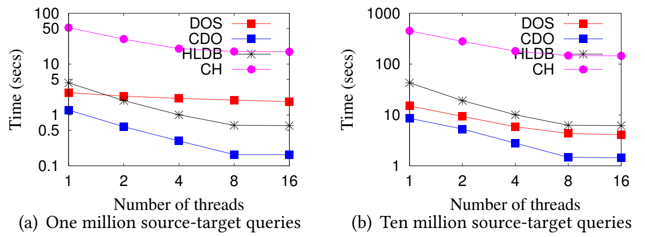
Performance	DOS-0.25
Cluster Size	Time
4	1.8 days
45	5.1 hours
Number of oracles	4.6 billion
Number of oracles / (n/ϵ^2)	11.9

Next we show scalability results in Table 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 6(a): 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. Moreover, the cluster compute and storage cost for precomputing the oracle is less than \$50 based on AWS April 2018 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.

5.2 Querying Performance

There are two group 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 PostgreSQL database. The other group is CDO, CH, and HLDB for small road networks, for which their graph representations are preloaded in memory. Among this group, CDO and HLDB in memory are only available for small road networks, e.g., number of vertices is less than one million, but CH in memory is available for large road networks.

Figures 9 and 10 show the time performance of DOS in a small road network, i.e., 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 9 shows the influence of the number of queries for DOS and CDO. Note that our DOS



(a) One million source-target queries (b) Ten million source-target queries

Figure 10: 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.

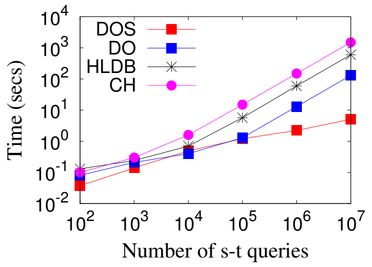


Figure 11: Time performance for DOS, DO, HLDB, and CH under the whole USA road network: DOS and CH is running with 16 threads, while DO and HLDB are running in PostgreSQL. In order to make the queries reasonable, each source-target query is generated by randomly picking one road vertex and the other road vertex within 200km.

is extended from CDO, and is available for large road networks. When the number of queries is small, DOS is much slower than CDO. This is because that DOS needs to search WSPs from disk, but CDO does the binary search in memory directly without counting tens of seconds to preloading 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 is $2.45M$ 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 10 describes the time performance varying with the number of concurrent threads for DOS, CDO, HLDB, and CH. Figure 10(a) is under $1M$ distance computations, and Figure 10(b) is under $10M$ distance computations. From Figure 9, we know DOS is under the cold-start during the first $1M$ distance computations, e.g., in Figure 10(a). This is the reason that DOS is even slower than HLDB in Figure 10(a), and its time performance does not significantly decrease with more threads. On the other hand, in Figure 10(b), DOS performs better than HLDB and gets closer to CDO as it passes the cold-start for last millions of distance computations.

Figure 11 shows the time performance for the methods that are available for large road networks. Note that our previous work

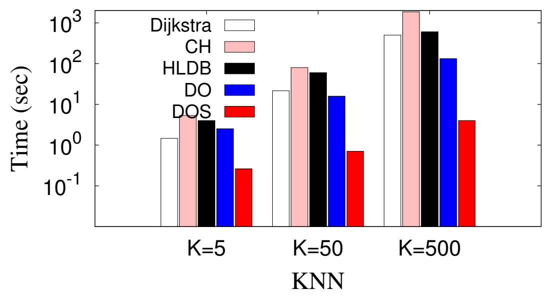


Figure 12: Response time for the KNN query including 6,070 **university sources and** 49,573 **restaurant destinations.**

CDO is not available here. To generate the source-target queries, as the source-target queries do not distribute uniformly in fact, we randomly pick one road vertex and randomly select the other road vertex within 200km. From Figure 11, DOS performs much better than DO, HLDB, and CH, especially after the cold-start stage.

5.3 Spatial Analytic Queries

DOS can efficiently process analytic queries as it can process a large number of network distance queries. The querying performance of DOS in the following experiments is after the cold-start stage. Figure 12 provides 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 http://roadsindb.com/. One location list contains 6,070 locations of universities from [4], and another location list contains 49,573 locations of fast food restaurants from [2]. The record in both lists is *(id, Z_2 code, latitude, longitude)* with the precomputed Z_2 code.

The *KNN* query with which we experimented obtains the K nearest restaurants for each university in Figure 12. 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), then 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 12, 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 [6] to obtain the job locations around the Bay Area. This workload shown in Figure 13 contains 120 million distance computations, where DOS takes 22 seconds under only the Bay Area road network, while CDO needs 18 seconds. In a general setting, i.e., DOS with the whole USA road network, this task can be also finished in 25 seconds. This is because of the spatial concentration property that make most WSPs in the Bay Area be usually in one

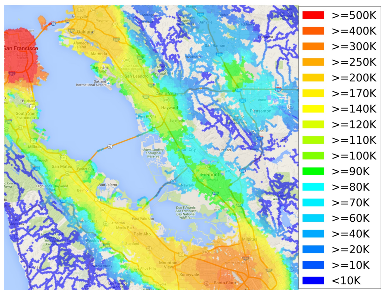


Figure 13: Nearby job opportunities (e.g., within 10 kms) for each census block in the Bay Area, requiring 120 million distance computations, which DOS finished it within 22 seconds for just the Bay Area road network, and 25 **seconds for the whole USA road network.**

or two FlatBuffers binary files. Then after the cold-start stage, these WSPs in the Bay Area would be all 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 regions.

Obviously, using DOS, many analytic queries could be solved and visualized in a much quicker way. All applications implemented in our previous work [27–29] 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 multi-machines as it is easy to copy one set-up machine to many.

6 CONCLUDING REMARKS

DOS is a practical system that utilizes all our previous distance oracle techniques such as ϵ-DO [35, 36], PCPD [38], SPDO [29], and CDO [28] . 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 would achieve a throughput of $5M$ per second after ten seconds.

Future work includes incorporating traffic and changes in the road networks such as road closures. This requires devising ways of computing the oracle in piecemeal-fashion so as to avoid doing it from scratch. We also want to incorparate our work into a spatial browser [21, 32] as well as using a distributed spatial index [40].

REFERENCES

[1] [n. d.]. DIMACS. http://www.dis.uniroma1.it/challenge9.
[2] [n. d.]. Fast food maps. http://www.fastfoodmaps.com/.
[3] [n. d.]. Flatbuffers. https://google.github.io/flatbuffers/.
[4] [n. d.]. GeoNames. http://www.geonames.org/.
[5] [n. d.]. Google Maps API. https://developers.google.com/maps/.
[6] [n. d.]. LODES. http://lehd.ces.census.gov/data/.
[7] [n. d.]. OpenStreetMap. http://www.openstreetmap.org/.
[8] [n. d.]. Protocol Buffers. https://github.com/google/protobuf/.
[9] [n. d.]. RDI. http://www.slideshare.net/CongressfortheNewUrbanism/andy-mortensonmeasuring-transportation-connectivity-by-rdi/.
[10] [n. d.]. TAREEG. http://tareeg.org/.
[11] I. Abraham, D. Delling, A. Fiat, A. Goldberg, and R. Werneck. 2012. HLDB: Location-based services in databases. In *ACM GIS*. Redondo Beach, CA, 339–348.
[12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *SEA*. Kolimpari Chania, Greece, 230–241.
[13] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *ESA*. Ljubljana, Slovenia, 24–35.
[14] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. 2007. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *ALENEX*. New Orleans, LA, 46–59.
[15] P. B. Callahan. 1995. *Dealing with higher dimensions: The well-separated pair decomposition and its applications*. Ph.D. Dissertation. The Johns Hopkins University, Baltimore, MD.
[16] P. B. Callahan and S. R. Kosaraju. 1995. A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields. *J. ACM* 42, 1 (1995), 67–90.
[17] L. Chang, J. Xu Yu, L. Qin, H. Cheng, and M. Qiao. 2012. The exact distance to destination in undirected world. *VLDB J.* 21, 6 (Dec 2012), 869–888.
[18] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. 2011. Customizable Route Planning. In *SEA*. Kolimpari Chania, Greece, 376–387.
[19] D. Delling, P. Sanders, D. Schultes, and D. Wagner. 2009. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*. Springer, Berlin, 117–139.
[20] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.
[21] C. Esperança and H. Samet. 2002. Experience with SAND/Tcl: a scripting tool for spatial databases. *JVLC* 13, 2 (Apr 2002), 229–255.
[22] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*. Cape Cod, MA, 319–333.
[23] A. V. Goldberg, H. Kaplan, and R. F. Werneck. 2006. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *ALENEX*. Miami, FL, 129–143.
[24] N. Linial, E. London, and Y. Rabinovich. 1995. The geometry of graphs and some of its algorithmic applications. *Combinatorica* 15 (Jun 1995), 215–245.
[25] S. Ma, K. Feng, H. Wang, J. Li, and J. Huai. 2014. Distance Landmarks Revisited for Road Graphs. *CoRR* abs/1401.2690 (Jan 2014).
[26] S. Nutanong, E. H. Jacox, and H. Samet. 2011. An incremental Hausdorff distance calculation algorithm. *PVLDB* 4, 8 (Aug 2011), 506–517.
[27] S. Peng and H. Samet. 2015. Analytical Queries on Road Networks: An Experimental Evaluation of Two System Architectures. In *ACM GIS*. Seattle, WA, 1:1–1:10.
[28] S. Peng and H. Samet. 2016. CDO: Extremely High-Throughput Road Distance Computations on City Road Networks. In *ACM GIS*. Burlingame, CA, 84:1–84:4.
[29] S. Peng, J. Sankaranarayanan, and H. Samet. 2016. SPDO: High-Throughput Road Distance Computations on Spark Using Distance Oracles. In *ICDE*. Helsinki, Finland, 1239–1250.
[30] M. Qiao, H. Cheng, L. Chang, and J. Xu Yu. 2014. Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme. *TKDE* 26, 1 (Jan 2014), 55–68.
[31] H. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA.
[32] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. 2003. Use of the SAND spatial browser for digital government applications. *CACM* 46, 1 (Jan 2003), 63–66.
[33] P. Sanders and D. Schultes. 2006. Engineering Highway Hierarchies. In *ESA*. Zurich, Switzerland, 804–816.
[34] J. Sankaranarayanan, H. Alborzi, and H. Samet. 2006. Distance Join Queries on Spatial Networks. In *ACM GIS*. Arlington, VA, 211–218.
[35] J. Sankaranarayanan and H. Samet. 2009. Distance oracles for spatial networks. In *ICDE*. Shanghai, China, 652–663.
[36] J. Sankaranarayanan and H. Samet. 2010. Query processing using distance oracles for spatial networks. *TKDE* 22, 8 (Aug 2010), 1158–1175.
[37] J. Sankaranarayanan and H. Samet. 2010. Roads belong in databases. *Data Engineering Bulletin* 33, 2 (Jun 2010), 4–11.
[38] J. Sankaranarayanan, H. Samet, and H. Alborzi. 2009. Path oracles for spatial networks. *PVLDB* 2, 1 (Aug 2009), 1210–1221.
[39] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. 2003. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica* 7, 3 (Sep 2003), 255–273.
[40] E. Tanin, A. Harwood, and H. Samet. 2005. A distributed quadtree index for peer-to-peer settings. In *ICDE*. Tokyo, Japan, 254–255.
[41] D. Wagner and T. Willhalm. 2003. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA*. Budapest, Hungary, 776–787.
[42] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. 2012. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *PVLDB* 5, 5 (Jan 2012), 406–417.
[43] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. 2013. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *SIGMOD*. New York, 857–868.

[3]http://roadsindb.com/