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

Shangfu Peng
University of Maryland
shangfu@cs.umd.edu

Jagan Sankaranarayanan
NEC Labs America
jagan@nec-labs.com

Hanan Samet
University of Maryland
hjs@cs.umd.edu

***Abstract*—In the past decades, shortest distance 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 analytical applications on road networks including simulations (e.g., evacuation planning), logistics, 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. A framework called SPDO is presented which implements an extremely fast distributed algorithm for computing road network distance queries on Apache Spark. The approach extends our previous work of developing the ϵ-distance oracle which has now been adapted to use Spark's resilient distributed dataset (RDD). Compared with state-of-the-art methods that focus on reducing latency, the proposed framework is able to improve the throughput by at least an order of magnitude, which makes the approach suitable for applications that need to compute thousands to millions of network distances per second.**

Keywords*—*Road networks, network distance computations, high throughput, Spark, analytical queries.

I. Introduction

A spatial analytical query on a road network performs thousands to millions of shortest distance computations in the process of answering the query. As an example, consider the heat map in Figure 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 [1] from Census by performing $13,645,807$ road network distance computations. Analytical queries that compute millions of network distances are commonplace in logistics, tour planning, and spatial business intelligence. Existing companies usually use the geodesic distance (Euclidean distance) instead of the network distance, which makes their results inaccurate [2]. 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 the geodesic distance is easy but only the shortest distance on the road is optimal. Computing such a distance matrix would require one million network distance queries. It is worthwhile to note that Google Distance Matrix [3] offers an API to compute the distance matrix but limits the service to 25×25 matrices (i.e., 625 distance computations) even to their paying customers. We need a framework to compute tens of millions of distance computations on a 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 busines s.

This work was supported in part by the NSF under Grants IIS-10- 18475, IIS-12-19023, and IIS-13-20791.

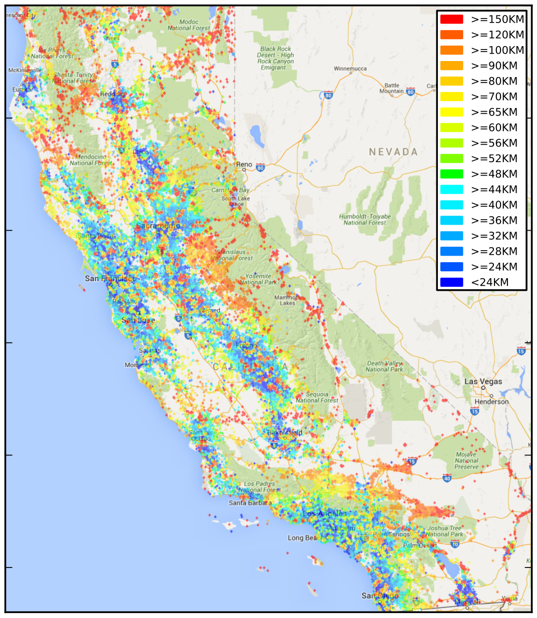


Fig. 1. The geographical heat map for the average drive distance from people's living places to their workplaces in the California region: the color of each pixel in this figure indicates the average drive distance of the people who live in the region around the pixel. The workload of this query includes $13,645,807$ shortest distance computations, and our distributed key-value method on a Spark cluster with 5 task machines finished this query in 13 seconds. In contrast, the state-of-the-art methods, e.g., CH [4], running on the same 5 machines in parallel need more than 20 minutes for the same query.

The majority of existing shortest distance methods on road networks mainly focus on decreasing the latency time for a single shortest path query. However, reviewing the spatial analytical queries in the business environment of the above-mentioned companies in [2], we see that they are less interested in the latency time of a single shortest path query, and much more about the time that they spend on computing millions of pair-wise network distances. For analytical queries, *throughput* which corresponds to the average number of distance computations per second, is a more relevant measure. Our focus is on throughput rather than latency of individual computation.

In this paper, we develop a distributed framework called SPDO (pronounced *speedo* and stands for *Spark and Distance Oracles*) using Apache Spark [5] that is optimized for high-throughput network distance computations. We extend our previous work on ϵ-Distance Oracle (ϵ-DO) [6], which is a technique to precompute and store the shortest distances between all pairs of vertices in a road network. The resulting representation is in $O(\frac{n}{\epsilon^2})$ space, where n is the number of vertices in the road network and ϵ is an approximation error bound on the result. Our previous applications [2] and [7] showed how the distance oracle representation maps to an RDBMS system and can solve complex analytical queries on a road network. Here we develop the mechanisms to map distance oracles to a distributed key-value store (i.e., hash abstraction) which we choose Spark in this paper. The combination of Spark and distance oracles results in a match made in heaven. Spark provides a highly scalable fault-tolerant distributed framework with the ability to cache large datasets in memory using RDD [5], while distance oracles provide 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 purpose, which is more than a key-value store. We can easily develop further functions in Spark combing 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 because of the ability to spread query processing across multiple machines in a Spark cluster as well as the in-memory representation of distance oracles.

Mapping the distance oracles to a distributed key-value store is challenging since 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 paper are: 1) A high-throughput architecture using distance oracles and Spark for a large set of spatial analytical 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 code-base 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 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.

The rest of this paper is organized as follows. Section II reviews the related work. Section III introduces our notation and summarizes our previous work developing ϵ-DO. Section IV explains the theoretical work about how the distance oracles can be mapped to a hash structure, and Section V presents three variants of our distributed key-value algorithms. They are denoted as Basic, BS, and WP, respectively. Section VI describes a detailed experimental evaluation of our methods, and also provides two real applications using our methods. Concluding remarks are drawn in Section VII.

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

II. 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 [8], which is very efficient for single source queries named *one-to-many pattern* in [2], e.g., find the nearest K destinations around a given source. However, for an s-t query, Dijkstra's algorithm has to scan many insignificant 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 the observation 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, [9] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. *Contraction Hierarchies* (CH) [4] assigns an importance score to each node and replaces some original edges by shortcuts. [10]–[14] build an explicit hierarchy graph to overcome the drawback of Dijkstra's algorithm.

The lookup-based methods usually need to store some pre-computing results. [15]–[18] 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 [19] based on hub labels (HL) [15] is a recent practical method that embeds the shortest distance computation into an RDBMS. *Road Network Embedding* (RNE) [20] applies a Lipschitz embedding [21] to a spatial networks, such that vertices of the spatial network become points in a high-dimensional vector space. [22] take 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. *Spatially Induced Linkage Cognizance* (SILC) [23] is based on the observation mentioned in [22] 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 [6], [24] 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) [24] gives one exact shortest path algorithm, while the *ϵ-Distane Oracles* (ϵ-DO) [6] proposes an approximate shortest distance algorithm, which balances between accuracy and storage.

A few of other approaches focus on speeding up specific analytical queries. Knopp et al. [25] explain how to use highway hierarchies [13] for computing many-to-many shortest distances. [20] and [23] show how to speed up the K nearest neighbor search by their s-t techniques, respectively. Delling et al. [26] utilize partition-based algorithms developed for s-t queries to handle POI queries. Cho et al. propose UNICONS [27] for continuous nearest neighbor queries, and ALPS [28] for top-k spatial preference search. Our recent paper [2] proposes an integrated architecture that embed the distance oracles into an RDBMS, and developed many SQL solutions for solving a variety of spatial analytical queries.

III. Preliminaries

A. Notation

TABLE I. The summary of Notations

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	the maximum depth of the DO-tree
M	the number of task machines

We first summarize all the notations that we will use later in Table I. A road network G is modelled 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. A property of road networks is that $\frac{m}{n}$ is typically a small positive number that is independant of n. In addition, without loss of generality, each node v has $p(v)$ denoting 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). We define the network distance $d_G(u, v)$ to be the shortest distance from u to v in the road network, while $d_E(u, v)$ to be the geodesic distance.

We use the Morton (Z) order space-filling curve [29] 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 $2^D \times 2^D$ equal sized blocks named *unit blocks*, where D is a positive integer named the maximal decomposition depth. Figure 2 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, number representation and string representation. As in Figure 2, since the number representation cannot distinguish the number 0 at depth 1 from the number 0 at depth 2. The completed number representation should be associated with 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, and use the number representation in practice since it is more efficient.

A spatial graph $G(V, E, w, p)$ on the domain S can also be divided into $2^D \times 2^D$ unit blocks. Given a vertex v in the unit block i, the Morton code $mc(v)$ is equal to $mc(i)$. 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 in the string representation.

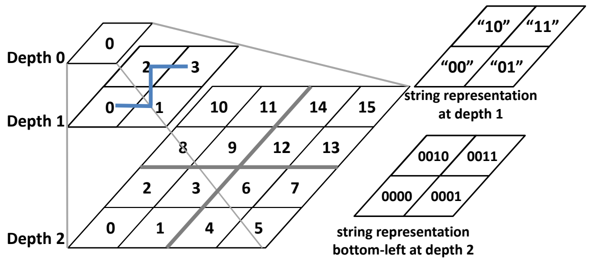


Fig. 2. Example of the number representation and the string representation of Morton codes in a domain space when $D = 0$, 1, and 2, respectively.

B. Review of WSP and ϵ-DO theory

The ϵ-DO [6] 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 describes a well-separated pair decomposition (WSPD) [30] of a road network in order to produce well-separated pairs (WSP), e.g., (A, B) with particular network distance 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), which satisfies the property that for any pair of vertices (x, y), $x \in A$ and $y \in B$, 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

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

As a result, ϵ-DO generates $O(\frac{n}{\epsilon^2})$ well-separated pairs, denoted as $(A, B, d_\epsilon(A, B))$. Both A and B are a PR quadtree block [29] at the same depth. In order to make a well-separated pair easily embed in a database as a key-value pair, ϵ-DO use the Morton (Z) order space-filling curve [29] 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 $mc(mc(A), mc(B))$.

To retrieval the network distance between a pair of geographic locations, $p_1 = (lat_1, lng_1)$ and $p_2 = (lat_2, lng_2)$, the ϵ-DO requires that the comparison operator COMP(mc_a,mc_b) in a database system be redefined, where mc_a and mc_b are two

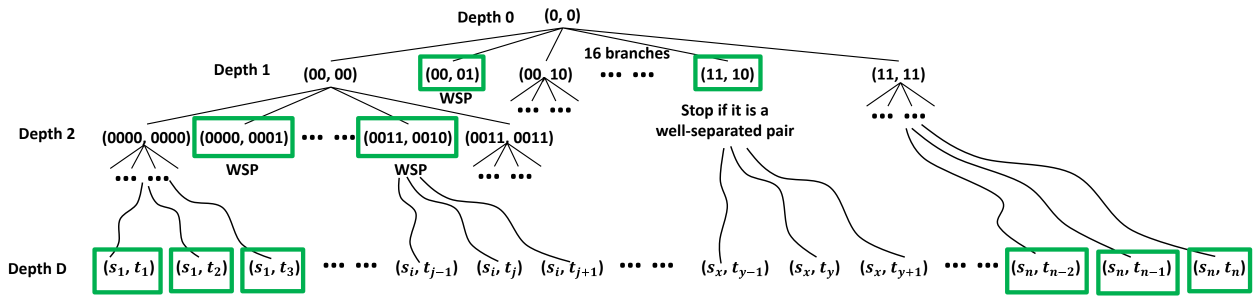


Fig. 3. Example of the DO-tree which represents the distance oracles pre-computation: each node is a 4-dimensional Morton code denoting a pair of 2-dimensional Morton codes. Each node is decomposed into 16 nodes unless it corresponds to a WSP, i.e., the nodes inside a green rectangle, in which case no 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.

4-dimensional Morton codes. COMP(mc_a,mc_b) should yield equality in the case that mc_a is a prefix of mc_b or mc_b is a prefix of mc_a. Thus, ϵ-DO obtains the network distance by finding the unique well-separated pair $(key, dist)$, where COMP(key, $mc(mc(p_1), mc(p_2))$) yields equality. Note that the uniqueness property which comes here from the property of WSP [30] guarantees that there is exactly one well-separated pair containing (p_1, p_2).

C. Benefits and Problems in a Distributed Key-value Store

In the past several years, many key-value stores appear such as Berkeley DB [31], HBase [32], Redis [33], etc. In the distributed environment, a key-value store supports hash access model well, which is like a HashMap data structure in Java, so that people can find a given key and its value in $O(1)$ time complexity. Note that although some key-value stores also support sorting keys in order, hash access model is better for parallel processing the workload.

Spark [5] is not a pure key-value store, but a general-purpose cluster computing framework. *IndexedRDD* [34] 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 ϵ-DO works seamlessly on any key-value store. Thus, we can load all well-separated pairs in Spark's distributed memory. Even for the ϵ-DO of the USA road network, several machines with 32 or 64GiB memory are enough. Embedding ϵ-DO in Spark is the best solution so far for the spatial analytical 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 [6]. 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.

IV. Hash Access for Distance Oracles

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 distance oracle of [6] 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.

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 3 shows the *DO-tree*, which has several properties that we develop to build a theortical 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 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 correspond 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 4.1: 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 contains 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 4.1, 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 can retrieve 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 make $(D+1)$ lookups. Given a source s and destination t, we start out by making a 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 if so the we can obtain the approximate network distance of s and t. If $\mathbb{H}_1$ does not contain the key, 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 4.1. ■

The advantage of one property of looking up values in $\mathbb{H}_1$ is that the $O(D)$ lookups can be done concurrently as opposed to doing them sequentially. The reason for it 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, since 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 since it results in 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 corresponds to exactly one WSP. To compute the number of non-leaf nodes, we use a similar approach to that taken in [6], [30]. 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 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 informs 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 $O(\log n)$ [6], 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.

V. Implementation in 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 paper 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 analytical query such as in Figure 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 [34], 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 [35] achieves good throughput and

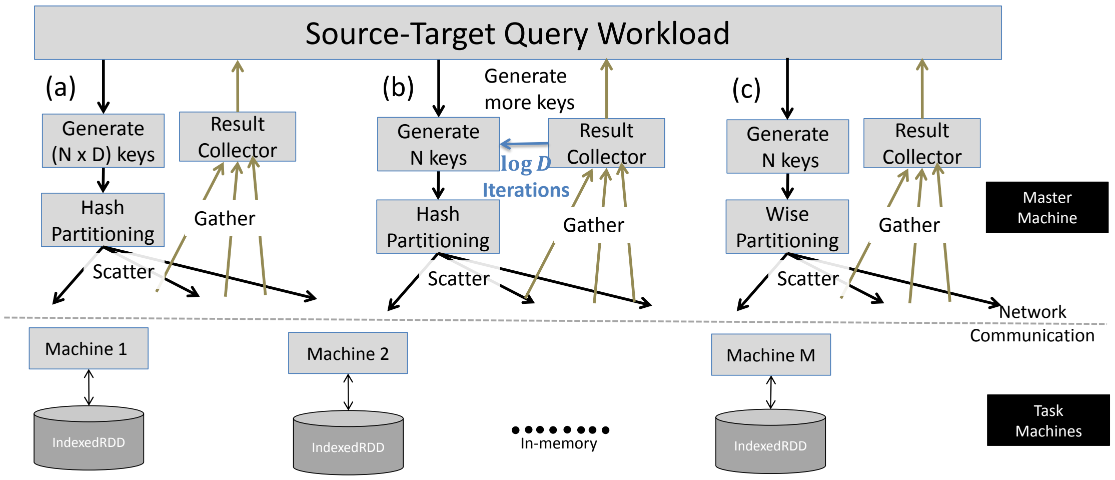


Fig. 4. The three implementations of our methods, namely (a) Basic, (b) Binary Search and (c) Wise Partitioning methods, on top of Apache Spark

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 having $O(1)$ time complexity.

Algorithm 1: 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 ($hash$, $Q$);
```

A Spark program consists of a master and a task programs. Algorithm 1 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, where the keys are assigned by the master machine. In the remaining section, we discuss three variants of Algorithm 1 that only vary from GETDIST().

A. Basic Method

Algorithm 2: 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 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 4.1 of $\mathbb{H}_1$ ensures that they cannot occur.

Figure 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 ϵ-DO, 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 the HP.

B. 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 3 and Figure 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 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 done 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

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

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 previous two iterations such that one search resulted in a success and the other in a failure. The search continues until it finds a depth d that is present in the hash table and a depth $d+1$ that is not present. This process can continue $\log D$ times since we are performing a binary search on the D depths of the OD-tree.

C. Wise Partitioning Method

```
Algorithm 4: 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 $result$;
```

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 further queries. Since each task machine simply looks up a local hash map, their computational workload is much smaller than that of the master machine. In the case of 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(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 the reason that the master machine takes on a heavy workload in the Basic and BS methods. Recall, that it needs to coordinate the search among multiple task machines, collect results from all the task machines, and even generate new keys to try out in the case of the BS method.

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 ranges from $[d, D]$ 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. At this point, the distance value is communicated to the master machine.

D. Analysis of Methods

Table II contains the summary of our analysis of the time and space complexity 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

TABLE II. ANALYSIS ON THE THREE DISTRIBUTED WORK FLOWS

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

iterations so multiple iterations in the case of 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 analytical 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.

VI. EVALUATION

In this section, we present a detailed evaluation of our distributed key-value solutions in comparison with the CH method [4] which is a state-of-the-art algorithm for finding a single shortest path in a road network, the distance oracle implementation from [6], and an efficient implementation of Dijkstra's algorithm. The comparisons are detailed in in Section VI-B. 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 VI-B. Our comparisons use four workloads: a batch of s-t pairs in Section VI-C, distance matrix queries in Section VI-D, route directness spectrum in Section VI-E and job accessibility map in Section VI-F. 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.

A. 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 V-A, the binary search method (BS) in V-B, and the wise partitioning method (WP) discussed in V-C.

DO. We compare against the distance oracle *DO* method of [6] as it is representative of methods that can perform network distance computations inside a database. In this case, we load DO 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 DO. 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 [4] 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 [8] from [2], 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 [36]. To make the comparisons fair, we restrict all the methods (i.e., our Spark-based methods, the CH method, and the Dijkstra method) to utilize just one CPU in each machine.

B. Datasets and Cluster Setup

TABLE III. DATASET CHARACTERISTICS

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 III 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 [37], and the Bay road network is from OpenStreetMap [38] extracted using the TAREEG [39] 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 DO 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 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 [40]. We use the LEHD dataset [1] 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.

C. 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 comparative methods on a workload where there may not be significant commonality across different queries Later in Section VI-D, we compare these methods on another workload where we compute the network distances from one source to multiple targets with the goal of taking advantage of location commonality.

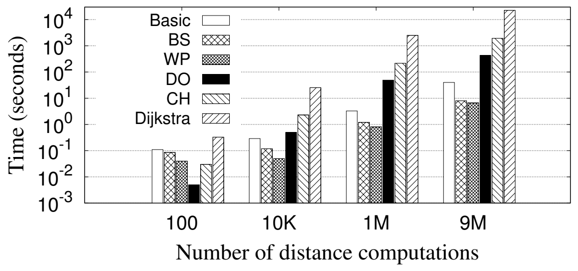


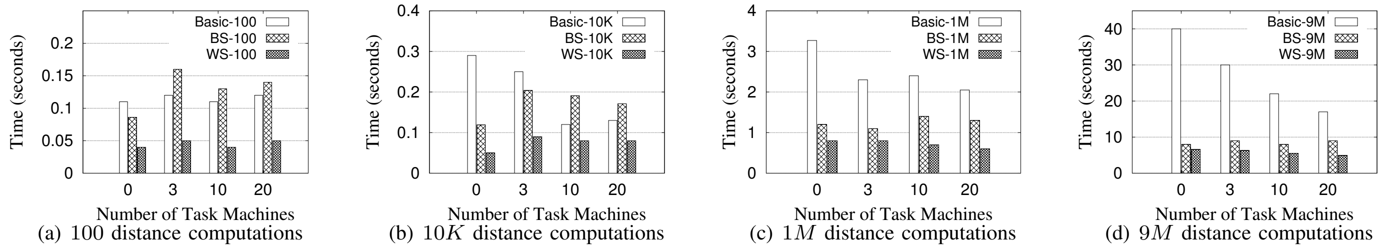
Fig. 5. The execution time in local mode in NYC: the number of s-t pairs are 100, 10 thousand, 1 million, and 9 million, respectively. Note that the y-axis is logarithmic scale. It shows that our BS and WP methods are significant faster than other methods.

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 Section VI-C1 since this represents the case when there is no network communication. Later in Section VI-C2, we show experimental results on a cluster of task machines. We will vary the number of task machines to study its effect on the performance of our Spark-based methods.

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 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 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. One reason for the similar performance of BS and WP is that without the network communication costs, there is very 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 [2] 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 dominating cost for the small query workload. Both WP and BS are significantly



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

Fig. 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 as increasing the number of task machines, we see that the bottleneck of our BS and WP methods is the master machine.

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.

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 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 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 V-D, 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 IV. THROUGHPUT OF THE 6 METHODS FOR THE US DATASET RUNNING ON 20 TASK MACHINES

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 IV 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 that 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.

D. Distance Matrix Workload

The distance matrix query is the simplest form of an analytical 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.

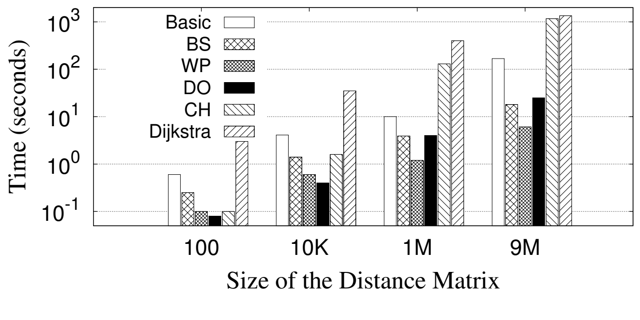


Fig. 7. Distance matrix computation queries for various methods on US dataset with 20 task machines

Figure 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. The details of this optimization are provided in [2].

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.

E. *Route Directness Spectrum*

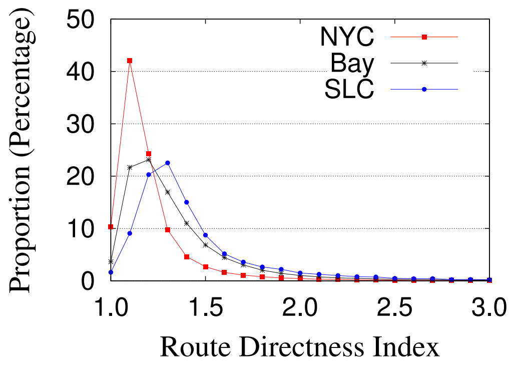


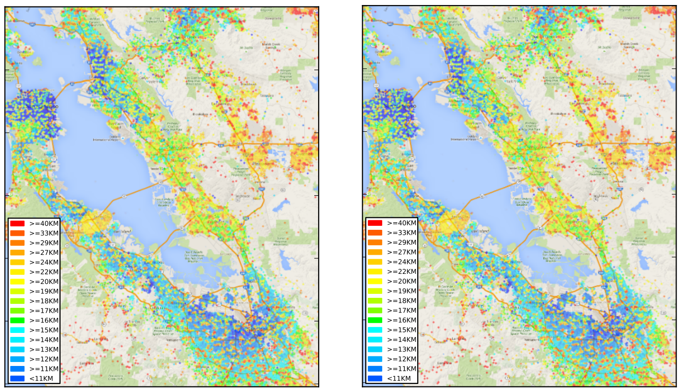
Fig. 8. The route directness spectrum (RDS) of New York City (NYC), the Bay Area (Bay), and Salt Lake City (SLC). In contrast, the maximum route directness index, which corresponds the maximum ratio between network distance and geodesic distance, is 10.6 in NYC, 30.4 in Bay, and 26.3 in SLC, respectively.

A query that is of immense interest to transportation planners is a measure called *route directness index* [41], which requires making billions of distance computations. The route directness index of any two locations in the road network is the ratio between the shortest network distance to the geodesic distance. The route directness spectrum is a distribution of the route directness index as a proportion of the total shortest paths in the road network. Figure 8 shows the Route Directness Spectrum of the NYC, Bay Area, 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 since 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).

One way of computing the route directness spectrum of a road network is to impose a grid on the road network. We compute the ratio of the network distance between the centroids of two grid cells to its geodesic distance and weight it by the product of the number of vertices in each grid cell. We use the results to approximate their route directness indexes since Narasimhan et al. [42] proved that the route directness index of a WSP provides a bound of the the route directness index between any pair of vertices that in the WSP. Figure 8 was produced by bucketing and counting up all 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 geodesic distance is always less than or equal to the network distance. As the values of the maximum route directness index that we found were 10.6 30.4, and 26.3 for NYC, Bay, and SLC, respectively, solely using the maximum route directness index is not a good way to measure the connectivity of a road network.

It is important to note that using our distributed key-value methods, the computing the route directness spectrum for a moderate-sized region such as NYC can be completed within 1 minute, while other methods complete it in a reasonable time (e.g., more than 4 hours using CH, since there is a very large number of distance computations to be performed).

F. *Job Accessibility*



(a) Using the WP method (b) Using the CH method

Fig. 9. The 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 seconds; (b) results computed by CH with 3 task machines in 5 mins. The 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 analytical queries.

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 [43]) pointed out that measures of accessibility can reveal if a transportation system meets peoples needs [43], not to mention revealing the economic vibrancy of a census block. The dataset that is used for such an analysis is the LEHD dataset [1] from the US census which first subdivides the map into 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 travelled 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 1. Our WP method generated Figure 1 in 13 seconds using 5 task machines, while using the same number of task machines CH needs 20 minutes. Figure 9 shows the result for a small section of California, i.e., San Francisco Bay Area. Figures 9(a) and 9(b) show the results of using the WP and CH methods, respectively.

VII. Concluding Remarks

We presented SPDO, a framework for computing road network distances using Apache Spark and ϵ-approximate distance oracles. We presented 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 compute millions of distance computations on a road network using just a few machines. We also showed how SPDO can significantly speed up complex spatial analytical queries and discussed two complex use-cases, namely the route directness spectrum and job accessibility.

References

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

[2] S. Peng and H. Samet, "Analytical queries on road networks: An experimental evaluation of two system architectures," in *ACM GIS*, Seattle, WA, Nov 2015.

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

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

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

[6] J. Sankaranarayanan and H. Samet, "Distance oracles for spatial networks," in *ICDE*, Shanghai, China, Apr 2009, pp. 652–663.

[7] ——, "Query processing using distance oracles for spatial networks," *TKDE*, vol. 22, no. 8, pp. 1158–1175, Aug 2010.

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

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

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

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

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

[13] P. Sanders and D. Schultes, "Engineering highway hierarchies," in *ESA*, Zurich, Switzerland, Sep 2006, pp. 804–816.

[14] 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*, New York, Jun 2013, pp. 857–868.

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

[16] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao, "The exact distance to destination in undirected world," *VLDB J.*, vol. 21, no. 6, pp. 869–888, Dec 2012.

[17] M. Qiao, H. Cheng, L. Chang, and J. X. Yu, "Approximate shortest distance computing: A query-dependent local landmark scheme," *TKDE*, vol. 26, no. 1, pp. 55–68, Jan 2014.

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

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

[20] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh, "A road network embedding technique for k-nearest neighbor search in moving object databases," *GeoInformatica*, vol. 7, no. 3, pp. 255–273, Sep 2003.

[21] N. Linial, E. London, and Y. Rabinovich, "The geometry of graphs and some of its algorithmic applications," *Combinatorica*, vol. 15, pp. 215–245, Jun 1995.

[22] D. Wagner and T. Willhalm, "Geometric speed-up techniques for finding shortest paths in large sparse graphs," in *ESA*, Budapest, Hungary, Sep 2003, pp. 776–787.

[23] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD*, Vancouver, Canada, Jun 2008, pp. 43–54.

[24] J. Sankaranarayanan, H. Samet, and H. Alborzi, "Path oracles for spatial networks," *PVLDB*, vol. 2, no. 1, pp. 1210–1221, Aug 2009.

[25] 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.

[26] D. Delling and R. F. Werneck, "Customizable point-of-interest queries in road networks," *TKDE*, vol. 27, no. 3, pp. 686–698, Mar 2015.

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

[28] H. Cho, S. J. Kwon, and T. Chung, "ALPS: an efficient algorithm for top-k spatial preference search in road networks," *KAIS*, vol. 42, no. 3, pp. 599–631, Mar 2015.

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

[30] P. B. Callahan, "Dealing with higher dimensions: The well-separated pair decomposition and its applications," Ph.D. dissertation, The Johns Hopkins University, Baltimore, MD, Sep 1995.

[31] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *USENIX*, Monterey, CA, Jun. 1999, pp. 183–191.

[32] D. Carstoiu, E. Lepadatu, and M. Gaspar, "HBase: Non-SQL database performances evaluation," *IJACT*, vol. 2, no. 5, pp. 42–52, 2010.

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

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

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

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

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

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

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

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

[41] Measuring Transportation Connectivity by RDI. http://www.slideshare.net/CongressfortheNewUrbanism/andy-mortensonmeasuring-transportation-connectivity-by-rdi/.

[42] G. Narasimhan and M. H. M. Smid, "Approximating the stretch factor of euclidean graphs," *SIAM J. Comput.*, vol. 30, no. 3, pp. 978–989, 2000.

[43] SSTI. http://www.ssti.us/events/accessibility-towards-a-new-multimodal-system-performance-metric/.