Proceedings of the 23nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, M. Ali, M. Gertz, Y. Huang, M. Renz, and J. Sankaranarayanan, eds., Seattle, WA, November 2015.

Analytical Queries on Road Networks: An Experimental Evaluation of Two System Architectures *

Shangfu Peng Hanan Samet

Center for Automation Research
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742 USA
{shangfu, hjs}@cs.umd.edu

ABSTRACT

Spatial analytical queries on road networks typically perform hundreds of thousands to several millions of shortest distance computations in the process of producing results. These queries require architectures that can compute a large number of network distances. Two architectures are evaluated on a variety of spatial analytical queries on road networks. The first architecture is a widely used hybrid architecture that uses a database to store spatial datasets, a road network distance computing module, and an analysis tool to tie them together into a single query processing pipeline. The second architecture uses of a distance oracle representation of a road network. This architecture stores the spatial datasets and the distance oracle inside the database, and the query processing is completely handled by the database. A detailed evaluation of the two architectures for a variety of analytical query processing tasks such as region, KNN, distance matrix and trajectory queries is presented and the lessons learned are discussed.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

spatial analytical query, distance oracle, road network, system architecture

1. INTRODUCTION

The past two decades has seen a steady increase in processing spatial queries. Such functionality has been implemented in early systems such as QUILT [34, 43] and SAND [22, 35] which had a browsing capability to full-fledged mapping applications such as MapQuest, Yahoo Maps, Google Maps, and Bing Maps. They all require the computation of the distance between two locations x and y which in our work is more accurately represented as the network distance *DIST (x,y)* instead of the Euclidean distance $||x-y||$, or variants of it such as a minimum distance to a block boundary (e.g., [33, 40]) or the Hausdorff distance (e.g., [29]). Beyond simple navigation queries, location-based web services like Google Maps repeatedly pose queries on a road network and utilize the results to serve a user base. For example, Google Distance Matrix offers an API that computes the distance matrix between a set of origins and a set of destinations. Other examples include analysts who use OLAP stores to perform complex simulations on road networks to help answer queries such as determining where to locate an additional Walmart among a number of potential locations, or the roads where bottlenecks exist for evacuation planning purposes. Moreover, mobile services frequently interact with write-optimized stores to store the current positions of mobile hosts as they move about in a road network. These services also frequently compute the distances from their mobile hosts to other mobile hosts or landmarks in order to provide services such as locating the k nearest restaurants or gas stations. We use the term *spatial analytical queries* to collectively describe such queries. The challenge lies in taking note of the realization that each such instance of a spatial analytical query invariably involves being able to make hundreds to as many as millions of computations of distance along a spatial network rather than as the crow flies.

In the face of a massive amount of spatial analytical queries from internet scale users, for example, Google Maps [6] drastically restricts the number of shortest distance results per query (e.g., a limit of 100 shortest distances per query using the Google Distance Matrix API). Most other existing services such as Yelp just use Euclidean distance instead of network distance. Figure 1 illustrates the drawback of using Euclidean distance. It shows Yelp's response to the query: find the restaurants around River Road, Edgewater, NJ (blue icon) with the distance filter that they are within a 2 mile biking distance. Obviously the 5^{th} and 9^{th} results that lie on the other side of the river are impossible to reach by biking less than 2 miles. However, they are in the result set (e.g., Flat Top, the 5^{th} result, is 1.3 miles away using Euclidean distance).

Reviewing previous research work, we find none that are concerned with general spatial analytical queries. Instead, they focus on speeding up one specific type of query, e.g., KNN search queries [16, 18, 29, 31], CNN queries [15], and distance matrix [25]. However, these algorithms are not easy to extend to include general spatial analytical queries. On the other hand, most state-of-the-

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

SIGSPATIAL'15, November 03-06, 2015, Bellevue, WA, USA

DOI: http://dx.doi.org/10.1145/2820783.2820806.

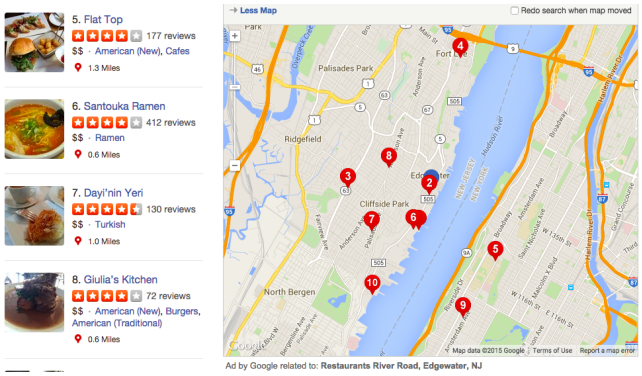

Figure 1: Yelp results for the query: find the restaurants around River Road, Edgewater, NJ that are within a 2 mile biking distance.

art methods such as HL [10], TNR [13], CH [23], etc, focus on decreasing the latency time for a single source-target (s-t) query, which is the basic unit of a spatial analytical query. Although decreasing the latency time for one s-t query results in reducing the total response time for a spatial analytical query, it is far from enough since these methods don't take into account considerations such as multi-users, multi-threads, reused results, and query optimization that can be used to speed up a spatial analytical query.

An alternative approach to speeding up spatial analytical queries is to take advantage of the query optimizer associated with the database system which makes use of selectivity factors about the underlying data (i.e., stored in the relations being operated on). For example, suppose that we want to find the cities with population greater than $500,000$ within 200 miles of the Mississippi River. We have two options here. Armed with knowledge about the complexity of executing a within (or buffer) algorithm as well as the data distribution, the query optimizer can either call for performing the spatial selection first or the relational selection first where the choice will depend on the number of cities with such a population and and the size of the spatial area in question. As another example, suppose that we want to find all stores within 25 driving miles of a warehouse. Here, armed with knowledge of the complexity of finding entities within a given driving distance as well as the data distribution, the query optimizer can either call for a solution based on finding the nearby stores vis-a-vis the individual warehouses or on finding the nearby warehouses vis-a-vis the individual stores where the choice depends on the number of warehouses and the number of stores. The important thing to note about these examples is that the query optimizer requires knowledge about the complexity of an external module or algorithm that is executed outside the database. Unfortunately, such knowledge is usually not present and thus users cannot rely on it.

In this paper, we study a general efficient solution for spatial analytical queries. Our contribution is two-fold. The first is the proposal of two architectures, the hybrid architecture (HY) and the integrated architecture (DO), for solving spatial analytical queries. The second is the formulation of efficient solutions for spatial analytical queries using our previously developed technology, the ϵ-distance oracle (ϵ-DO) [38, 39].

We propose the architectures and their modules by reviewing and summarizing the existing solutions and use cases. Most existing spatial analysis tools use the HY architecture illustrated in Figure 2, which separates the modules into two parts. The first part deals with point-of-interest (POI) locations, relations, and attributes in a database system, and the second part retrieves shortest distance results on a road network, which is usually processed in memory. Our previous work called ϵ-DO [38] is the first attempt to answer spatial queries within a database system, which makes the DO architecture in Figure 3 possible. Embedding map-based services within a database system is attractive as it allows developers to leverage the power of a database language to create new types of online services resulting in easy programming, customization, and maintenance. In addition, there are ample opportunities to use optimization to speed up a spatial analytical query like finding the nearest restaurant for each coffee shop which ends up making millions of shortest path queries.

In order to make our ϵ-DO more powerful, in this paper, we develop more efficient solutions. In particular, here we present SQL solutions for KNN and trajectory queries. These two types of queries serve as building blocks to enable people to easily write SQL solutions for more complex queries. In our examples, each spatial analytical query is expressed by just a few lines of SQL that utilize pre-defined functions. In contrast, the situation is far more complicated if for each of the queries users would have to devise efficient programs in Java (or other high level programming languages) to obtain the necessary query results.

Finally, we experimentally evaluate our DO solution in a database in conjunction with a high-performance implementation of Dijkstra's algorithm with multi-threads on the entire USA road network. Dijkstra's algorithm is the most widely used method for spatial analytical queries (e.g., in Esri [4]) as it is adaptable for most spatial analytical queries and is efficient for the single source query. Our experimental results demonstrate that our DO solution is able to yield at least $34,000$ shortest distance query results per second on a commodity machine. This means that our solution is at least an order of magnitude faster than the highest-performance implementation of Dijkstra's algorithm for most analytical tasks.

The rest of this paper is organized as follows. Section 2 introduces the analytical queries and provides an overview of our two architectures. Section 3 presents the traditional hybrid architecture for most existing spatial analysis tools, while Section 4 proposes the integrated architecture using the ϵ-distance oracle. Section 5 describes a detailed experimental evaluation of our solutions, and we provide our SQL functions and codes online in [8]. Section 6 discusses the lessons we learned, Section 7 summarizes related work, and conclusions are drawn in Section 8.

2. SPATIAL ANALYTICAL QUERIES

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

USE CASE 2.1. *I am a taxi operator running a fleet of taxis. I have a dataset of taxi trips each with a unique ID such that each trip has a latitude and longitude values for both a pickup and a drop off point, as well as for way points at irregular intervals. Such a dataset constitutes the trajectory information for each taxi trip. I want to obtain the total number of miles travelled by each taxi this month. This information is useful in computing the actual profit per km of all the vehicles in my fleet and determining which of my drivers are better performers.*

USE CASE 2.2. *I am an operator of a large hospital and have the geocoded address of my patients and the locations of my clinics. Our hospital has more than 500 clinics across the country. Each patient is assigned to the nearest clinic. I want to get the average*

drive time for patients per clinic. This is an important metric in healthcare since the further one has to drive, the poorer are the health outcomes. This distance also informs us of the need to open new clinics or relocate existing ones to better serve our patients.

USE CASE 2.3. *I have a trucking company with 10 trucks that deliver thousands of packages for a popular retailer. A common operation that I run several times during each day is determining which packages should be loaded on to which truck and the order in which they should be delivered. This decision process constitutes a complex tour planning query that tries to minimize the total network distance travelled by all trucks, as well as to accommodate the priority assigned to each package. For this purpose, the input is a network distance matrix between the delivery locations of all current packages. An optimization program would decide how to assign packages to trucks and the order in which to deliver them. Note that even with 1000 packages, the query will compute up to 1 million pair-wise network distances.*

For spatial analytical queries on road networks, there are two common reasons why such queries end up making a very large number of distance computations. First, spatial analytical queries are typically used for generating insights into the data in the form of reports or visual representations. So it is common for these queries to end up accessing large portions of the data. Second, the queries may join two or more datasets on the basis of the network distance to other objects on the road network, such as finding the nearest neighbors from one dataset for each location in another dataset, or group one or more datasets based on the closest distance to objects in another dataset. Executing all of these operations can easily end up making millions of distance computations on the road network. For instance, just the simple query that obtains the network distance between all pairs of objects drawn from a set of 1000 objects to one another ends up making 1 million distance computations on the road network.

Since the spatial analytical queries are an important use-case whose efficiency depends on being able to compute millions of network distance computations efficiently on road networks, there is a need to examine which of existing available architectures are capable of efficiently processing these queries. Most existing tools for spatial analytical queries have several limitations, or use the basic Dijkstra's algorithm. For example, the Google Distance Matrix API [6] limits non-paying users to submit 100 shortest distances (10 origins and 10 destinations) per query, and obtain $2,500$ shortest distances per 24 hour period. For paying customers, the limits are 625 shortest distances per query, and $100,000$ shortest distances per 24 hour period. Esri [4] also claims that the ArcGIS Network Analyst extension, namely the Route, Closest Facility, and OD Cost Matrix solvers, are based on the well-known Dijkstra's algorithm for finding shortest paths. All these tools are not good enough to solve spatial analytical queries.

Spatial analytical queries make two distinct kinds of access patterns on road networks, and make millions of these accesses in the process of answering a query. The most basic pattern is called *one-to-one* pattern which computes the distance between a source and a destination on the road network. Another access pattern is *one-to-many* that makes several s-t pair computations from the same source vertex. For instance, computing the K nearest neighbors for each point from a large dataset makes one-to-many access patterns. There are opportunities for speeding up one-to-many patterns even though they are nothing more than multiple one-to-one access patterns. We use the term *scan* to describe the actual implementation of the execution of an access pattern. Note that there can be many options for executing a scan including Dijkstra's algorithm, contraction hierarchies (CH) [23], etc.

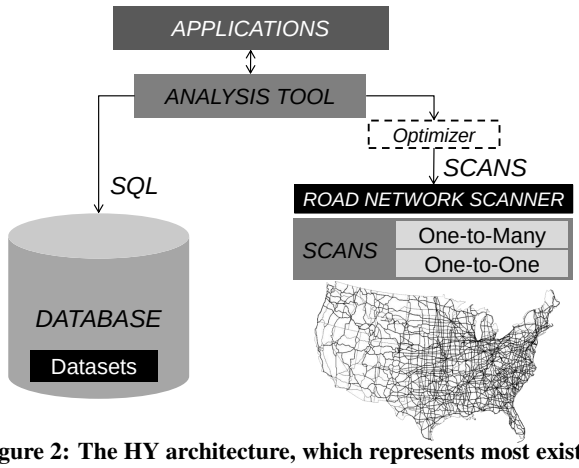

Figure 2: The HY architecture, which represents most existing spatial analysis tools

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

The first architecture is a hybrid architecture that uses a database to store and query spatial datasets, but then uses an external module that loads the road network in the main memory and performs fast in-memory scans on the road network. This approach takes advantage of the large amount of available memory in modern computers as well as the high number of processing cores to be able to compute a large number of scans quickly. An analysis tool coordinates the data transfer and the issuance of scans to the road networks. A common example of such an approach is the process used by the ArcGIS Network Analyst to solve problems such as Use Case 2.2. The information pertaining to both clinics and patients is maintained in a database system. In order to compute the average driver distance for each clinic, the ArcGIS Network Analyst first retrieves the related data from the database, and then scans the network starting at the clinic using Dijkstra's algorithm which here is implementing a one-to-many access pattern. The scan process stops when it has obtained all the network distances of the clinic's patients.

The second architecture incorporates the road network inside the database as a single relation. The road network is stored as a distance oracle [38] relational table indexed by a B-tree. Scans on the road network become lookups on the B-tree index which is very efficient to perform. This method relies on being able to perform the queries entirely inside a database and on using the declarative nature of an RDBMS to automatically optimize queries.

3. HYBRID ARCHITECTURE

The most common architecture for responding to spatial queries on a road network is a hybrid (denoted by HY) one consisting of a database to store the spatial datasets and a module external to the database to execute the actual operations on the road network. Figure 2 shows such a representative architecture that combines a database, an analysis tool, and an external module for network processing. An example of a system that deploys such an architecture is ArcGIS from Esri [4], a popular platform for performing deep analysis to make informed decisions. The analysis tool is at the heart of this architecture in the sense that it extracts the necessary data from the database, pre-processes it, and contacts the road network scanner to perform the necessary scans on the road network.

In this architecture, the analysis tool partitions the query processing into two parts. The first part queries the database to access the spatial datasets, such as restaurants, gas stations, real estate information, warehouses, etc. When the volume of datasets is large (which is the usual case), this database is usually a conventional off-the-shelf database; if their volume is small, then they may even be loaded into main-memory from files (e.g., shapefiles) in which case we have a main-memory database. The second part uses the analysis tool's road network module to compute network distances between the objects. Actually, the network distance computations are incorporated into the processing and the result is either displayed to the user or stored back into the database. In this model, the network processing happens entirely outside the database while the analysis tool serves as the "glue" that coordinates computations between the database and the road network module.

The road network scanner is an in-memory processing module that implements the execution of the access patterns on large road networks. It contains at least one spatial index such as a k-d tree or R-tree to locate the given locations, and one or more priority queues to speed up the scan process. In order to fully utilize the computing power of multi-cores, this module also needs to employ several processing threads. Each thread responds to one scan process at a time. Its not worth to parallelize the workload inside one scan process using several threads as previous work [17, 28] has shown that parallelization of Dijkstra's algorithm and similar scan-based algorithms with traditional locks and barriers has disappointing performance. In particular, our implementation of HY built a k-d tree in the main thread which was pre-loaded with the vertices of the graph. The main thread uses the k-d tree to locate the source point of a given scan task, while the destination points of the scan task are obtained from the POI table. Next, our implementation sets up several scanning threads to process Dijkstra's algorithm (we could have also used another method like CH) to obtain distance results. All scanning threads share the graph representation, and each thread keeps a private scanned queue to store the visited vertices and a heap (as we are using Dijkstra's algorithm) to determine the next vertex to scan. Each time that the main thread is presented with a scan task, the main thread first locates the source point using the k-d tree, and then assigns the scan task with its associated source and destination points to a waiting scan thread. All threads use busy-waiting.

In addition, between the analysis tool and the road network scanner, an optimizer module would be useful to automatically optimize the scan plan. However, often, users dispense with this step as most existing analysis tools rely on the user's specification of the scan plan. At the end of Section 4, we give an example to see the importance of having an optimizer.

The road network scanner in Figure 2 shows two access patterns for retrieving shortest distances. The one-to-one access pattern is the commonest, although the one-to-many access pattern is also commonly used as it is optimized for multi-destinations. Note that ideally there should also be a third access pattern of the form many-to-many. However, very few access pattern implementation algorithms are designed for a many-to-many access pattern because its effect can be obtained by resorting to multiple instantiations of the one-to-one and one-to-many access patterns as in [25].

Each specific algorithm that retrieves the shortest distances and paths implements one of the two access patterns. For example, Dijkstra's algorithm is good for the one-to-many access pattern, while CH [23] is good for the one-to-one access pattern. The road representation of a specific algorithm is the lowest component in Figure 2, e.g., Dijkstra's algorithm uses the original graph representation and TNR [13] uses the hierarchical tree representation.

Below we provide an abstraction of the operation of the scan procedures instead of the details of the algorithms that implement them. In particular, our HY architecture given in Figure 2 implements the following operators on road networks:

DEFINITION 3.1. *The* SCAN*() operator scans the road network in memory using one of the scan-based algorithms, $\mathcal{A}$. Given $G(\mathcal{A})$, the graph representation of $\mathcal{A}$, and vertex s,* SCAN*(s) uses $\mathcal{A}$ to scan $G(\mathcal{A})$ starting at s.*

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

DEFINITION 3.2. SCAN_UNTIL_K*(k, s, P) scans the graph starting at vertex s, and returns the k nearest objects o to s, where $o \in P$ and P is the POI set.*

DEFINITION 3.3. SCAN_UNTIL_DIST*(d, s, P) scans the graph region within network distance d from vertex s, and returns all the objects o, where $o \in P$ and P is the POI set, and o is within network distance d from s.*

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

4. INTEGRATED ARCHITECTURE

The database community desires an integrated architecture, which means that all components and procedures reside in a database. This makes the architecture more compact and efficient, as the analytical query executes entirely within the database. The database knows how to optimize such queries, since the road representation appears as one or several relations in the database, and thus the query appears like any other relational query to the database. The core challenge lies in how to embed the road representation in the database. For example, pgRouting [7] extends the PostGIS / PostgreSQL geospatial database to provide geospatial routing functionality. Oracle Corporation proposed the Network Data Model Graph (NDM) [30], which persistently manages the network connectivity in the database, while a Java API provides fast in-memory graph analytics. However, since the road representation in both pgRouting and NDM is the original graph representation, they are similar to the HY architecture except for storing the nodes and edges in their database. HLDB [11] proposed by Microsoft Research is the first practical system that can answer spatial queries on continental road networks stored entirely within a database. It stores the vertices of the road network, as well as sets of "forward" and "backward" hub labels (HL) of the vertices [10] in the database. Each $s-t$ query is solved by performing a JOIN on the corresponding "forward" and "backward" relational tables of s and t, respectively.

In this section, we propose the integrated architecture that makes use of our previously developed technology, the ϵ-DO [38]. The distance oracle [38] takes a road network as input, and reduces it to a single database relation that captures the network distances between every pair of vertices in the road network. The technique is

based on the notion of *spatial coherence*, which can be described intuitively using the following example. The network distance between any vertex (more generally any location denoted by its latitude and longitude) in the Washington, DC region to any vertex in the Boston, MA region can be reasonably approximated by a single distance value. This is because the shortest path regardless of where one starts in the DC region or wants to go in the Boston region ends up using I-95N. This large overlap in the shortest paths means that the network distance between sources in Washington, DC and destinations in Boston, MA can be approximated by a single value with a bounded approximation or error tolerance. Furthermore, as the sources and destinations get farther apart, one can approximate even larger regions of sources and destinations with a single value. For instance, Maryland and California can be approximated by a single value with a bound on the approximation error since the sources and destinations are quite far from one another.

Using a cluster of 50 EC2 machines, it took us about 6 hours to compute the distance oracle of the USA road network which contained 24 million vertices. The precomputation process decomposed the road network with n vertices into $O(\frac{n}{\epsilon^2})$ triples (A, B, d) stored in a relational table, such that A and B are denoted by blocks in a PR quadtree and d is the network distance that approximates the network distance between every pair of vertices contained in A and B within an ϵ error tolerance. In particular, d is also said to be ϵ-approximate, which means that the resulting error in using d instead of the exact network distance between the any vertex in A and any vertex in B is bounded by the ϵ error tolerance. The resulting representation for the entire USA was about 55GB in size with an error tolerance $\epsilon = 0.25$. This is a reasonable setting for real road networks. In particular, in our previous work [38], we showed that although the error tolerance ϵ is 0.25, the approximate distance value of at least 20% of the vertex pairs has an error of less than 1%. Moreover, the average error for random queries is just 2.74%. The relational table corresponding to the distance oracle is indexed by a B-tree representation that allows disk efficient loopkups for approximate distances.

Using the distance oracle, we created an integrated architecture illustrated in Figure 3. The key difference from the hybrid architecture is the use of the distance oracle road representation, which has been embedded in a database as a simple relational table. To query the distance oracle, we implemented an SQL function called DIST(), which queries the distance oracle relational table to compute the road distance between any source and destination. In particular, given two latitude/longitude pairs, DIST() first computes a unique code which it looks up in the distance oracle relational table, and then uses a simple SELECT query that is facilitated by the B-tree index. For example, computing the network distance between the White House and the US Capitol Building in Washington, DC becomes as simple as the following query that

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

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

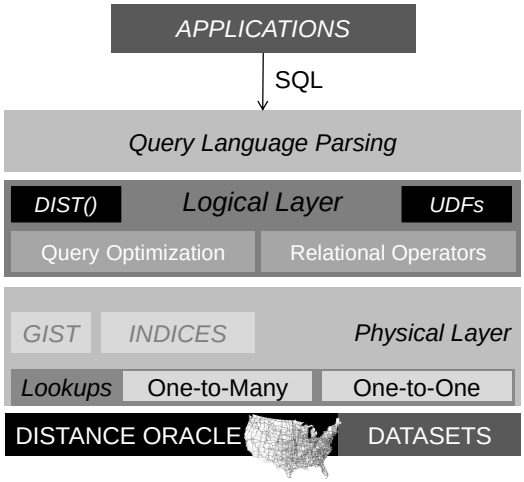

Figure 3: Integrated architecture DO for analytical queries using the distance oracle.

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

Here we see how to express a complex query with just a few lines of SQL. Now contrast this with performing the same query in the traditional setup which uses a module where the road network would be stored externally as in Figure 2. The road network would typically be accessible through an API such as SCAN_UNTIL_K() and SCAN_UNTIL_DIST() for computing shortest paths and distances. In this setup, the hybrid architecture would first obtain a table of houses and parks from the database. Next, we have two options to obtain the distance results using the SCAN_UNTIL_DIST(). The first is for each house, to compute the number of parks within 0.5 km. The second is for each park, to compute the houses within 0.5 km, and then to group the distance results by the house ids. Finally, for each house, count the number of parks and order them in descending order of the count.

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

5. EXPERIMENTS

In this section, we present a detailed evaluation of the two architectures in order to compare and contrast their query performance. Section 5.1 describes the experimental setup and datasets. Sections 5.2 and 5.3 evaluate the performance of both HY and DO for

the region and throughput queries, respectively. We synthetize the queries in these two subsections from the POI tables that we used. Next, Sections 5.4 and 5.5 show our solutions for the KNN and trajectory queries, respectively, in realistic settings.

5.1 Experimental Setup and Datasets

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

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

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

We used two POI tables for the evaluation. The *restaurant* table consists of $49,573$ fast food restaurants obtained from [9], and the *university* table consists of $5,964$ locations of universities from [5]. The schemas of both tables are identical and are *(id, latitude, longitude, gid, geom)*, where `gid` and `geom` are needed for the GiST index on the latitude/longitude values.

We also used a taxi trajectory dataset. This taxi dataset was from San Francisco Yellow Cab [1] collected by CRAWDAD [3]. It contained $11,220,058$ GPS entries for 537 taxis covering a one month period in 2008 comprising $928,307$ trips. The schema for the taxi trajectory relation is given in Table 1.

Table 1: Schema for table *taxi* storing the taxi GPS information.

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

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

5.2 Region Query

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

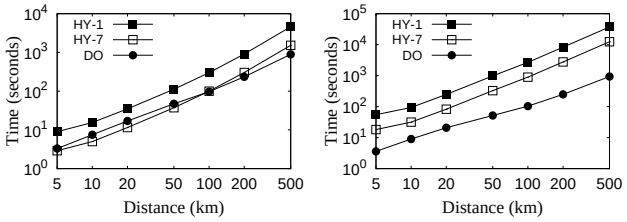

Figure 4: Time comparison between HY and DO varying with the furthest distance values for (a) restaurant is destination, and (b) university is destination

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

In the DO architecture we don't want to compute the distance between all pairs of universities and restaurants. Therefore, we need a simple way of reducing the number of invocations to the distance oracle. One way to do this is to take advantage of the fact that the Euclidean distance is a lower bound on the road network distance and thus we restrict the pairs of objects that we examine to be within their Euclidean distance. This is achieved by using a query search window of width $2X$ around each university and only examining the restaurants lying in it. This translates to a query window of width $2X/111$ degrees assuming that 1 degree of latitude/longitude roughly equates to a geodesic distance of 111 kms. The SQL statement for this query is as follows.

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

Figure 4 shows the execution time of DO and HY when varying the width of the query region. For HY, we show the performance of running 1 and 7 scanning threads using HY-1 and HY-7 respectively. The reasons for using 7 scanning threads are explained in Appendix A. Figure 4(a) shows the results of region queries that find the restaurants within X kilometers of each university, while Figure 4(b) interchanges the sets that form the sources and destinations so that now we find the universities within X kilometers of each restaurant.

This experiment is to HY's advantage in the sense that it can amortize the costs of the scans from a single source to multiple destinations. Nevertheless, Figure 4(a) shows that for smaller values of the distance X, HY is slightly better than DO but this advantage vanishes as X increases with DO performing better than HY for $X > 100$. The setting of Figure 4(a) where we find the restaurants near the universities represents the worst case for DO as we expect many restaurants to be clustered around each university compelling DO to query the distance oracle once for each pair. On the other hand, when we change the setting so that we find the universities near the restaurants as in Figure 4(b), we find that the execution time of HY is at least one order of magnitude greater than DO

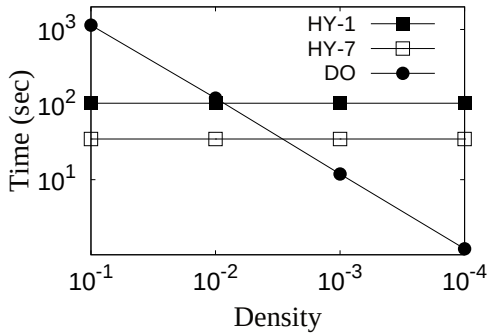

Figure 5: Execution time versus a synthetically varying density of destinations for $5,964$ distance queries (corresponding to the size of the university sources relation at distance 50 km.

since there may not be too many universities around each of the restaurants thereby greatly reducing the number of queries to the distance oracle. While this experiment shows that DO is sensitive to the density of the destinations, in the worst case it still performs as well as HY, while easily outperforming it in other cases.

To further explore the effect of density on DO's performance, we performed the following experiment. Define *density* to be the ratio of the number of destinations found to the number of vertices of the road network visited by HY during its scan around each source point for a given region. In some sense, density controls the *work* efficiency of HY vis-a-vis DO in that the larger the density, the greater is the benefit obtained by HY from amortizing the scans. Figure 5 shows the execution times of HY and DO when performing the region query that finds all synthetically generated destinations within 50 km of all universities. We created destinations around each university by generating points with a probability denoted by the density value. Figure 5 shows that the density does indeed affect DO as we had expected but does not affect HY for the region query. In particular, as the density decreases (i.e., the points become sparse), DO improves dramatically as the number of invocations of the distance oracle is greatly reduced. Note that although DO is slower when the density is larger than 0.01, it is fairly obvious that for real world datasets, a density of more than 1 restaurant per 100 vertices is extremely large to be realistic.

5.3 Throughput Query

Table 2: Comparison between s-t pair and one-to-many

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

From the previous results we see that DO provides a single DIST() function that computes the road distance between any pair of source and destination locations on the road network. HY on the other hand is optimized for one-to-many distance computations since the scan amortizes the work done for scanning from a single source to multiple destinations. To better understand the performance of each architecture we compare them using a distance matrix query.

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

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

Table 2 shows the performance of the DO and HY architectures. DO computes the distance matrix in 8853.9 seconds, while HY does it in 20139 seconds. The throughput for DO is 33.3k distances/second while it is 14.6k distances/second for HY. Note that this is in spite of choosing a query workload that is favorable to HY. This shows that for a practical real query, DO is still $2.4\times$ better than HY in terms of throughput.

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

5.4 KNN Query

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

The HY architecture invokes SCAN_UNTIL_K(), which uses an in-memory graph representation and stores the $49,573$ restaurants relation in a k-d tree data structure. During processing, HY uses 7 threads to scan the road network. Each thread starts scanning from one of the university locations and for each vertex it performs a lookup on the k-d tree to determine if there are any restaurants in its vicinity within a certain distance range. If yes, then they are enqueued with the appropriate network distance if they have not been visited before. This check is not necessary if while building the k-d tree, each restaurant is associated with its nearest vertex. This process is entirely in-memory and extremely efficient to perform.

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

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

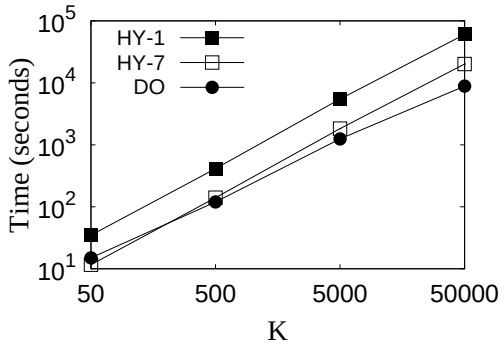

Figure 6: The execution time of $5,964$ KNN queries where $K = 50, 500, 5000$, and 49573.

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

Figure 6 shows the execution time of the KNN queries for different values of K. We see that HY has nearly identical performance compared to DO for smaller values of K less than 500. It becomes $2-3$ times worse for larger values of K such as for $K = 49,573$. At $K = 49,573$, the query degenerates to compute the distance matrix between the source and destination tables.

Figure 7 illustrates the effect of the density on the execution time of DO and HY for the KNN queries. In particular, each point in the figure corresponds to the performance of one KNN query for either HY or DO. Here we use a real world dataset and thus for the values of K that we used, the density of most scans is less than 0.01. Compared to the region query discussed in Section 5.2, the effect of varying the density has a different effect on the execution time of DO and HY. In particular, for the KNN query, the execution time of HY increases significantly as the density decreases, while the execution time of DO does not change much. This is because the number of DIST() invocations for DO is proportional to K in real world datasets. On the other hand, SCAN_UNTIL_K() for HY needs to scan further to visit at least K destinations when the density of the nearby destinations is sparser.

5.5 Trajectory Query

We now examine a simple trajectory query implemented on both the HY and DO architectures. The goal of the query is to compare performance on a real trajectory dataset consisting of GPS readings. GPS devices are becoming commonplace and are now deployed on many different commercial and non-commercial vehicles. A company operating a fleet of taxis usually has a GPS installed in all of its vehicles, which enables an operator to know the locations of its vehicles. For instance, when a customer requests

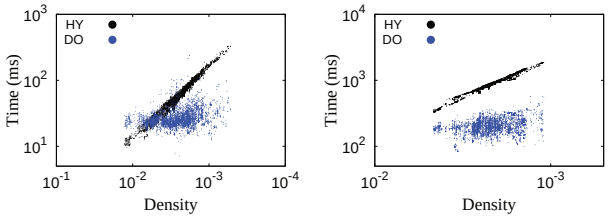

Figure 7: The execution time of the KNN query as a function of the density for (a) K = 500 and (b) K = 5000.

a ride, the taxi operator uses the current locations of all of its vehicles to send the nearest vehicle to the customer. Of course, there are more complex analyses that an operator may want to perform from historical (i.e., a day/week/month/year) worth of GPS information collected from vehicles which can shed light on several aspects of their businesses.

For example, consider a query that seeks the execution time of computing the total trip distance of each taxi. More SQL queries can be found online in [8]. Executing it using DO involves a few simple steps as detailed below.

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

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

Figure 8 demonstrates that DO is much better for the trajectory query. Each point in Figure 8 corresponds to one taxi. The x-axis is the number of segments for each taxi. For each method, we first sorted the 537 points by the number of segments, and then connected the 537 points with a line. DO computed the travel distance of one taxi within 0.2 to 0.5 seconds when the number of segments is around $10,000$. It is at least one order of magnitude faster than HY with 7 scanning threads, and two or more orders of magnitude faster than HY with just 1 scanning thread.

6. LESSONS LEARNED

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

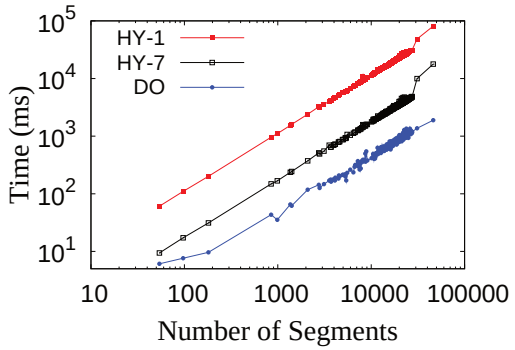

Figure 8: The execution time of computing the total travel distance of each one of 537 taxis.

with the analysis tool. This can be viewed as a drawback of the architecture since an optimization opportunity may be be lost in the part of the processing of the spatial analytical queries that is outside the database system.

There are some operations that can be executed entirely outside a database. For instance, most logistics applications take a road distance matrix as input and apply a complex optimization function. These queries are inherently procedural and cannot be executed entirely inside a database. Thus even the DO architecture for these cases degenerates to HY. Nevertheless, even for these cases, our experimental results show that often DO provides a higher throughput in computing the distance matrix.

From the perspective of ease of use, DO is better than HY since users can easily express complex queries using SQL. There is no need for much of a learning curve since we only extended SQL by one function (i.e., DIST()). DO can be implemented on any database system as it requires no modification to the database system which means that now road networks can be incorporated with any legacy database that is already hosting spatial datasets. The DO architecture is also easier to extend to distributed database or distributed system such as Apache Spark, when the scale of the network and datasets becomes larger.

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

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

7. RELATED WORK

It is well-known that Dijkstra's algorithm [21] is very efficient for single source queries such as finding the nearest K restaurants to a given location. However, for an s-t query, Dijkstra's algorithm has to scan many irrelevant vertices to reach the given target vertex. A number of techniques have been proposed to overcome the drawbacks of Dijkstra's algorithm for s-t queries on road networks. They fall into two main categories: memory-based methods and database-centric methods.

Memory-based methods: Most of the state-of-the-art approaches are memory-based, They can be subdivided into two groups. The first group are *graph-based*, which are based on the observation that some vertices in a spatial network are more important for shortest path queries, while offering different trade-offs between preprocessing time, storage usage, and query time. Goldberg et al. [24] prunes unimportant vertices using a bidirectional version of Dijkstra's algorithm. CH [23] assigns an importance score to each node and replaces some original edges by shortcuts. [10, 14, 32, 27] precompute the shortest distances between landmarks or hub nodes and other vertices, and then answer the shortest distance queries by assuming the shortest path passes through one landmark or hub node. [12, 13, 20, 37, 46] build an explicit hierarchy graph to overcome the drawback of Dijkstra's algorithm. The second group are *spatial based methods*, which overcomes the drawback of Dijkstra's algorithm by using geometric techniques. RNE [44] applies a Lipschitz embedding [26] to a spatial network so that vertices of the spatial network become points in a high-dimensional vector space. [36, 40, 45] use the fact that the set of shortest paths from vertex u to all other vertices can be decomposed into subsets based on the first edges on the shortest paths from u to them. SILC [36, 40, 41] stores these subsets in a variant of a region quadtree where all vertices stored in a quadtree block are in the same subset.

Database Centric methods. On the other hand, approaches rooted in database mainly focus on database-centric methods. [38, 39, 42] exploit the spatial coherence so that if two clusters of vertices are sufficiently far away, then distances between pairs of points in different clusters are similar. PCPD [42] gives one exact shortest path algorithm, while the ϵ-distance oracle [38, 39] propose an approximate shortest distance algorithm, which balances the trade-offs between accuracy and storage. HLDB [11] is a recent practical database-centric method that is based on hub labels (HL) [10], which is a popular memory-based method. HLDB [11] claims that most of the memory-based approaches surveyed in [19] are difficult to embed into a database system and to use with SQL queries since they rely on complicated data structures such as graphs and priority queues. One of the main contributions of HLDB is embedding the memory-based HL method into a database.

On the other hand, there are also a few approaches that focus on speeding up specific spatial analytical queries. Some are based on the techniques that speed up the s-t queries. Knopp et al. [25] explain how to use highway hierarchies [37] for computing many-to-many shortest distances. Shahabi et al. [44] and Samet et al. [36] show how to speed up the K nearest neighbor search by using different source-target techniques. Delling et al. [18] utilize partition-based algorithms developed for s-t queries to handle POI queries. Cho et al. [15] propose UNICONS for continuous nearest neighbor queries, and then propose ALPS [16] for top-k spatial preference search.

8. CONCLUDING REMARKS

In this paper we compared two architectures HY and DO with a focus on computing large numbers of road network distances. Here HY represents a traditional approach to dealing with road networks, which is an external module outside of a database system. HY is inspired by the view that road networks are too cumbersome to be stored in a database system, RDBMS specifically, since performing operations on road networks inside an RDBMS typically required extensive changes to the database system (e.g., [7, 30]).

DO represents an approach where the road network distances are precomputed and represented as a relational table. This approach reduces a road network into a single distance relational table such

that computing the road network distance requires a simple look up into a table, which aligns well with the strengths of a database system. Of course, DO can be trivially modified to not only store network distances but also the travel time factoring in the current traffic, walking distance, using public transit, etc.

Taking into account the broad computing trends of plummeting computing and storage costs, and that road networks change only gradually over time, it is not inconceivable that DO has the best chance of success. The DO architecture of the future should not only compute distances, but can also provide routing and all within the confines of a database system. Performing complex operations on road networks is as simple as downloading an appropriate "road oracle" and querying is as simple as writing an SQL query.

9. REFERENCES

[1] Cabspotting. http://cabspotting.org/.
[2] DIMACS. http://www.dis.uniroma1.it/challenge9.
[3] CRAWDAD. http://crawdad.cs.dartmouth.edu/~crawdad/epfl/mobility/.
[4] ESRI. http://www.esri.com/.
[5] GeoNames. http://www.geonames.org/.
[6] Google Maps API. https://developers.google.com/maps/.
[7] pgRouting. http://pgrouting.org/.
[8] SQL Examples of Distance Oracles. http://roadsindb.com/.
[9] Fast food maps. http://www.fastfoodmaps.com/.
[10] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, Kolimpari Chania, Greece, May 2011.
[11] I. Abraham, D. Delling, A. Fiat, A. Goldberg, and R. Werneck. HLDB: Location-based services in databases. In *ACM GIS*, pages 339–348, Redondo Beach, CA, Nov 2012.
[12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, Ljubljana, Slovenia, Sep 2012.
[13] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, pages 46–59, New Orleans, LA, Jan 2007.
[14] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB J.*, 21(6):869–888, Dec 2012.
[15] H. Cho and C. Chung. An efficient and scalable approach to CNN queries in a road network. In *PVLDB*, pages 865–876, Trondheim, Norway, Aug 2005.
[16] H. Cho, S. J. Kwon, and T. Chung. ALPS: an efficient algorithm for top-k spatial preference search in road networks. *KAIS*, 42(3):599–631, Mar 2015.
[17] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *IPDPS*, pages 1–8, Long Beach, CA, Mar 2007.
[18] D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. *TKDE*, 27(3):686–698, Mar 2015.
[19] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. Springer, Berlin, Jan 2009.
[20] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *SEA*, pages 376–387, Kolimpari Chania, Greece, May 2011.
[21] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
[22] C. Esperança and H. Samet. Experience with SAND/Tcl: a scripting tool for spatial databases. *JVLC*, 13(2):229–255, Apr 2002.
[23] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, Cape Cod, MA, May 2008.
[24] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, Miami, FL, Jan 2006.
[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] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, Jun 1995.
[27] S. Ma, K. Feng, H. Wang, J. Li, and J. Huai. Distance landmarks revisited for road graphs. *CoRR*, abs/1401.2690, Jan 2014.
[28] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, pages 393–404, Venice, Italy, Aug 1998.
[29] S. Nutanong and H. Samet. Memory-efficient algorithms for spatial network queries. In *ICDE*, pages 649–660, Brisbane, Australia, Apr 2013.
[30] Oracle Corporation. Oracle spatial and graph network data model white paper. Technical report, Mar 2015.
[31] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, Berlin, Germany, Sep 2003.
[32] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *TKDE*, 26(1):55–68, Jan 2014.
[33] H. Samet. Distance transform for images represented by quadtrees. *IEEE TPAMI*, 4(3):298–303, May 1982.
[34] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, Nov 1984.
[35] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *CACM*, 46(1):63–66, Jan 2003.
[36] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, Vancouver, Canada, Jun 2008.
[37] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA*, pages 804–816, Zurich, Switzerland, Sep 2006.
[38] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, Shanghai, China, Apr 2009.
[39] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *TKDE*, 22(8):1158–1175, Aug 2010.
[40] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *ACM GIS*, pages 200–209, Bremen, Germany, Nov 2005.
[41] J. Sankaranarayanan, H. Alborzi, and H. Samet. Distance join queries on spatial networks. In *ACM GIS*, pages 211–218, Arlington, VA, Nov 2006.
[42] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, Aug 2009.
[43] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, Apr–Jun 1990.
[44] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, Sep 2003.
[45] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA*, pages 776–787, Budapest, Hungary, Sep 2003.
[46] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *SIGMOD*, pages 857–868, New York, Jun 2013.

APPENDIX

A. HY PERFORMANCE TUNING: NUMBER OF THREADS

The reason we use 7 scanning threads in our experiments is that the EC2 machine only has 8 cores. In our implementation, Dijkstra's algorithm in HY starts T threads for scanning. Figure 9 shows the execution time for the distance matrix query as the number of threads started by the main thread is varied. From the figure we observe that the execution time for 7 scanning threads is between $\frac{1}{4}$ and $\frac{1}{3}$ of the time when we have just one scanning thread. As we expected, in order to utilize the whole computing power, the optimum number of threads is equal to the number of cores minus one. This means that one core runs the main thread, and the remaining cores run the remaining threads, one thread per core.

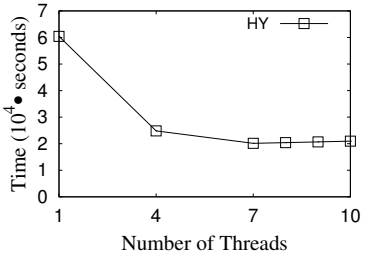

Figure 9: Execution time of a multi-thread Dijkstra's algorithm implementation for $5,964$ SCAN_UNTIL_K() with $K = 49,573$