Route-Saver: Leveraging Route APIs for Accurate and Efficient Query Processing at Location-Based Services

Yu Li, and Man Lung Yiu

Abstract—Location-based services (LBS) enable mobile users to query points-of-interest (e.g., restaurants, cafes) on various features (e.g., price, quality, variety). In addition, users require accurate query results with up-to-date travel times. Lacking the monitoring infrastructure for road traffic, the LBS may obtain live travel times of routes from online route APIs in order to offer accurate results. Our goal is to reduce the number of requests issued by the LBS significantly while preserving accurate query results. First, we propose to exploit recent routes requested from route APIs to answer queries accurately. Then, we design effective lower/upper bounding techniques and ordering techniques to process queries efficiently. Also, we study parallel route requests to further reduce the query response time. Our experimental evaluation shows that our solution is 3 times more efficient than a competitor, and yet achieves high result accuracy (above 98%).

Index Terms—H.2.4.h Query processing, H.2.4.k Spatial databases

1 Introduction

The availability of GPS-equipped smartphones leads to a huge demand of location-based services (LBSs), like city guides, restaurant rating, and shop recommendation websites, e.g.,OpenTable, Hotels, UrbanSpoon.[1] They manage points-of-interest (POIs) specific to their applications, and enable mobile users to query for POIs that match with their preferences and time constraints. As an example, consider a restaurant rating website that manages a dataset of restaurants $\mathcal{P}$ (see Fig. 1a) with various attributes like: location, food type, quality, price, etc. Via the LBS (website), a mobile user q could query restaurants based on these attributes as well as travel times on road network to reach them. Here are examples for a range query and a KNN query, based on travel times on road network.

```
select * from P where P.TV = 'yes'
                and TIME(q,P.loc) < 10
select * from P where P.price < 5
       order by TIME(q,P.loc) limit 2
```

A successful LBS must fulfill two essential requirements: ($\mathcal{R}1$) accurate query results, and ($\mathcal{R}2$) reasonable response time. Query results with inaccurate travel times may disrupt the users' schedules, cause their dissatisfaction, and eventually risk the LBS losing its users and advertisement revenues. Similarly, high response time may drive users away from the LBS.

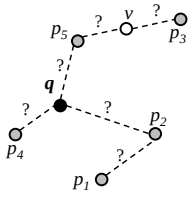
Observe that the live travel times from user q to POIs vary dynamically due to road traffic and factors like rush hours, congestions, road accidents. As a case study, we used

• *Y. Li and M. L. Yiu are with the Department of Computing, Hong Kong Polytechnic University, Hong Kong.*
E-mail: {csyuli, csmlyiu}@comp.polyu.edu.hk

1. www.opentable.com www.hotels.com www.urbanspoon.com

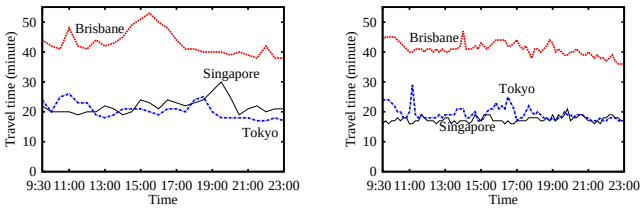
ID	loc.	food type	quality	price	TV
p_1	(4,1)	burger	5	4	√
p_2	(6,2)	grill	2	2	×
p_3	(8,7)	cafe	5	3	√
p_4	(1,3)	burger	3	2	×
p_5	(3,6)	grill	4	5	√

(a) dataset $\mathcal{P}$



(b) live travel times

Fig. 1. A restaurant rating website: data and queries



(a) on Oct 17, 2012 (Wed) (b) on Oct 24, 2012 (Wed)

Fig. 2. Measurement of live travel times

Google Maps to measure the live travel times for three pairs of locations in Brisbane, Singapore, and Tokyo, on two days (see Fig.2). Even on the same weekday (Wednesday), the travel times exhibit different trends. Thus, historical traffic data may not provide accurate estimates of live travel times.

Unfortunately, if the LBS estimates travel times based on only local information (distances of POIs from user q), then query results (for range and KNN) would have low accuracy (50% for NoAPI, see Fig. 7). Typical LBS lacks the infrastructure and resources (e.g., road-side sensors, cameras) for monitoring road traffic and computing live travel times [32] [33]. To meet the accuracy requirement ($\mathcal{R}1$), the framework SMashQ [32] [33] is proposed for the LBS to answer KNN queries accurately by retrieving live travel times (and routes) from *online route APIs* (e.g.,

Google Directions API [7], Bing Maps API [4]), which have live traffic information [6]. Given a query q, the LBS first filters POIs by local attributes in $\mathcal{P}$. Next, the LBS calls a route API to obtain the routes (and live travel times) from q to each remaining POI, and then determines accurate query results for the user. As a remark, online maps (e.g., Google Maps, Bing Maps), on the other hand, cannot process queries for the LBS either, because those queries may involve specific attributes (e.g., quality, price, facility) that are only maintained by the LBS.

Using online route APIs raises challenges for the LBS in meeting the response time requirement ($\mathcal{R}2$). It is important for LBS to reduce the number of route requests for answering queries because a route request incurs considerable time (0.1s-0.3s) which is high compared to CPU time at LBS (see Fig. 8 and 11). SMashQ [32] [33] obtains the latest travel times for queries from online route API. Though it guarantees accurate query results, it may still incur a considerable number of route requests.

In this paper, we exploit an observation from Fig. 2, namely that travel times change smoothly within a short duration. Routes recently obtained from online route APIs (e.g., 10 minutes ago) may still provide accurate travel times to answer current queries. This property enables us to design a more efficient solution for processing range and KNN queries. Our experiments show that our solution is 3 times more efficient than SMashQ, and yet achieves high result accuracy (above 98%). Specifically, our method Route-Saver keeps at the LBS the routes which were obtained in the past δ minutes (from an online route API), where δ is the expiry time parameter [17]. For instance, based on Fig. 2, we may set δ to 10 minutes. These recent routes are then utilized to derive lower/upper bounding travel times to reduce the number of route requests for answering range and KNN queries.

Another related work [31] studies how to cache shortest paths for reducing the response times on answering shortest path queries (but not range/KNN queries in this paper). They mainly exploit the *optimal subpath* property [15] of shortest paths, i.e., all subpaths of a shortest path must also be shortest paths. Given a shortest path query (s, t), if both nodes s, t fall on the same (cached) shortest path, then the shortest path from s to t can be extracted from that cached path. Unfortunately, this optimal subpath property is not powerful enough in reducing the number of route requests significantly in our problem. This is because each path contains a few data points and thus the probability for points lying on the same path with the query point is small. We show in an experiment (see Fig. 10) that the optimal subpath property ('$\tau_{\mathcal{L}}$' in black) saves very few route requests, whereas our techniques (e.g., lower/upper bounds to be discussed below) provide the major savings in route requests. Furthermore, Ref. [31] has not considered the expiry time requirement as in our work.

To reduce the number of route requests while providing accurate results, we combine information across multiple routes in the log to derive tight lower/upper bounding travel times. We also propose effective techniques to compute such bounds efficiently. Moreover, we examine the effect of different orderings for issuing route requests on saving route requests. And we study how to parallelize route requests in order to reduce the query response time further.

In the following, we first review related work in Section 2. Then, we describe the system architecture and our objectives in Section 3. Our contributions are:

- Combine information across multiple routes in the log to derive lower/upper bounding travel times, which support efficient and accurate range and KNN search (Section 4);
- Develop heuristics to parallelize route requests for reducing the query response time further (Section 5);
- Evaluate our solutions on a real route API and also on a simulated route API for scalability tests (Section 6).

Finally, we conclude this paper in Section 7.

2 Related Work

2.1 Query Processing on Road Networks

Indexing on road networks have been extensively studied in the literature [19], [20], [22], [26], [28]. Various shortest path indices [19], [20], [28] have been developed to support shortest path search efficiently. Papadias et al. [26] study how to process range queries and KNN queries over points-of-interest (POIs), with respect to shortest path distances on a road network. The evaluation of range queries and KNN queries can be further accelerated by specialized indices [19], [22], [28].

In our problem scenario, query users require accurate results that are computed with respect to live traffic information. All the above works require the LBS to know the weights (travel times) of all road segments. Since the LBS lacks the infrastructure for monitoring road traffic, the above works are inapplicable to our problem. Some works [16], [21] attempt to model the travel times of road segments as time-varying functions, which can be extracted from historical traffic patterns. These functions may capture the effects of periodic events (e.g., rush hours, weekdays). Nevertheless, they still cannot reflect live traffic information, which can be affected by sudden events, e.g., congestions, accidents and road maintenance.

Landmark [24], [25], [27] and distance oracle [29] can be applied to estimate shortest path distance bounds between two nodes in a road network, which can be used to prune irrelevant objects and early detect results. The above works are inapplicable to our problem because they consider constant travel times on road segments (as opposed to live traffic). Furthermore, in this paper, we propose novel lower/upper travel time bounds derived from both the road network and the information of previously obtained routes; these bounds have not been studied before.

2.2 Querying on Online Route APIs

Online route APIs. An online route API [4], [7] has access to current traffic information [6]. It takes a route request as input and then returns a route along with travel times on

route segments. The example below illustrates the request and response format of Google Directions API [7]. Bing Maps API [4] uses a similar format.

HTTP request

```
http://maps.googleapis.com/maps/api/directions/xml?
origin=44.94033,-93.22294&destination=44.94198,-93.23722
mode=driving
```

XML response

```
<step>
   <start_location>
      <lat>44.9403300</lat> <lng>-93.2229400</lng>
   </start_location>
   <end_location>
      <lat>44.9395900</lat> <lng>-93.2229500</lng>
   </end_location>
   <duration> <value>8</value> </duration>
</step>
..... remaining steps .....
```

The request is an HTTP query string, whose parameters contain the origin and destination locations in latitude-longitude, as well as the travel mode. In this example, the origin is at $(44.94033, -93.22294)$, the destination is at $(44.94198, -93.23722)$, and the user is at 'driving' mode.

The response is an XML document that stores a sequence of route segments from the origin to the destination. Each segment, enclosed by `<step>` tags, contains its endpoints and its travel time by driving (see the `<duration>` tags). The segment in this example takes 8 seconds to travel. We omit the remaining segments here for brevity. Besides, the XML response contains the total travel time on this route (the sum of travel times on all segments).

Query processing algorithms. Thomsen et al. [31] study the caching of shortest paths obtained from online route APIs. They exploit the optimal subpath property [15] on cached paths to answer shortest path queries. As we discussed in the introduction and verified in experiments, this property cannot significantly reduce the number of route requests in our problem. Also, they have not studied the processing of range/KNN queries, the lower/upper bound techniques developed in this paper, as well as the accuracy of query results.

The framework SMashQ [32], [33] is the closest work to our problem. It enables the LBS to process KNN queries by using online route APIs. To reduce the number of route requests (for processing queries), SMashQ exploits the maximum driving speed $\mathbb{V}_{MAX}$ and the static road network G_S (with only distance information) stored at the LBS. Upon receiving a KNN query from user q, the LBS first retrieves K objects with the smallest network distance from q and issues route requests for them. Let γ be the K^{th} smallest current travel time (obtained so far). The LBS inserts into a candidate set C the objects whose network distance to q is within $\gamma \cdot \mathbb{V}_{MAX}$. Next, SMashQ groups the points in C to road junctions, utilizes historical statistics to order the road junctions, and then issues route requests for junctions in above order. Compared with our work, SMashQ does not utilize route log to derive exact travel times nor lower/upper bounds to boost the query performance of the LBS. As we will show in the experiments, even if we extend SMashQ to use a route log and apply the optimal subpath property [15] [31] to save route requests, it still incurs much more route requests than our proposed method.

Efficient algorithms [23], [30] have been developed for KNN search on data objects with respect to generic distance functions. It is expensive to compute the exact distance from a query object q to a data object p (e.g., using exact spatial object geometry). On the other hand, it is cheap to compute the lower/upper bound distance from q to p (e.g., using bounding rectangle). Seidl et al. [30] propose a KNN search algorithm that fetches the optimal number of objects from the dataset $\mathcal{P}$. Kriegel et al. [23] further improve the algorithm by utilizing both lower and upper bound distances. These generic solutions [23], [30] are applicable to our problem; however, they do not exploit the rich information of routes that are specific in our problem. In our problem, the exact route from q to p reveals not only the current travel time to p, it may also provide the current travel times to other objects p' on the route, and may even offer tightened lower/upper bounds of travel times to other objects, as we will illustrate in Section 4.

3 Problem Statement

In this section, we first describe the system architecture and then formulate the objectives of our problem.

System architecture and notations. In this paper, we adopt the system architecture as depicted in Fig. 3. It consists of the following entities:

- **Online Route API**. Examples are: Google / Bing route APIs [7] [4]. Such API computes the shortest route between two points on a road network, based on live traffic [6]. It has the latest road network G with live travel time information.
- **Mobile User**. Using a mobile device (smartphone), the user can acquire his current geo-location q and then issue queries to a location-based server. In this paper, we consider range and KNN queries based on live traffic.
- **Location-Based Service/Server (LBS)**. It provides mobile users with query services on a dataset $\mathcal{P}$, whose POIs (e.g., restaurants, cafes) are specific to the LBS's application. The LBS may store a road network G with edge weights as spatial distances, however G cannot provide live travel times. In case $\mathcal{P}$ and G do not fit in main memory, the LBS may store $\mathcal{P}$ as an R-tree and store the G as a disk-based adjacency list [26].

We then define route, travel time, and queries formally.

DEFINITION 1 (Route and travel time). *The route $\psi_t(v_s, v_d)$ between v_s and v_d, obtained from route API at timestamp t, is a sequence of pairs $\{ \langle v_i, \tau_t(v_s, v_i)\rangle : v_i \in \psi_t \}$. Each pair stores a node v_i and its travel time $\tau_t(v_s, v_i)$ from the source v_s. Let $\tau_t(v, v')$ be the (shortest) travel time between two locations v and v' (obtained at timestamp t).*

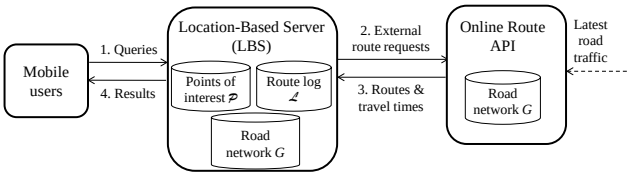


Fig. 3. System architecture

DEFINITION 2 (Query results).
Let q be a query point and t_{now} be the current time. Given q and a travel time limit T, the result set of range query *is:* $R = \{\, p \in \mathcal{P} : \tau_{t_{now}}(q,p) \leq T \,\}$.
Given q and a result size K, the result set of KNN query *is:* $R = \{\, p \in \mathcal{P} : \tau_{t_{now}}(q,p) \leq \tau_{t_{now}}(q,p'), p' \in \mathcal{P} - R \,\}$ *with size K.*

As discussed in the introduction, queries in real applications may involve filters on (i) non-spatial features (e.g., quality, price) of $\mathcal{P}$ as well as (ii) live travel times from the query point q to POIs in $\mathcal{P}$. These queries cannot be solved by the LBS alone nor an online map (e.g., Google Map) alone. LBS lacks access to live traffic information (i.e. travel times), whereas the dataset $\mathcal{P}$ maintained by the LBS is not available to an online map.

The flow of the system is as follows. A user first issues a query to the LBS via his/her mobile client (Step 1). The LBS then determines the necessary route requests for the query and submits them to the route API (Step 2). Next, the route API returns the corresponding routes back to the LBS (Step 3). Having such information, the LBS can compute the query results and report them back to the user (Step 4). As a remark, our system architecture is similar to [32], except our LBS maintains a route log $\mathcal{L}$ and some additional attributes with edges on G (to be elaborated soon).

Objective and our approach. Our objective is to reduce the response time of queries (i.e., requirement $\mathcal{R}2$) while offering accurate query results (i.e., requirement $\mathcal{R}1$). It is important to minimize the number of route requests issued by the LBS because route requests incur considerable time (see introduction).

As observed in Fig. 2, travel times change slightly within a short duration (e.g., 10 minutes). Based on this observation, we approximate the travel time (from v to v') at current time t_{now} as the travel time obtained from a route API at an earlier time t':

ASSUMPTION 1 (Temporal approximation). *For any locations v, v', we have: $\tau_{t_{now}}(v,v') \approx \tau_{t'}(v,v')$ if $t' \geq t_{now} - \delta$.*

This approximation enables the LBS to save route requests significantly, while still providing high accuracy. Specifically, at the LBS, we employ a log $\mathcal{L}$ of routes that were requested from an online route API within the last δ minutes.

Like in [32] [33], we assume that the road network G used in LBS is the same with that used in route service. This is feasible when the LBS can obtain accurate maps from the government [12], route service providers [9] or their map suppliers [3]. However, when the LBS cannot have access to the same G as the route service, we will discuss the applicability of our techniques in Section 4.5.

To achieve low response time, we will exploit the route log and road network G to reduce the number of external route requests (issued to online route API) for answering queries (Section 4). We will also parallelize route requests (Section 5) to further reduce the response time.

4 QUERY PROCESSING

This section presents our approach Route-Saver for answering queries efficiently. First, we discuss the maintenance of the time-tagged road network G and the route log $\mathcal{L}$ (Section 4.1). Then, we exploit G and $\mathcal{L}$ to design effective bounds for travel times (Section 4.2). Next, we present our algorithms for answering range and KNN queries in Sections 4.3, 4.4 respectively. Finally, we discuss the applicability of our techniques when no local maps are available in Section 4.5.

In subsequent discussion, we drop the subscript t in $\tau_t(v,v')$ as we only use valid routes (and their travel times).

4.1 Maintenance of Structures at LBS

Conservative travel time bounds. Given an edge $e(v,v')$, we define $c\omega^-(e)$ and $c\omega^+(e)$ as *conservative* lower-bound and upper-bound of travel time on e, respectively. Observe that the lower-bound $c\omega^-(e)$ is limited by the Euclidean distance of e and the maximum driving speed $\mathbb{V}_{MAX}$:

$$c\omega^-(e) = dist(e)/\mathbb{V}_{MAX} \quad (1)$$

On the other hand, the upper-bound is $c\omega^+(e) = \infty$ because the travel time on e can be arbitrarily long in case of traffic congestion.

Structures. We employ a route log $\mathcal{L}$ and a time-tagged network G in the LBS.

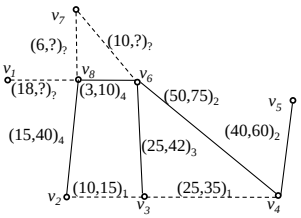
The *route log* $\mathcal{L}$ stores all routes obtained from an online route API within the last δ time units, as described in Section 3. Recall from Definition 1 that the timestamp of a route $\psi_t(v,v')$ is indicated by its subscript t. Assume that we use $\delta = 2$ in Fig. 4a. At time $t_{now} = 4$, $\mathcal{L}$ keeps the routes obtained during time 2–4.

To support query operations efficiently, we summarize the travel times of edges in $\mathcal{L}$ into a time-tagged network G. Specifically, each edge e in G is tagged with a tuple $(c\omega^-(e), \omega(e))_{\mu(e)}$, where $c\omega^-(e)$ is the conservative lower-bound travel time on e (Eqn. 1), $\omega(e)$ is the exact travel time stored in $\mathcal{L}$, and $\mu(e)$ is the last-update timestamp for $\omega(e)$. We call an edge e to be *valid* if its last-update timestamp $\mu(e)$ satisfies $\mu(e) \geq t_{now} - \delta$.

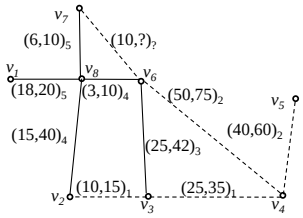
As an example, consider the time-tagged network G at current time $t_{now} = 4$ in Fig. 4b. Assume that the expiry time is $\delta = 2$. We draw valid edges by solid lines and invalid edges by dotted lines. For the solid edge (v_3, v_6), the tuple $(25, 42)_3$ means that its conservative lower bound $c\omega^-(v_3,v_6)$ is 25, its exact travel time $\omega(v_3,v_6)$ is 42, and its last-update timestamp $\mu(v_3,v_6)$ is 3. The dotted edge

Route ID	Route Content	t_{now}=4	t_{now}=5	t_{now}=6
$\psi_1(v_2, v_4)$	$(v_2, 0)$, $(v_3, 15)$, $(v_4, 50)$			
$\psi_2(v_5, v_6)$	$(v_5, 0)$, $(v_4, 60)$, $(v_6, 135)$	√		
$\psi_3(v_3, v_6)$	$(v_3, 0)$, $(v_6, 42)$	√	√	
$\psi_4(v_2, v_6)$	$(v_2, 0)$, $(v_8, 40)$, $(v_6, 50)$	√	√	√
$\psi_5(v_1, v_7)$	$(v_1, 0)$, $(v_8, 20)$, $(v_7, 30)$		√	√
$\psi_6(v_2, v_3)$	$(v_2, 0)$, $(v_3, 15)$			√

(a) Route Log $\mathcal{L}$ (at different times)



(b) G at $t_{now} = 4$



(c) G at $t_{now} = 5$

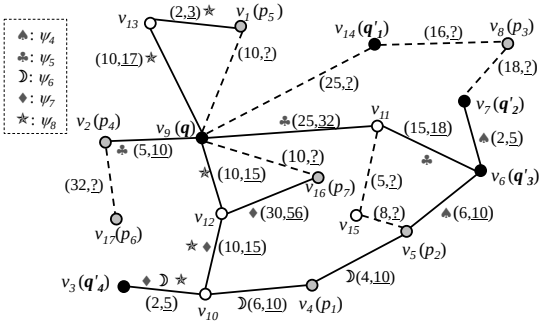
Fig. 4. Example route log $\mathcal{L}$ and time-tagged road network G, with expiry time $\delta = 2$; solid edges have valid travel times

Point	Loc.
p_1	(x_1, y_1)
p_2	(x_2, y_2)
p_3	(x_3, y_3)
p_4	(x_4, y_4)
p_5	(x_5, y_5)
p_6	(x_6, y_6)
p_7	(x_7, y_7)

(a) dataset $\mathcal{P}$

Route ID	Route Content
$\psi_4(q'_2, p_2)$	$(v_7, 0)$, $(v_6, 5)$, $(v_5, 15)$
$\psi_5(q'_3, p_4)$	$(v_6, 0)$, $(v_{11}, 18)$, $(v_9, 50)$,$(v_2, 60)$
$\psi_6(q'_4, p_2)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_4, 15)$, $(v_5, 25)$
$\psi_7(q'_4, p_7)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_{12}, 20)$, $(v_{16}, 76)$
$\psi_8(q'_4, p_5)$	$(v_3, 0)$, $(v_{10}, 5)$, $(v_{12}, 20)$, $(v_9, 35)$, $(v_{13}, 52)$, $(v_1, 55)$

(b) route log $\mathcal{L}$



(c) Road network G with travel time information

Fig. 5. Data stored at the LBS, at $t_{now} = 9$ ($\delta = 5$); each edge is tagged with ($c\omega^-(e)$, $\underline{\omega(e)}$)

TABLE 1
Example travel time information (for user q), $t_{now} = 9$

Point	$p.\tau_c^-$	$p.\tau_G^-$	$p.\tau_G$	$p.\tau_G^+$	$p.\tau_I^-$
p_1 (v_4)	26	40	40	40	20
p_2 (v_5)	30	45	NIL	50	40
p_3 (v_8)	41	41	NIL	NIL	NIL
p_4 (v_2)	5	10	10	10	NIL
p_5 (v_1)	10	10	NIL	20	NIL
p_6 (v_{17})	37	42	NIL	NIL	NIL
p_7 (v_{16})	10	10	NIL	71	41

(v_2, v_3) is invalid since its timestamp $\mu(v_2, v_3) = 1$ is less than $t_{now} - \delta = 4 - 2 = 2$.

Maintenance. We then discuss how to maintain the route log $\mathcal{L}$ and the time-tagged road network G.

To support efficient lookup on $\mathcal{L}$, we employ inverted lists of routes for each node [31]. Specifically, each inverted list of node v stores a list of route IDs that contain v. The insertion/deletion of a route can be implemented to take $O(|\psi|)$ time, where $|\psi|$ is the number of vertices on a route.

At time t_{now}, we remove from $\mathcal{L}$ the routes ψ_t having $t < t_{now} - \delta$ (i.e., expired). E.g., at $t_{now} = 5$, we remove $\psi_2(v_5, v_6)$ from $\mathcal{L}$ (see Fig. 4a). Also, we update the inverted lists for $v_4, v_5, v_6 \in \psi_2(v_5, v_6)$. We need not update G now because it stores the last-update timestamps of edges.

When we retrieve a route $\psi_{t_{now}}$ from online route API, e.g., $\psi_5(v_1, v_7) : v_1 \rightarrow v_8 \rightarrow v_7$, we insert it into $\mathcal{L}$ (see Fig. 4a), and update the inverted lists for nodes v_1, v_7, v_8. For the edges on $\psi_5(v_1, v_7)$, e.g., (v_1, v_8), (v_8, v_7), we update their $\omega(e)$ and $\mu(e)$ in G (see Fig. 4c).

4.2 Exact Travel Times and Their Bounds

In this section, we exploit the time-tagged road network G and the route log $\mathcal{L}$ to derive lower and upper bounds of travel times for data points. As we will elaborate soon, these bounds enable us to save route requests during query processing.

Before presenting these techniques, we first show an example of data stored at LBS (see Fig. 5). Besides G and $\mathcal{L}$, the LBS also stores a dataset $\mathcal{P}$ (points p_j with locations). Assume the current time $t_{now} = 9$ and the expiry time $\delta = 5$. The route log $\mathcal{L}$ contains only valid routes (not yet expired). For the time-tagged network G (see Fig. 5c), solid edges are valid while dotted edges are not. Each edge e is tagged with $c\omega^-(e)$ and $\omega(e)$ (underlined), and the icons of routes via e (if any). For clarity, we omit $\mu(e)$ (i.e., the last-update timestamp) of edges.

We first introduce the concept of travel time bounds:

DEFINITION 3 (Travel time bounds). *Given a query point q and a data point p, we denote $p.\tau^-$ and $p.\tau^+$ as a lower bound and an upper bound of the exact travel time $\tau(q, p)$. Specifically, we require that $p.\tau^- \leq \tau(q, p) \leq p.\tau^+$. For convenience, we may denote $\tau(q, p)$ by $p.\tau$.*

As an example, consider a data point p and a range query with travel time limit T. The upper-bound time $p.\tau^+$ helps detect true results early. If p satisfies $p.\tau^+ \leq T$, then p must be a result. The lower-bound time $p.\tau^-$ enables pruning unpromising points. If p satisfies $p.\tau^- > T$, then p cannot be a result. In either case, we save a route request for p. Observe that a tight upper bound should be as small as possible because it is more likely to satisfy $p.\tau^+ \leq T$. Similarly, a tighter lower bound should be as large as possible to satisfy $p.\tau^- > T$. Techniques discussed below aim to derive tight bounding travel times for data points.

Conservative lower-bound. Let $spt_{c\omega^-}(q, p)$ be the shortest travel time from q to p defined on the edge weight $c\omega^-(e)$ (see Eqn.1). The conservative lower-bound travel time from q to p is:

$$p.\tau_c^- = spt_{c\omega^-}(q, p) \tag{2}$$

We take the point p_3 in Fig. 5c as an example. With respect to the weight $c\omega^-(e)$, the shortest path from q to p_3 is $q \rightarrow v_{14} \rightarrow p_3$, with length $c\omega^-(q, v_{14}) + c\omega^-(v_{14}, p_3) =$

41. Table 1 shows $p.\tau_c^-$ for each data point p.

Bounding travel times based on ω^+ and ω^-. Recall that the exact travel time $\tau(q,p)$ is defined as the shortest travel time based on live traffic information. Thus, we have: $\tau(q,p) = spt_{\omega^*}(q,p)$, where $\omega^*(e)$ denotes the current travel time for an edge e. We use the notations $\tau(q,p)$ and $spt_{\omega^*}(q,p)$ interchangeably in the following discussion.

Our idea is to define upper-bound weight $\omega^+(e)$ and lower-bound weight $\omega^-(e)$ for each edge e, by using the information in the time-tagged road network G.

$$\omega^+(e) = \begin{cases} \omega(e) & \text{if } \mu(e) \geq t_{now} - \delta \\ \infty & \text{otherwise} \end{cases} \quad (3)$$

$$\omega^-(e) = \begin{cases} \omega(e) & \text{if } \mu(e) \geq t_{now} - \delta \\ c\omega^-(e) & \text{otherwise} \end{cases} \quad (4)$$

Note that $\omega^*(e)$ is unknown to the LBS in general. If $\mu(e) \geq t_{now} - \delta$, then the last-update travel time $\omega(e)$ (in above equations) serves as an approximation of $\omega^*(e)$, due to Assumption 1.

With these edge weights, we establish the upper and lower bounds for travel times from q to p (Lemma 1).

LEMMA 1 (bounding travel times on ω^+, ω^-).
Let $spt_{\omega^+}(q,p)$ be the shortest travel time from q to p with respect to the edge weight $\omega^+(e)$, on a time-tagged road network G.
Similarly, let $spt_{\omega^-}(q,p)$ be the shortest travel time with respect to the edge weight $\omega^-(e)$. With Assumption 1, we have: $spt_{\omega^-}(q,p) \leq spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$.

Proof: We first aim to prove: $spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$. Let SP^* and SP^+ be the shortest path between q and p defined on the edge weights $\omega^*(e)$ and $\omega^+(e)$, respectively. According to Eqn. 3, we have $\omega(e) \leq \omega^+(e)$. By Assumption 1, we approximate $\omega^*(e)$ by $\omega(e)$. Thus, we have $\omega^*(e) \leq \omega^+(e)$. Applying it on all edges on SP^+, we obtain: $\sum_{e\in SP^+} \omega^*(e) \leq \sum_{e\in SP^+} \omega^+(e)$ —(▲). By the definition of shortest path on the edge weight $\omega^*(e)$, the travel time of SP^* is no larger than that of SP^+. Thus, we have: $\sum_{e\in SP^*} \omega^*(e) \leq \sum_{e\in SP^+} \omega^*(e)$ —(▼). Combining inequalities (▲) and (▼), we obtain: $\sum_{e\in SP^*} \omega^*(e) \leq \sum_{e\in SP^+} \omega^+(e)$. Therefore, $spt_{\omega^*}(q,p) \leq spt_{\omega^+}(q,p)$.

The proof for $spt_{\omega^-}(q,p) \leq spt_{\omega^*}(q,p)$ is similar to the above proof, except that we apply Eqn. 4 instead. □

In subsequent discussion, we represent the bounds $spt_{\omega^+}(q,p)$ and $spt_{\omega^-}(q,p)$ by $p.\tau_G^+$ and $p.\tau_G^-$ respectively. Observe that we can compute $p.\tau_G^+$ (or $p.\tau_G^-$) for all points efficiently by running the Dijkstra algorithm using edge weight $\omega^+(e)$ (or $\omega^-(e)$).

Table 1 shows the upper-bound $p.\tau_G^+$ and lower-bound $p.\tau_G^-$ for all data points. We take the candidate p_2 in Fig. 5c as an example. After running Dijkstra on G using $\omega^-(e)$, the shortest path from q to p_2 is $q \to v_{11} \to v_{15} \to p_2$ with the length as $\omega(q,v_{11}) + c\omega^-(v_{11},v_{15}) + c\omega^-(v_{15},p_2) = 45$. After running Dijkstra on G using $\omega^+(e)$, the shortest path from q to p_2 as $q \to v_{12} \to v_{10} \to v_4 \to p_2$ with length as $\omega(q,v_{12}) + \omega(v_{12},v_{10}) + \omega(v_{10},v_4) + \omega(v_4,p_2) = 50$.

Condition for exact travel time. When a point p satisfies certain condition (see Lemma 2), its lower-bound travel time ($p.\tau_G^-$) serves as its exact travel time from q (denoted by $p.\tau_G$). In this case, we save a route request for p regardless of the value of $p.\tau_G$.

LEMMA 2 (Road network exact travel time).
Let SP^- (with travel time $spt_{\omega^-}(q,p)$) be the shortest path from q to p with respect to the edge weight $\omega^-(e)$, on a time-tagged road network G. If each edge on SP^- satisfies $\mu(e) \geq t_{now} - \delta$, then we have: $spt_{\omega*}(q,p) = spt_{\omega^-}(q,p)$.

Proof: If an edge e satisfies $\mu(e) \geq t_{now} - \delta$ (i.e., valid edge), then we have: $\omega^-(e) = \omega^+(e) = \omega(e)$ $(= \omega^*(e))$. Applying this to each edge on SP^-, we obtain: $spt_{\omega^-}(q,p) = \sum_{e\in SP^-} \omega^-(e) = \sum_{e\in SP^-} \omega^*(e)$.

We then claim that SP^- is the shortest path with respect to the edge weight ω^*. For the sake of contradiction, assume there exists a path SP' shorter than SP^- on edge weight ω^*. Then, SP' has a shorter travel time than $spt_{\omega^-}(q,p)$, contradicting the fact that $spt_{\omega^-}(q,p)$ is a lower-bound. Thus, this lemma is proved. □

Table 1 lists the exact travel time $p.\tau_G$ of all data points. Take the candidate p_1 in Fig. 5c as an example. With respect to edge weight $\omega^-(e)$, the shortest path from q to p_1 is $q \to v_{12} \to v_{10} \to p_1$. Since each edge on this path has valid exact travel time, thus we obtain: $p_1.\tau_G = \omega(q,v_{12}) + \omega(v_{12},v_{10}) + \omega(v_{10},v_4) = 40$.

Tightening the lower-bound using route log. As we will illustrate soon, the lower-bound $p.\tau_G^-$ (derived from edge weight $\omega^-(e)$) may not be tight for some data points.

Next, we utilize a shared node $\imath$ among routes in $\mathcal{L}$ to derive another lower-bound travel time for candidates (see Lemma 3). We denote this lower-bound travel time as $p.\tau_I^- = |\tau(q,\imath) - \tau(p,\imath)|$.

LEMMA 3 (Route log lower-bound travel time). *Let ψ_i, ψ_j be two different routes in the route log $\mathcal{L}$ such that they share a node $\imath$. If q, p fall on ψ_i, ψ_j respectively, then we have: $|\tau(q,\imath) - \tau(p,\imath)| \leq \tau(q,p)$ (i.e. $spt_{\omega^*}(q,p)$).*

Proof: For the sake of contradiction, assume that: $\tau(q,p) < |\tau(q,\imath) - \tau(p,\imath)|$. For case I ($\tau(q,\imath) \geq \tau(p,\imath)$), we obtain: $\tau(q,p) + \tau(p,\imath) < \tau(q,\imath)$. This contradicts with that $\tau(q,\imath)$ is the shortest between q and $\imath$. For case II ($\tau(q,\imath) < \tau(p,\imath)$), we obtain: $\tau(p,q) + \tau(q,\imath) < \tau(p,\imath)$. This contradicts with that $\tau(p,\imath)$ is the shortest between p and $\imath$. Thus, the lemma is proved. □

Since ψ_i, ψ_j are routes in $\mathcal{L}$, so the values $\tau(q,\imath), \tau(p,\imath)$ can be directly obtained from $\mathcal{L}$ according to *optimal subpath property* [15]. Through using the inverted node index [31] of $\mathcal{L}$, we can efficiently retrieve the subset of routes which contains q (i.e. L_q) and the corresponding subset for each remaining candidate point p (i.e. L_p). Then, we can identify the shared node between routes in L_q and L_p, and use it to calculate $p.\tau_I^-$.

Continuing with the example, we show how to derive

$p_7.\tau_I^-$ of point p_7 (in Fig. 5c). First, we find the subset of routes that contain q, i.e., $L_q = \{\psi_5, \psi_8\}$. Then, we find the subset of routes that contain p_7, i.e., $L_{p_7} = \{\psi_7\}$. Next, we identify a shared node between L_q and L_{p_7}, which is the node v_{12} on the routes $\psi_8 \in L_q$ and $\psi_7 \in L_{p_7}$. With these routes, we obtain these exact travel times: $\tau(q, v_{12}) = 15$ and $\tau(p_7, v_{12}) = 56$. Thus, we derive $p_7.\tau_I^- = |\tau(p_7, v_{12}) - \tau(q, v_{12})| = |56 - 15| = 41$. Observe that this bound $p_7.\tau_I^- = 41$ is tighter than the bound $p_7.\tau_G^- = 10$.

Nevertheless, the bound $p.\tau_I^-$ can be looser or unavailable for some points, e.g., p_1, p_3 in Table 1. So, we combine bounds τ_G^- and $p.\tau_I^-$ into a tighter lower bound for p:

$$p.\tau^- = \max\{p.\tau_G^-, p.\tau_I^-\} \quad (5)$$

4.3 Range Query Algorithm

In this section, we present our Route-Saver algorithm for processing a range query (q, T). It applies the travel time bounds discussed above to reduce the number of route requests. To guarantee the accuracy of returned results, it removes all expired routes ψ_t in $\mathcal{L}$. The algorithm first conducts a distance range search $(q, T \cdot \mathbb{V}_{MAX})$ for $\mathcal{P}$ on G [26] to obtain a set C of candidate points. Algorithm 1 consists of two phases to process the candidate points in C and store the query results in the set R.

The first phase (Lines 4–17) aims to shrink the candidate set C, so as to reduce the number of route requests to be issued in the second phase. First, we execute Dijkstra on G two times, using edge weight $\omega^-(e)$ and $\omega^+(e)$ respectively. Then, we obtain the bounds $p.\tau_G^+$, $p.\tau_G^-$ and $p.\tau_G$ for every candidate $p \in C$. If $p.\tau_G^+ \leq T$ or $p.\tau_G \leq T$, then p must be a true result so we place it into R. If $p.\tau_G^- > T$, then p cannot become a result and it gets removed from C. Next, for each candidate p remaining in C, we compute its exact travel time $p.\tau_{\mathcal{L}}$ using optimal subpath property in $\mathcal{L}$ [15] [31], and use $p.\tau_{\mathcal{L}}$ to detect true result. Moreover, we derive the lower bound travel time $p.\tau_I^-$ using route log $\mathcal{L}$ for pruning.

In the second phase, we issue route requests for the remaining candidates in C, based on a certain ordering. We will elaborate the effect of candidate ordering at the end of this section. For the moment, suppose that we examine candidates in ascending order, i.e., pick a candidate $p \in C$ with the minimum $p.\tau^-$ (Line 19). Next, we issue a route request for p and then insert the returned route $\psi_{t_{now}}$ into the route log $\mathcal{L}$. For each edge on the returned route $\psi_{t_{now}}$, we update its $\omega(e)$ and $\mu(e)$ accordingly.

This route provides not only the exact travel time for p, but also potential information for updating the bounds for other candidate $p' \in C$. We remove p' from C if (i) it cannot become result, i.e., $p'.\tau^- > T$, or (ii) its exact travel time $p'.\tau_{\mathcal{L}}$ is known (i.e., p' lies on route $\psi_{t_{now}}$). In case $p'.\tau_{\mathcal{L}} \leq T$, we insert p' into R. Whenever C becomes empty, the loop terminates and the algorithm reports R as the result set.

Example. Consider the range query at q with $T = 40$ in Fig. 5c. We illustrate the running steps of Route-Saver in Table 2. Entries without values are labeled as '/'.

Algorithm 1 Route-Saver Algorithm for Range Queries

```
function Route-Saver-RANGE ( Query (q, T), Dataset P )
        ▷ system parameters: time-tagged graph G, route log L, expiry time δ
1:  Remove from the log L any route ψ_t with t < t_now − δ
2:  Create a result set R ← ∅
3:  Cand. set C ← range search (q, T · V_MAX) for P on G                ▷ By [26]
4:  Run Dijkstra for q on G using ω+(e) and ω−(e) to retrieve
    p.τ_G+, p.τ_G−, p.τ_G                  ▷ Phase 1:detect results,prune objects
5:  for each p ∈ C do                                  ▷ use time-tagged graph G
6:      if p.τ_G is known or p.τ_G− > T or p.τ_G+ ≤ T then
7:          Remove p from C
8:      if p.τ_G ≤ T or p.τ_G+ ≤ T then
9:          Insert p into R
10: for each p ∈ C do                                  ▷ use route log L
11:     if ∃ route ψ ∈ L such that ψ contains p and q then
12:         Compute p.τ_L                  ▷ optimal subpath property [15] [31]
13:         if p.τ_L is known and p.τ_L ≤ T then
14:             Insert p into R
15:     Compute p.τ− as max{p.τ_G−, p.τ_I−}
16:     if p.τ− > T or p.τ_L is known then
17:         Remove p from C
18: while C is not empty do                  ▷ Phase 2: Issue route requests
19:     Pick an object p ∈ C with minimum p.τ−                    ▷ ordering
20:     Route ψ_{t_now} ← RouteRequest(q, p)                  ▷ call external API
21:     Insert ψ_{t_now} into L;   Update ω(e), μ(e) in G for e ∈ ψ_{t_now}
22:     Update p.τ_L for all p on ψ_{t_now}    ▷ optimal subpath property [15] [31]
23:     Run incremental Dijkstra to update all p.τ−                   ▷ By [14]
24:     for each p′ ∈ C do
25:         if p′.τ− > T or p′.τ_L is known then
26:             Remove p′ from C
27:         if p′.τ_L ≤ T then
28:             Insert p′ into R
29: Return R
```

TABLE 2
Range/KNN query example for Route-Saver, $T = 40$

	$p.\tau_G^-$	$p.\tau_G$	$p.\tau_G^+$	$p.\tau_{\mathcal{L}}$	$p.\tau_I^-$	$p.\tau$ by route API	Is result?
p_1	40	40	40	/	/	/	√
p_2	45	/	50	/	/	/	×
p_3	41	/	/	/	/	/	×
p_4	10	10	10	/	/	/	√
p_5	10	/	20	/	/	/	√
p_6	42	/	/	/	/	/	×
p_7	10	/	71	/	41	/	×

Suppose that $\mathbb{V}_{MAX} = 110\ km/h$. First, we do a range search at q with distance $T \cdot \mathbb{V}_{MAX}$, and obtain the candidate set $C = p_1, p_2, p_4, p_5, p_6, p_7$. Note that further away points (e.g. p_3) are not in C. Then, we derive the $p.\tau_G^+, p.\tau_G^-$ and $p.\tau_G$ using the time-tagged road network G, as shown in the first three columns of Table 2. Candidates p_1, p_4, p_5 are inserted into the result set R, since their exact or upper-bound travel times are smaller than $T = 40$. Candidates p_2, p_6 are pruned with lower bounds larger than $T = 40$. Then, we compute the lower bound for the remaining candidate using $\mathcal{L}$: $p_7.\tau_I^- = 41$, and p_7 is pruned. We skip the second phase as the candidate set becomes empty. Thus, the algorithm returns $R = \{p_1, p_4, p_5\}$ to the user. In this example, Route-Saver issues 0 route request.

Candidate ordering and its analysis. This section studies the effect of candidate orderings on the cost of Algorithm 1, i.e., the number of route requests issued. Various orderings can be used for processing the candidates (in phase 2). We consider two orderings for picking the next candidate $p \in C$ (at Line 19):

Ascending order (ASC): Pick a candidate with the mini-

mum $p.\tau^-$. This order is the same as in Algorithm 1.

Descending order (DESC): Pick a candidate with the maximum $p.\tau^-$. The rationale is that longer routes are more likely to cover other candidates and thus save route requests for them.

We proceed to analyze the number of route requests incurred by ASC and DESC.

For simplicity, we assume that the underlying graph is a unit-weight grid network (in 2D space). Let q be at the origin $(0, 0)$ and T be the travel time limit. Let α be the data density, i.e., the probability that a node contains a point. Let the layer i be the set of nodes whose travel times from q equal to i. Observe that, in layer i, there are $4i$ nodes and $4\alpha i$ candidates. Summing up this from layer 1 to layer T, the number of candidates is: $Cand(\alpha, T) = \sum_{i\in[1,T]} 4\alpha i \approx 2\alpha T^2$

ASC issues route requests for candidates in ascending order of their layers. Thus, it cannot save any route request for candidates. The cost of ASC is:

$$Cost_{ASC}(\alpha, T) = Cand(\alpha, T) \approx 2\alpha T^2$$

On the other hand, DESC issues route requests for candidates in descending order of their layers. Consider a node v in the layer i. Note that the number of candidates from layer $i+1$ to layer T is: $2\alpha T^2 - 2\alpha i^2 = 2\alpha(T^2 - i^2)$. If the route from q to any of these candidates passes v, then we can save a route request for v. Since there are $4 \cdot i$ possible locations for v, the probability of saving a route request for v is: $\max\{\frac{2\alpha(T^2-i^2)}{4i}, 1\} = \max\{\frac{\alpha(T^2-i^2)}{2i}, 1\}$. Thus, the cost of DESC is: $Cost_{DESC}(\alpha, T) = \sum_{i\in[1,T]} 4\alpha i \cdot (1 - \max\{\frac{\alpha(T^2-i^2)}{2i}, 1\})$ To simplify the above equation, we find the maximum value for i such that: $\frac{\alpha(T^2-i^2)}{2i} \geq 1$. By solving this quadratic inequality, we get: $i \leq \frac{\sqrt{1+1\alpha^2T^2}-1}{\alpha}$. When $T > \frac{1}{\alpha}$, the cost of DESC is upper-bounded by:

$$Cost_{DESC}(\alpha, T) \leq 4T$$

In summary, DESC incurs a much lower cost than ASC.

4.4 KNN Query Algorithm

In this section, we extend our Route-Saver algorithm for processing KNN queries. We will also examine suitable orderings for processing candidates.

Unlike range queries, KNN queries do not have a (fixed) travel time limit T for obtaining a small candidate set. Instead, we first compute a (temporary) result set R so that it contains K candidates with the smallest $p.\tau_G^+$ or $p.\tau_G$. Recall that we can obtain these bounds/values for all candidates efficiently by two Dijkstra traversal on G. Let γ be the largest $p.\tau_G^+$ or $p.\tau_G$ in R. Having this value γ, we can prune each candidate p that satisfies $p.\tau^- > \gamma$, as it cannot become the result.

Algorithm 2 is the pseudo-code of our KNN algorithm. First, we initialize the candidate set C with the dataset $\mathcal{P}$, insert K dummy pairs (with ∞ travel time) into the result set R, and set γ to the largest travel time in R. The algorithm consists of three phases. In the first phase, it obtains γ by using the idea discussed above. In the second phase, it prunes candidates whose lower bounds or exact times are larger than γ. In the third phase, it examines the candidates according to a certain order and issues route requests for them. The algorithm terminates when the candidate set contains exactly K objects, and then reports them as query results.

Algorithm 2 Route-Saver Algorithm for KNN Queries

```
function Route-Saver-KNN ( Query (q, K), Dataset P )
        ▷ system parameters: time-tagged graph G, route log L, expiry time δ
1: Remove from the log L any route ψ_t with t < t_now − δ
2: Create a candidate set C ← P
3: Create a result set R with K pairs ⟨NULL, ∞⟩
4: γ ← the largest travel time in R
5: Run Dijkstra for q on G using ω+(e) and ω−(e) to retrieve
   p.τ_G^+, p.τ_G^−, p.τ_G                    ▷ Phase 1: obtain the threshold γ
6: for each p ∈ C do
7:     Update R, γ by p with p.τ_G^+ or p.τ_G
8: for each p ∈ C do                          ▷ Phase 2: prune objects
9:     if p.τ_G > γ or p.τ_G^− > γ then
10:        Remove p from C
11:    if ∃ route ψ ∈ L such that ψ contains p and q then
12:        Compute p.τ_L                      ▷ optimal subpath property [15] [31]
13:        Update R, γ by p with p.τ_L
14:    Compute p.τ^− as max{p.τ_G^−, p.τ_I^−}
15:    if p.τ^− > γ or (p.τ_L is known and p.τ_L > γ) then
16:        Remove p from C
17: while |C| > K do                          ▷ Phase 3: Issue route requests
18:    Pick an object p ∈ C with minimum p.τ^−        ▷ ordering
19:    Route ψ_{t_now} ← RouteRequest(q, p)           ▷ call external API
20:    Insert ψ_{t_now} into L;  Update ω(e), μ(e) in G for e ∈ ψ_{t_now}
21:    Update p.τ_L for all p on ψ_{t_now}            ▷ optimal subpath property [15]
22:    Run incremental Dijkstra to update all p.τ^−   ▷ By [14]
23:    for each p' ∈ C do
24:        if p'.τ^− > γ or p'.τ_L > γ then
25:            Remove p' from C
26:        if p'.τ_L < γ then
27:            Update R by p' with p'.τ_L
28: Return R
```

Example. Consider the KNN query with $K = 3$ in Fig. 5c. We illustrate the running steps of Route-Saver in Table 2. Entries without values are marked as '/'.

In the first phase, we derive the upper bounds $p.\tau_G^+, p.\tau_G, p.\tau_G^-$ using the time-tagged road graph G, which are shown in the first three columns in Table 2. Since $p_1.\tau_G$, $p_4.\tau_G$ and $p_5.\tau_G^+$ are the smallest three travel times, we insert them into R and update $\gamma = 40$. In the second phase,first we prune candidates p_2, p_5, p_6 since their $p.\tau_G^-$ are larger than γ. Then, we calculate the lower-bound travel time for p_7 using $\mathcal{L}$: $p_7.\tau_I^- = 41 > \gamma$, so p_7 is pruned. We skip the third phase as the candidate set contains exactly $K = 3$ objects, the same as the result set R. Thus, the algorithm returns $R = \{p_1, p_4, p_5\}$ as the query result. Route-Saver issues 0 route request in this example. On the other hand, SMashQ incurs 7 route requests when solving this query (see the method description in Sec. 2.2).

Candidate ordering. For the orderings to rank candidates in C (Line 18) in Algorithm 2, in addition to the orderings discussed in Section 4.3, we propose a new ordering:

Maximum difference (DIFF): Pick a candidate with the maximum $p.\tau^+ - p.\tau^-$. This order tends to tighten the lower and upper bounds of candidates rapidly. A tight $p.\tau^+$ helps refine the value γ whereas a tight $p.\tau^-$ helps prune the candidate itself.

4.5 Applicability of Techniques without Map

In this section, we discuss how to adapt the Route-Saver in case the LBS cannot obtain the same map G used in the route service. We observe that, if the LBS uses the map G' (e.g., a free map [10]) which are not the same with that used in route services, bounding travel times $p.\tau_G^-$ can be over-estimated. For example, if the real shortest path from q to p is missing in local map G', then it is possible that Route-Saver calculates a higher $p.\tau_G^-$ for p and mistakenly prunes it from results. Therefore, the LBS is not allowed to use inaccurate maps.

In case that the LBS cannot access to the map G used in route services, the applicability of our techniques are as follows:

- $p.\tau_G^-$, $p.\tau_G$ and $p.\tau_G^+$ are not applicable because they are calculated based on G, which is not available to the LBS.
- $p.\tau_{\mathcal{L}}$ is applicable, since it is solely calculated using route logs which are obtained from route services.
- $p.\tau_I^-$ is applicable, as it is solely based on route logs.
- $p.\tau_{G_{\mathcal{L}}}^+$ is applicable, where $G_{\mathcal{L}}$ is a road network formed by routes in the log. Observe that $G_{\mathcal{L}}$ must be a subgraph of G.

5 Parallelized Route Requests

Our objective (see Section 3) is to minimize the response time of queries. Section 4 optimizes the response time through reducing the number of route requests. Can we further reduce the response time? In this section, we examine how to parallelize route requests in order to optimize user response time further. We propose two parallelization techniques that achieve different tradeoffs on the number of route requests and user response time.

The execution of algorithms in Section 4 follows a sequential schedule like Fig. 6a. The user response time consists of: (i) the time spent on route requests (in gray), and (ii) local computation at the LBS (in white).

Consider the sequential schedule in Fig. 6a. An experiment (see Fig. 11) reveals that the user response time is dominated by the time spent on route requests. Let a *slot* be the waiting period to obtain a route from the route API[2]. In Fig. 6a, the sequential schedule takes 5 slots for 5 route requests. Intuitively, the LBS may reduce the number of slots by issuing multiple route requests to a route API in parallel. Fig. 6b illustrates a parallel schedule with 2 slots; each slot contains 3 route requests issued in parallel.

Although parallelization helps reduce the response time, it may prevent sharing among routes and cause extra route requests (e.g., request for route p_2), as we will explain later. Existing parallel scheduling techniques [18] have not exploited this unique feature in our problem. We also want to avoid extra route requests because a route API may impose a daily route request limit [8] or charge the LBS based on route requests [5].

2. Different route requests incur similar time (see Fig. 8).

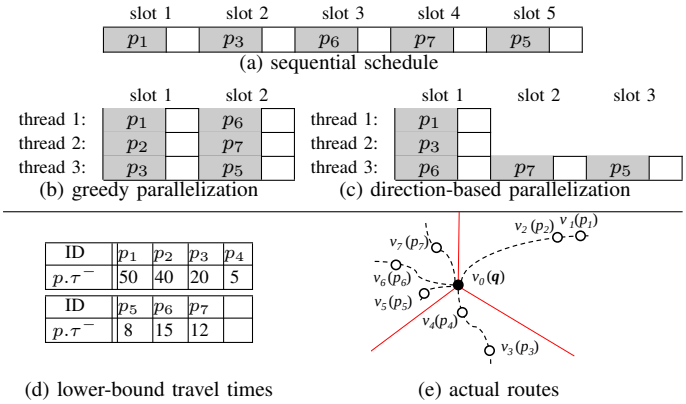


Fig. 6. Effect of parallelization on schedules

We proceed to present two parallelization techniques. They achieve different tradeoffs on the number of route requests and the number of slots. Our discussion focuses on range queries only. Our techniques can be extended to KNN queries as well.

Greedy parallelization. Let m be the number of threads for parallel execution (per query). Our *greedy parallelization* approach dispatches route request to a thread as soon as it becomes available. Specifically, we modify Algorithm 1 as follows. Instead of picking one object p from the candidate set C (at Lines 19–20), we pick m candidate objects and assign their route requests to m threads in parallel. Observe that this approach minimizes the number of time slots in the schedule (Fig. 6b).

We proceed to compare the sequential schedule with the greedy schedule on the example. Consider a range query at q with $T = 60$. Suppose that the candidate set is $C = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. Fig. 6d shows the lower-bound travel time of each object and Fig. 6e depicts the locations of all objects. Assume that the routes (dotted lines) are missing from the the route log $\mathcal{L}$ at the LBS. Here, we order the candidates using DESC ordering (see Section 4.3), and set the number of threads $m = 3$.

Fig. 6a shows a sequential schedule of route requests (issued by the original Algorithm 1). By the DESC ordering, the candidates will be examined in the order: $p_1, p_2, p_3, p_6, p_7, p_5, p_4$. First, a route request is issued for p_1. Since the route to p_1 covers p_2, we save a route request for p_2. Similarly, after issuing a route request for p_3, we save a route request for p_4. After that, route requests are issued for the remaining candidates p_6, p_7, p_5. Note that the sequential schedule (Fig. 6a) takes 5 slots.

Fig. 6b illustrates a parallel schedule of route requests by using the greedy approach. First, it selects m (= 3) objects in the DESC order: p_1, p_2, p_3. Thus, 3 route requests are issued for them at the same time. Since p_4 lies on the route from q to p_3, the route request for p_4 is saved. After that, 3 route requests are issued for remaining candidates p_5, p_6, p_7 at the same time. In summary, the greedy approach takes only 2 slots, but incurs 6 route requests.

Direction-based parallelization. Observe that the extra route request(s) in the greedy approach is caused by objects

at similar directions from q (e.g., p_1, p_2 in Fig. 6e). If we issue route requests to candidates in different directions in parallel, then we may avoid extra route requests. This is the intuition behind our *direction-based parallelization* approach.

In this approach, the LBS divides the candidate set C into m groups $(C_1, C_2, \cdots, C_m)$, based on the direction angle $\measuredangle(q, p)$ of each candidate p from user q. A candidate p is inserted into the group C_i if $\frac{(i-1)\cdot 360^o}{m} \leq \measuredangle(q,p) \leq \frac{i\cdot 360^o}{m}$. This step can be implemented just before Line 18 of Algorithm 1. Then, we modify Lines 19–20 as follows: pick a candidate from each group C_i and then assign their route requests to m threads in parallel.

For example, in Fig. 6e, the candidates are divided into m (= 3) groups based on their direction angles from q: $C_1 = \{p_1, p_2\}$, $C_2 = \{p_3, p_4\}$, and $C_3 = \{p_5, p_6, p_7\}$. Again, the candidates within each C_i are examined by the DESC order. Fig. 6c illustrates the schedule of the direction-based approach. First, this approach selects the candidates $p_1 \in C_1$, $p_3 \in C_2$, and $p_6 \in C_3$, and issues route requests for them in parallel. Since the routes to p_1 and p_3 cover p_2 and p_4 respectively, we saved two requests. After that, C_1 and C_2 become empty. In each subsequent slot, only one route request (for a candidate in C_3) is issued to the route APIs. In total, the direction-based approach incurs only 5 route requests, but it takes 3 slots.

Comparison. In summary, the greedy approach offers the best response time but with considerable extra route requests; the direction-based approach reduces the number of extra route requests and yet provides a competitive response time.

6 Experimental Evaluation

In this section, we compare the accuracy and the performance of our Route-Saver (abbreviated as RS) with an existing method SMashQ (abbreviated as SMQ) [33]. Although SMQ handles only KNN queries, we also adapt it to process range queries. Note that SMQ does not utilize any route log to save route requests. We also consider an extension of SMQ, called SMQ*, which keeps the routes within expiry time into a route log. SMQ* applies only the optimal subpath property [15] [31] and retrieves exact travel times from the log; however, it does not apply the upper/lower bounding techniques in this paper. By default, RS uses the DESC and DIFF orderings for range and KNN queries respectively.

Section 6.1 describes our experimental setting. We first examine the accuracy of the methods on real traffic data in Section 6.2. Then, we study the performance and scalability of the methods in Section 6.3. Finally, in Section 6.4, we conduct small-scale experiments on Google Directions API [7], as it imposes a daily request limit 2,500 per evaluation user [8]. Due to this limit, we use a simulated route API in Sections 6.2, 6.3.

TABLE 3
Experiment Parameters

Parameters	Default	Range
Road map [only for simulation]	Erie	Chowan, Erie, Florida
Dataset size $\|\mathcal{P}\|$	10 (K)	1, 5, 10, 15, 20 (K)
Distribution of query q	uniform	uniform, gaussian
For KNN: Result size K	10	1, 5, 10, 15, 20
For range: Time limit T (*seconds*)	60	10, 30, 60, 90, 120
Expiry time δ (*minutes*)	10	0,2,5,10,20,30
Query rate λ (*queries/minute*)	60	30, 60, 120, 300
Number of threads	1 [sequential]	1, 2, 4, 6, 8, 10

6.1 Experimental Setting

Road networks. For accuracy experiments on real traffic data, we will discuss the road network and traffic data in Section 6.2.

For the performance and scalability study (Section 6.3), we obtain three road maps in USA from [1]: *Chowan* County, in North Carolina (14K nodes, 14K edges), *Erie* County, in Pennsylvania (106K nodes, 115K edges) and *Florida* State (1,049K nodes, 1,331K edges). Following [32], the maximum speed limit $\mathbb{V}_{MAX}$ is set to 110 km/h. According to [2], the travel speed of each road segment is set to a fraction of $\mathbb{V}_{MAX}$, based on its road category.

For the experiments on Google Directions API [7], we consider the *Manhattan* region (in New York), whose area is 87.5km^2.

Performance measure and parameters. For each method, we measure its result accuracy (Sec. 6.2), its number of route requests and user response time (Sec. 6.3). Table 3 summarizes the default values and ranges of parameters used in our experiments. The values for *dataset size* $|\mathcal{P}|$, K, T follow [32]. The default expiry time δ is 10 minutes, according to Fig. 2. To simulate the arrival of queries, we set the default query rate λ to 60 queries / min and uniformly generate query points on the road network. This query rate (60 queries / min) is justified by visit statistics[3] from restaurant and travel guide websites [11].

All methods were implemented in C++ and ran on an Ubuntu 11.10 machine with a 3.4GHz Intel Core i7-3770 processor and 16GB RAM. In experiments, the route log contains at most 30,000 routes and occupies at most 30 MB. The largest road network (Florida) and dataset occupies 87 MB and 1 MB respectively. Thus, the largest map, route log, and dataset can fit in the main memory.

6.2 Accuracy on Real Traffic Data

In this section, we test the result accuracy of the methods on real traffic data, for various expiry time δ (2, 5, 10, 20 and 30 minutes). Other parameters ($|\mathcal{P}|, K, T$) are set to default values in Table 3.

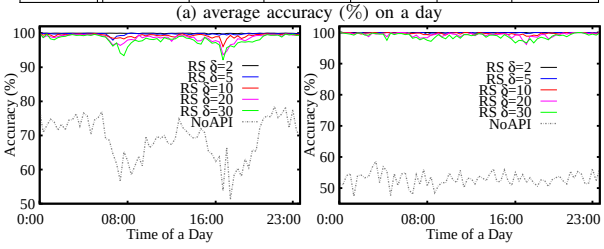
Real traffic data. We downloaded historical real traffic on freeways in Los Angeles from PeMS[4]. The corresponding

3. E.g., *Hotels* has 2.38 million monthly visits, corresponding to the query rate λ = 2,380,000 / (30 · 24 · 60) = 55.1 queries / min. Similarly, *OpenTable* and *UrbanSpoon* have 2.9 and 4 million monthly visits respectively, corresponding to $\lambda = 67.1$ and $\lambda = 92.4$. See the statistics at: http://www.quantcast.com/hotels.com http://www.quantcast.com/opentable.com http://www.quantcast.com/urbanspoon.com

4. California Dept. of Transportation http://pems.dot.ca.gov/

δ (Min)	range			KNN		
	SMQ*	RS	NoAPI	SMQ*	RS	NoAPI
2	99.97	99.95	69.04	99.99	99.99	52.95
5	99.89	99.75		99.95	99.94	
10	99.36	99.28		99.68	99.65	
20	99.02	98.60		99.12	99.10	
30	98.63	98.11		98.95	98.86	

(a) average accuracy (%) on a day



(b) accuracy [range] (c) accuracy [KNN]

Fig. 7. Result accuracy on real traffic data [in color]

road network contains 17,563 nodes and 17,694 edges. We use the traffic data on 31 Dec. 2012 for 24 hours; the travel times on edges are updated every 30 seconds. We also conduct this experiment with traffic data on other dates, and obtain similar results.

Accuracy measure. Besides the methods discussed before, we also consider a baseline method NoAPI, which uses only local distance information to answer queries, without issuing route requests.

We measure the *accuracy* of a method by the $F1$ score:

$$\begin{aligned} F1 &= 2 \cdot precision \cdot recall / (precision + recall) \\ precision &= \mid R_{method} \cap R^* \mid / \mid R_{method} \mid \\ recall &= \mid R_{method} \cap R^* \mid / \mid R^* \mid \end{aligned}$$

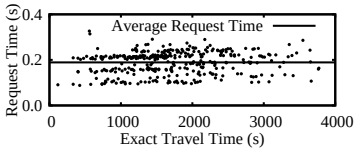
where R^* is the exact result set derived from the current traffic and R_{method} is the result set obtained by a method.

The accuracy of SMQ is always 100% because it does not use route log. We only measure the accuracy of RS, SMQ*, NoAPI in this experiment. Fig. 7a shows the average accuracy of the methods on a day. NoAPI has low accuracy as it does not use live traffic information. Our proposed RS and SMQ* can find results with very high accuracy. When the expiry time δ increases, the route log contains less accurate travel time information and thus the accuracy decreases. The standard deviation of the accuracy is within 1.5% for SMQ* and RS, whereas NoAPI has a higher standard deviation. Fig. 7b, 7c show the accuracy of RS and NoAPI along the timeline. As a remark, the traffic changes most rapidly during rush hours in the morning and the evening. During those intervals, the accuracy of the methods on range queries drops because their result sizes are sensitive to the traffic. The accuracy on KNN queries is insensitive to the traffic due to the fixed result size.

As we will show in Section 6.3, RS issues much fewer route requests than SMQ*. RS still achieves high accuracy because our proposed bounding techniques offers tight lower/upper bounds. We found in our experiments that, the upper bounds, if exist, are almost equal to the exact travel time in most cases, and the lower bounds are at least 60% of the exact travel time.

6.3 Performance and Scalability Study

For the sake of obtaining the user response time in our simulations, we measure the time of route requests on Google Directions API [7]. On each roadmap, we randomly sample 400 pairs of points and issue route requests for them to Google Directions API. Fig. 8a plots the time of each route request versus its length (exact travel time), on the Erie roadmap. Fig. 8b summarizes the average and standard deviation of route request time on all roadmaps.



(a) Erie

Roadmap	AVG. (s)	STDEV. (s)
Chowan	0.173	0.060
Erie	0.189	0.0287
Florida	0.165	0.0490
New York	0.113	0.003

(b) statistics on all roadmaps

Fig. 8. Time of route requests on roadmaps

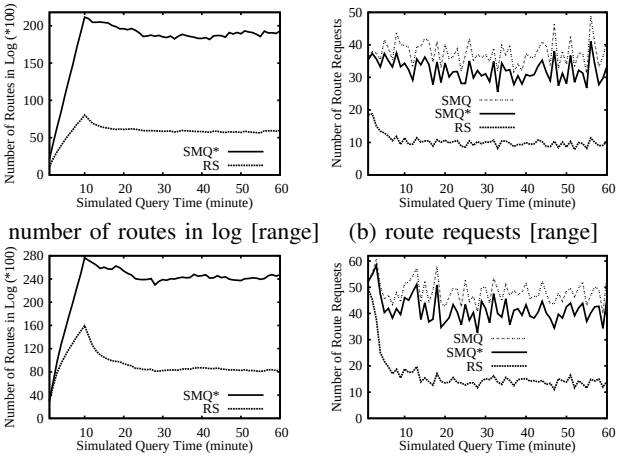
Section 6.3.1 studies the temporal stability of the methods along the timeline. Section 6.3.2 examines the effect of our proposed optimizations. Section 6.3.3 tests the scalability of the methods with respect to various parameters. Section 6.3.4 evaluates the performance of RS with parallelization.

6.3.1 Temporal Stability

In this section, we simulate the arrival of queries along a 60-minute (1-hour) timeline, while fixing all parameters to default. Thus, each test uses $60 \cdot \lambda = 3600$ queries. The route log $\mathcal{L}$ is initially empty. To report temporal behavior, we measure (i) the route log size and (ii) the number of route requests of each query.

We first conduct experiments with uniformly distributed queries and datasets. Fig. 9a shows the number of routes in $\mathcal{L}$ of RS and SMQ* versus the timeline, for range queries. SMQ is not plotted here as it does not utilize the log $\mathcal{L}$. The log size rises steadily in the first $\delta = 10$ minutes (the warm-up period) and then the expiration mechanism starts its effect. Observe that the drop in the log size during the $[10, 20)$ minutes matches with the drop in the number of route requests during the $[0, 10)$ minutes (see Fig. 9b). After that, the log size remains stable in subsequent minutes because $\mathcal{L}$ contains only the routes requested by the latest $\lambda \cdot \delta$ queries. SMQ* has a larger log size because it incurs more route requests than RS.

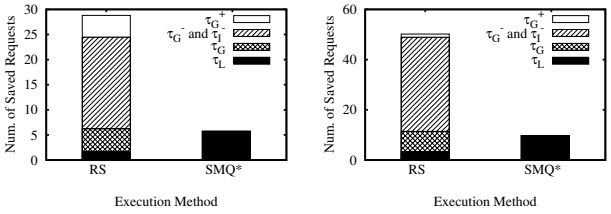
Fig. 9b illustrates the number of route requests of each query versus the timeline, for range queries. The performance of SMQ remains constant since it does not utilize the route log. In the first $\delta = 10$ minutes, as the log size of RS rises, it could exploit more information, like deriving exact values and tight lower/upper bounds for travel times, to reduce the number of route requests. After that, its log size keeps stable so its performance also keeps stable. The trend of SMQ* is similar to RS, except that SMQ* incurs much more route requests than RS. That is because SMQ* uses only the optimal subpath property to derive exact travel times from the route log, but it does not use the lower/upper bounds applied in RS. Experimental results on KNN queries are similar (see Fig. 9c,d).



(a) number of routes in log [range] (b) route requests [range]

(c) number of routes in log [KNN] (d) route requests [KNN]

Fig. 9. Temporal behavior, expiry time $\delta = 10$



(a) saved route request [range] (b) saved route request [KNN]

Fig. 10. Effectiveness of techniques

TABLE 4
Effect of candidate ordering

	RS [range]		RS [KNN]		
	ASC	DESC	ASC	DESC	DIFF
route requests	17.12	11.57	21.29	20.29	18.39

As observed in the above experiments, the number of route requests converges to a stable value after the first δ minutes (the warm-up period). Thus, in subsequent experiments, we simulate the arrival of queries along 2δ-minute time interval. We view the first δ minutes as the *warm-up period*, and the last δ minutes as the *stable period*. We only measure the average performance in the stable period.

6.3.2 Effect of Optimization Techniques

First, we investigate the effectiveness of our proposed lower/upper bound techniques. Recall that RS exploits the travel time information obtained from recent routes for three techniques: (i) retrieve the exact travel time of a point p, (ii) prune p by its lower bound $p.\tau_G^-, p.\tau_I^-$ (excluding cases using $p.\tau_c^-$), and (iii) detect p as a true hit by its upper bound $p.\tau_G^+$. We further divide technique (i) into two types: (i.a) existing technique using the optimal subpath property [15] on the route log $\mathcal{L}$, and (i.b) our proposed technique using Lemma 2 on the time-tagged network G. Note that SMQ* applies only technique (i.a), but not techniques (i.b), (ii), (iii).

Fig. 10 depicts the statistics of applying these techniques in the methods, at the default setting. Observe that our proposed lower-bound technique (for computing $p.\tau_G^-, p.\tau_I^-$) saves the largest number of route requests, while the existing technique for computing exact travel time $p.\tau_{\mathcal{L}}$ (using optimal subpath property) saves the least. The reason for $p.\tau_G^-, p.\tau_I^-$ outperforming $p.\tau_G^+$ is that, RS has a higher chance to derive a tight $p.\tau_G^-, p.\tau_I^-$ for each data point, but a finite $p.\tau_G^+$ may not exist for a data point.

Next, we study the effect of candidate orderings on RS in terms of the number of route requests per query. It can apply the ASC / DESC orderings for range queries, and ASC / DESC / DIFF orderings for KNN queries. Table 4 shows that RS-DESC and RS-DIFF achieve the best performance for range and KNN queries, respectively.

6.3.3 Scalability Experiments

As discussed before, in this section, we simulate the arrival of queries along 2δ-minute time interval. And we measure the performance in terms of: (i) average number of route requests per query in the stable period, and (ii) average user response time per query in the stable period. Furthermore, we also plot the breakdown of user response time into server CPU time and the time spent on route requests, as illustrated in Section 5. The server CPU time already includes the overhead of maintaining the structures in Section 4.1.

Effect of expiry time δ. Fig. 11a shows the average number of route requests for range queries with respect to various δ. To illustrate the trend of route requests for smaller expiry times, we add the result for four more δ (20, 30, 60, 90 seconds) apart from the values listed in Table 3. Since SMQ does not use the log, its cost remains constant and much higher than that of RS and SMQ*. When δ increases, the route log of RS and SMQ* accumulates routes requested from more warm-up queries ($\lambda \cdot \delta$). Thus, RS and SMQ* could exploit more information in the log to reduce the cost. Fig. 11c illustrates the decomposition of the user response time for various δ. Here, 'R', 'S', 'S*' refer to RS, SMQ, SMQ*, respectively. To make the server CPU time visible, we plot the y-axis in log scale. Clearly, the time on route requests dominates the user response time. RS achieves a low server CPU time (0.1s) and user response time (1s). As a remark, the LBS's query throughput is decided only by its CPU time because it remains idle while issuing route requests. Fig. 11b,d depict the performance for KNN queries. The trends are similar to those for range queries. Due to the overhead on using route log, RS and SMQ* incur slightly higher server CPU time than SMQ.

Since the user response time is mostly spent on route requests, we only report the number of route requests in experiments below.

Effect of query rate λ. As shown in Fig. 12a,b, the effect of query rate λ on the performance is similar to that of expiry time δ as discussed above. The reason is that, the route log of RS and SMQ* accumulate routes requested from more warm-up queries ($\lambda \cdot \delta$), as λ increases.

Effect of dataset size $|\mathcal{P}|$. In this experiment, we vary dataset size $|\mathcal{P}|$ and plot the number of route requests for range queries in Fig. 12c. The number of route requests rises proportionally to $|\mathcal{P}|$ as more objects are

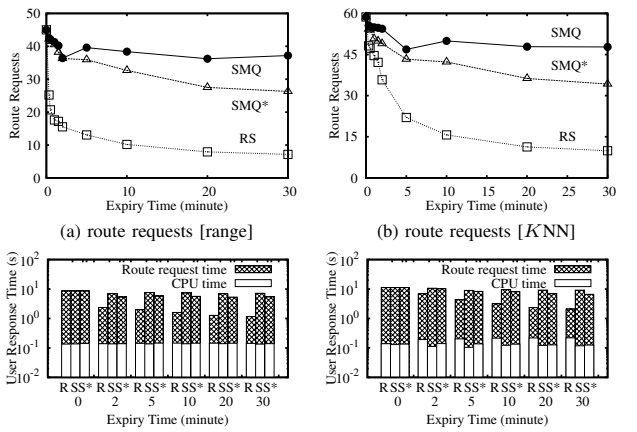


Fig. 11. Effect of expiry-time δ

TABLE 5
Effect of roadmap on range and KNN queries

Roadmap	route requests [range]			route requests [KNN]		
	SMQ	SMQ*	RS	SMQ	SMQ*	RS
Chowan	37.81	18.76	2.37	44.03	20.38	3.29
Erie	40.53	36.1	11.92	49.23	41.36	16.8
Florida	44.7	40.07	18.71	55.31	54.51	25.01

covered by the query range. The performance gap between SMQ/SMQ* and RS widens because RS applies effective bounding techniques. In contrast, for KNN queries, the performance is insensitive to $|\mathcal{P}|$, as depicted in Fig. 12d. When $|\mathcal{P}|$ increases, the travel time from q to its KNN decreases. This enables pruning more candidates, canceling out the effect of $|\mathcal{P}|$.

Effect of time limit T and result size K. Fig. 12e shows the performance of the methods on range queries versus the travel time limit T. As T increases, the number of query results increases and so does the number of route requests. RS outperforms SMQ/SMQ* by a wide margin. As a remark, the average number of query results rises from 1.5 to 35.6 when T increases from 10s to 120s. Fig. 12f depicts the performance of the methods by varying K. Again, when the result size K increases, so does the number of route requests.

Effect of roadmaps. We then examine the effect of the roadmaps on the performance of the methods. Table 5 lists the roadmaps in ascending sizes (Chowan, Erie, Florida), together with the average number of route requests of the methods, for range queries and KNN queries. We follow the experimental methodology in [34] and fix the *object density* (i.e., the ratio of $|\mathcal{P}|$ over the number of nodes in the network) to 10%. That is, we have $|P| =$ 1.4K, 10.6K and 104.9K for Chowan, Erie and Florida, respectively. As the log routes have fewer intersections in larger road networks, the derived lower/upper bounds become looser, and thus the number of route requests increases in larger networks.

Effect of query distribution. This experiment illustrates the effect of query distribution on the performance of the methods. For each query set 'Gau_x%', we select 30 Gaussian bells randomly, set the standard deviation of each Gaussian to be x% of the map domain length [13], and

TABLE 6
Effect of query distribution

Distribution of query q	route requests [range]			route requests [KNN]		
	SMQ	SMQ*	RS	SMQ	SMQ*	RS
Gau_2.5%	42.43	17.44	3.83	48.61	15.33	5.51
Gau_5.0%	40.84	23.52	5.91	51.02	25.52	8.25
Gau_10%	41.42	28.53	8.51	51.24	28.43	11.52
Gau_20%	41.73	32.00	10.31	50.81	33.33	14.74
Gau_50%	40.92	33.21	10.98	48.85	36.53	15.65
Uniform	39.37	34.67	11.57	49.97	42.29	18.39

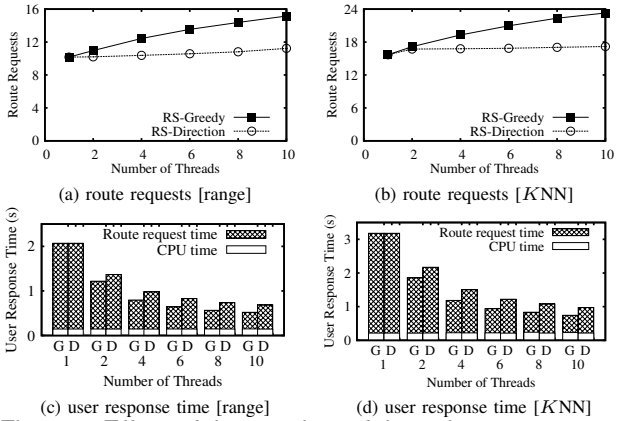


Fig. 13. Effect of the number of threads m

generate points in these bells following such distribution. For comparison, we also use an uniformly generated query set ('Uniform').

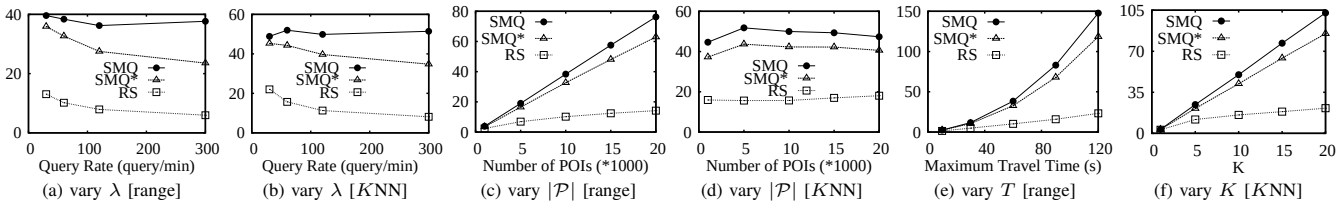
Table 6 shows that, SMQ is insensitive to the distribution of the queries since it does not utilize the logs obtained from recent queries. For Gaussian queries, when the standard deviation is small, the current query is likely to be near to some recent queries, and thus recent routes provide valuable information for RS and SMQ* to save route requests. Observe that uniform query distribution, i.e., our default query distribution, leads to the the worst-case performance because the current query can be located far from recent queries and reuse less information from their routes.

6.3.4 *Effect of Route Request Parallelization*

This section studies the user response time of parallelization variants of RS: (i) RS-Greedy using greedy parallelization, and (ii) RS-Direction using direction-based parallelization. Fig. 13a,c show their average number of route requests and user response time versus the number of threads m, for range queries. At $m = 1$, both variants are the same as RS which issues route requests sequentially and incurs the longest user response time. As expected in Section 5, RS-Direction results in fewer route requests but a slightly longer user response time than RS-Greedy. We obtain similar experimental results for KNN queries in Fig. 13b,d.

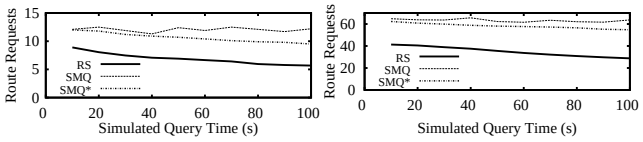
6.4 Experiments on Google Directions API

We have implemented SMQ, SMQ* and RS with Google Directions API [7], whose request/response format has been described in Section 2.2. Due to the daily request limit (2,500) for evaluation users [8], we conduct this experiment on the Manhattan region (see Section 6.1). We randomly



(a) vary λ [range] (b) vary λ [KNN] (c) vary $|\mathcal{P}|$ [range] (d) vary $|\mathcal{P}|$ [KNN] (e) vary T [range] (f) vary K [KNN]

Fig. 12. Effect of various parameters [y-axis: route requests]



(a) route requests [range] (b) route requests [KNN]

Fig. 14. Temporal behavior vs. timeline, $\delta = 10$ minutes, Manhattan region (in New York), on Google Directions API

select 100 POIs[5] in this region, and generate 100 queries (along a 100-second time period).

Fig. 14 depicts the number of route requests of each query versus the timeline, for range queries and KNN queries. RS outperforms SMQ and SMQ* on both range queries and KNN queries. Also, the performance gap between them widens with the timeline. The number of route requests is still decreasing as the timeline has not yet reached the (default) expiry time $\delta = 10$ minutes.

7 Conclusion

In this paper, we propose a solution for the LBS to process range/KNN queries such that the query results have accurate travel times and the LBS incurs few number of route requests. Our solution Route-Saver collects recent routes obtained from an online route API (within δ minutes). During query processing, it exploits those routes to derive effective lower-upper bounds for saving route requests, and examines the candidates for queries in an effective order. We have also studied the parallelization of route requests to further reduce query response time. Our experimental evaluation shows that Route-Saver is 3 times more efficient than a competitor, and yet achieves high result accuracy (above 98%).

In future, we plan to investigate automatic tuning the expiry time δ based on a given accuracy requirement. This would help the LBS guarantee its accuracy and improve their users' satisfaction.

Acknowledgments

This work was supported by ICRG grant G-YN38 from Hong Kong Polytechnic University.

References

[1] 2011 Census TIGER/Line Shapefiles. http://www.census.gov/cgi-bin/geo/shapefiles2011/main.

[2] 9th DIMACS Implementation Challenge on Shortest Paths. http://www.dis.uniroma1.it/challenge9/data/tiger/.

[3] Bing Data Suppliers. http://windows.microsoft.com/en-HK/windows-live/about-bing-data-suppliers/.

[4] Bing Maps API. http://www.microsoft.com/maps/developers/web.aspx.

[5] Bing Maps Licensing and Pricing Information. http://www.microsoft.com/maps/product/licensing.aspx.

[6] Google Directions & Bing Maps: Live Traffic Information. http://support.google.com/maps/bin/answer.py?hl=en&answer=2549020&topic=1687356&ctx=topic http://msdn.microsoft.com/en-us/library/aa907680.aspx.

[7] Google Directions API. https://developers.google.com/maps/documentation/directions/.

[8] Google Directions API Usage Limits. https://developers.google.com/maps/faq#usagelimits.

[9] Google Map Maker Data Download. https://services.google.com/fb/forms/mapmakerdatadownload/.

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

[11] Statistics of Usage. http://www.quantcast.com.

[12] US Maps from Government. http://www.usgs.gov/pubprod/.

[13] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: a multidimensional workload-aware histogram. In *SIGMOD*, 2001.

[14] E. P. F. Chan and Y. Yang. Shortest path tree computation in dynamic graphs. *IEEE Trans. Computers*, 58(4):541–557, 2009.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, USA, 2009.

[16] U. Demiryurek, F. B. Kashani, C. Shahabi, and A. Ranganathan. Online computation of fastest path in time-dependent spatial networks. In *SSTD*, pages 92–111, 2011.

[17] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, 1996.

[18] M. Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[19] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *VLDB*, 2006.

[20] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE TKDE*, 14(5):1029–1046, 2002.

[21] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, page 10, 2006.

[22] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, 2004.

[23] H.-P. Kriegel, P. Kröger, P. Kunath, and M. Renz. Generalizing the optimality of multi-step *k* -nearest neighbor query processing. In *SSTD*, pages 75–92, 2007.

[24] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt . Hierarchical graph embedding for efficient query processing in very large traffic networks. *SSDBM*, pages 150–167, 2008.

[25] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Proximity queries in large traffic networks. In *ACM GIS*, 2007.

[26] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.

[27] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *ICDE*, 2012.

[28] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, 2008.

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

[30] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, pages 154–165, 1998.

[31] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *SIGMOD*, 2012.

5. E.g., There are about 50 ATMs in the Manhattan region http://locators.bankofamerica.com/locator/locator/ListLoadAction.do

[32] D. Zhang, C.-Y. Chow, Q. Li, X. Zhang, and Y. Xu. Efficient evaluation of k-nn queries using spatial mashups. In *SSTD*, 2011.
[33] D. Zhang, C.-Y. Chow, Q. Li, X. Zhang, and Y. Xu. Smashq: spatial mashup framework for k-nn queries in time-dependent road networks. In *Distributed and Parallel Databases*, 2012.
[34] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *CIKM*, 2013.

Yu Li received the bachelor's degree in 2010 from Northwestern Polytechnical University, China. She is currently a PhD student in Hong Kong Polytechnic University, under the supervision of Dr. Man Lung Yiu.

Man Lung Yiu received the bachelor's degree in computer engineering and the PhD degree in computer science from the University of Hong Kong in 2002 and 2006, respectively. Prior to his current post, he worked at Aalborg University for three years starting in the Fall of 2006. He is now an assistant professor in the Department of Computing, Hong Kong Polytechnic University. His research focuses on the management of complex data, in particular query processing topics on spatiotemporal data and multidimensional data.