Efficient Top-k Matching for Publish/Subscribe Ride Hitching

Yafei Li, Hongyan Gu, Rui Chen, Jianliang Xu, Shangwei Guo, Junxiao Xue, Mingliang Xu

Abstract—With the continued proliferation of mobile Internet and geo-locating technologies, carpooling as a green transport mode is widely accepted and becoming tremendously popular worldwide. In this paper, we focus on a popular carpooling service called *ride hitching*, which is typically implemented using a publish/subscribe approach. In a ride hitching service, drivers subscribe ride orders published by riders and continuously receive matching ride orders until one is picked. The current systems (e.g., Didi Hitch) adopt a threshold-based approach to filter ride orders. That is, a new ride order will be sent to all subscribing drivers whose planned trips can match the ride order within a pre-defined detour threshold. A limitation of this approach is that it is difficult for drivers to specify a reasonable detour threshold in practice. In addressing this problem, we propose a novel type of top-k subscription queries called *Top-k Ride Subscription (TkRS)* query, which continuously returns the best k ride orders that match drivers' trip plans to them. We propose two efficient algorithms to enable the top-k result maintenance. We also design a novel hybrid grid index and a two-level buffer structure to efficiently track the top-k results for all *TkRS* queries. Finally, extensive experiments on real-life datasets suggest that our proposed algorithms are capable of achieving desirable performance in practical settings.

Index Terms—Location-based service, ridesharing, order dispatch, query processing, optimization.

1 Introduction

As a green transport mode, carpooling has been increasingly prevalent around the globe. With carpooling, empty seats in taxis and private vehicles can be utilized to reduce traffic congestion, travel costs, and carbon emissions. As shown in a recent study [10], the potential urban traffic reduction can be as high as 59% if people are willing to share a ride with others whose trip plans are similar. In general, there are two kinds of carpooling services: 1) *ride sharing* (e.g., UberPool [28], Lyft Shared [21], GrabShare [14]), in which passengers going in the same direction are grouped together for shared rides by the professional drivers whose destinations are determined by the riders [9], [22], [23]; 2) *ride hitching* (e.g., DidiHitch [11], GrabHitch [14]), in which drivers have their trip plans and share their unoccupied seats with passengers heading towards similar destinations [17], [19], [27]. In this paper, we focus on ride hitching.

Ride hitching is typically implemented as a publish/subscribe service. Drivers subscribe ride orders published by riders and continuously receive ride orders matching their trip plans until one is picked. The current systems (e.g., Didi Hitch) adopt a threshold-based approach for filtering ride orders. That is, a new ride order will be sent to all subscribing drivers whose planned trips can match the requested ride within a predefined detour threshold. A limitation of this approach is that it is difficult for drivers to specify a reasonable detour threshold in practice. If the threshold is set too large, drivers may be overwhelmed by a large number of ride orders with some of them being far way. On the other hand, setting a small threshold may end up with no matching ride orders.

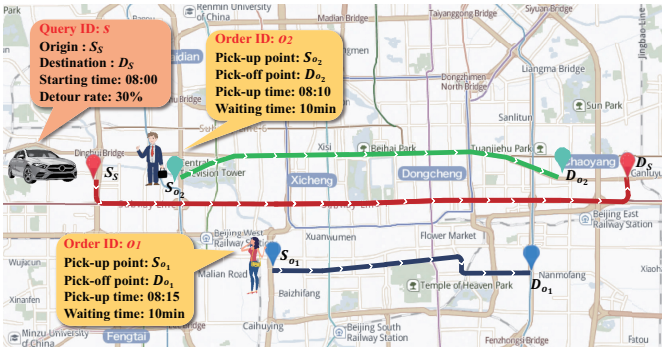


Fig. 1: A toy example of a *TkRS* query

In addressing this problem, in this paper we propose a novel type of top-k subscription queries, referred to as *Top-k Ride Subscription (TkRS)* queries, which continuously returns the best k ride orders that match drivers' trip plans to them. To evaluate *TkRS* queries, we consider the following two aspects: i) *matching*, meaning that a driver should pick up a rider within the rider's expected pick-up time window and that the detour incurred by picking up and dropping off the rider should be less than the driver's tolerable detour; ii) *ranking*, meaning that the higher the trip similarity between the driver and the rider, the higher the ranking.

Example 1. *Consider the example in Fig. 1, where s is a TkRS query (from source S_s to destination D_s) subscribed by a driver, and o_1 (from S_{o_1} to D_{o_1}) and o_2 (from S_{o_2} to D_{o_2}) are two ride*

- *Y. Li, H. Gu, J. Xue, and M. Xu are with the School of Information Engineering, Zhengzhou University, Zhengzhou, China. E-mail: {ieyfli, xuejx, iexumingliang}@zzu.edu.cn, guhy@gs.zzu.edu.cn*
- *R. Chen is with the College of Computer Science and Technology, Harbin Engineering University, China. E-mail: ruichen@hrbeu.edu.cn (Corresponding author: Rui Chen)*
- *J. Xu is with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong SAR, China. E-mail:xujl@comp.hkbu.edu.hk*
- *S. Guo is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail:shangwei.guo@ntu.edu.sg*

Manuscript received April 19, 2005; revised August 26, 2015.

orders that arrive in sequence. The server continuously maintains the top-1 result for s. When o_1 arrives, the server returns o_1 as the top-1 result of s. Later on, when o_2 arrives, the server compares o_2 with o_1 and determines which one matches s better. Obviously, o_2 wins over o_1 since o_2 is much closer to s than o_1. Thus, the top-1 result of s is updated from o_1 to o_2.

While effectively facilitating drivers to select ideal ride orders in practice, supporting *TkRS* queries in ride hitching services brings new technical challenges. First, when a new ride order arrives, we need to determine the extent of its trip matching and calculate a ranking score for each matched *TkRS* query. Obviously, this process is computationally prohibitive when the number of *TkRS* queries is enormous or the ride orders arrive at a very high rate. Second, when a ride order in the top-k results expires, it may also incur a huge computational cost for each affected *TkRS* query to find the best substitute from a list of active ride orders. Third, tackling the two aforementioned processes involves a significant of routing/re-routing computations, making it very time consuming. As such, designing efficient algorithms to handle the *TkRS* query problem requires non-trivial efforts.

In this paper, we aim at maintaining up-to-date top-k results for a large number of *TkRS* queries over a mass of ride orders arriving in a stream. We propose several efficient algorithms to solve this problem. More specifically, we propose a set of pruning techniques to efficiently reduce the solution space, which can speed up the searching process. To enhance the pruning capability, we also devise an effective hybrid index that encodes high-level clustering information, which can efficiently filter out the unaffected *TkRS* queries in a group manner. Moreover, inspired by the concepts of *k-skyband* and *reachable area*, we design a novel two-level buffer to further strengthen the capability of top-k result maintenance upon expiry of ride orders in the top-k results.

Overall, the main contributions of this paper are summarized as follows:

- We formally define a new *TkRS* query problem where the server continuously maintains the top-k ride orders for each *TkRS* query over a ride-order stream. To the best of our knowledge, we are the first to study this problem.
- We propose an efficient *TkRSMonitor* algorithm to handle the *TkRS* query problem, which is equipped with several efficient pruning techniques to substantially accelerate the maintenance process.
- We design a novel index structure called *hybrid grid (HG)* index to encode routing query sets, reachable cell sets, and bounds of travel constraints, endowing the *TkRSMonitor* algorithm with effective group pruning capability.
- We also devise an effective two-level buffer structure to optimize the top-k result maintenance for each *TkRS* query upon expiry of ride orders in the top-k results.
- We evaluate the performance of our proposed algorithms on two real-world datasets. The experimental results demonstrate that our proposed algorithms achieve desirable performance under a wide range of practical settings.

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 formulates the definitions used in this paper. Section 4 introduces several efficient algorithms and two kinds of index mechanisms. Section 5 evaluates our proposed algorithms. Finally, Section 6 concludes this paper and highlights the future work.

2 Related Work

To the best of our knowledge, there are no existing works studying the problem of top-k publish/subscribe for ride hitching. As such, we review the existing studies in relevant areas including publish/subscribe and ridesharing services.

2.1 Publish/Subscribe Services

A publish/subscribe service uses a message distribution approach where users register their interests as the long-term queries in the system, and messages from the senders are classified into different groups and delivered to relevant users whose interests are matched.

A line of existing works [1], [3], [16], [30], [32] focuses on developing publish/subscribe systems for spatial message distribution. Chen et al. [3], [30] consider spatial objects with a spatial point and match them with a set of subscriptions if they fall into the associated regions of the matched subscriptions, while [16], [32] consider spatial-temporal objects with a spatial region and send them to a set of subscriptions when the spatial-temporal objects overlap their associated spatial regions. To further enhance the usability of publish/subscribe services, several other existing works [2], [4], [5], [26], [29] study top-k message publish/subscribe where a server delivers timely messages with top-k matching scores to relevant subscribers. Chen et al. [2], [4], [5] study how to maintain the top-k messages in a time period that is defined by a sliding time window. Both top-k techniques and spatial-temporal subscription are used to monitor the top-k spatial-temporal terms over a real-time document stream. The above works are inherently different from ours, and it is not trivial to adapt these techniques to support publish/subscribe ridesharing services.

2.2 Ridesharing Services

As a new model of transportation, ridesharing has largely reshaped the transportation market and also attracted much more attention from both academic and industry communities [8], [11]–[13], [21], [23], [27], [33], [34]. Two primary categories of ridesharing services are *join-based ridesharing* and *search-based ridesharing*. Join-based ridesharing [9], [13], [27], [33] intends to solve the problem of how to find driver-rider pairs with the global optimal rewards between a set of drivers and a set of riders. Search-based ridesharing [6], [8], [12], [15], [17], [19], [23], [31] focuses on the situation where rides' ride requests are served in a stream fashion. For each arriving ride request, the server immediately returns the most suitable driver from the available drivers in the road network.

Different from previous ridesharing works, the problem defined in this paper is the first work to study how to efficiently maintain the up-to-date top-k results for a mass of *TkRS* queries over a ride-order stream.

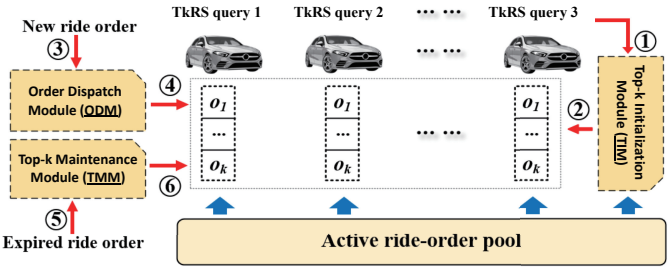


Fig. 2: The system framework of the *TkRS* query service

3 Problem Formulation

In this section, we first brief the framework of the *TkRS* query service, then present several definitions used in this paper, followed by a running example to illustrate the entire process, and finally give the problem statement. Table 1 summarizes the main notations used throughout this paper.

3.1 Framework

The general framework of the *TkRS* query service is shown in Fig. 2. It includes three main modules: a top-k initialization module (TIM), an order dispatch module (ODM), and a top-k maintenance module (TMM). The main workflow is as follows. Drivers subscribe their *TkRS* queries in the server. The TIM sends back the initial top-k results of their *TkRS* queries to the drivers (① & ②). When a new ride order arrives from a rider, the ODM is invoked to find all affected *TkRS* queries and update their top-k results (③ & ④). Once a ride order expires, the TMM automatically removes it from the top-k results of all affected *TkRS* queries and simultaneously replenishes with the most suitable ride order from the active ride-order pool (⑤ & ⑥). Note that a *TkRS* query remains active until the driver selects a ride order from the top-k results or its departure time arrives; a ride order keeps active until it is selected by a driver or it is expired. Active ride orders are stored in the active ride-order pool until they expire or are selected. Based on the setting of real-world ride hitching services, we assume each ride order will be expired a certain time before its pick-up time, e.g., 30 minutes. Meanwhile, the travel route in a typical ride hitching is that the driver first departs from his/her origin, then picks up and drops off the riders at some locations, and finally drives to his/her destination.

In our service, the system is responsible for continuously notifying the up-to-date top-k ride orders for each driver, until the driver decides which ride order to serve from the top-k results. Only after a driver confirms to serve a rider will the rider be notified of the serving driver, and the service relationship will not change. Moreover, if a ride order is selected by a driver, the ride order will be removed from the system and other drivers' top-k results. In addition, the ride hitching services, such as Didi Hitch and Grab Hitch, usually do not allow drivers to select riders on their way, because frequent updating the riders may distract the drivers, which is a dangerous behavior that may cause accidents. Based on the real-world business setting, in this paper we assume that the matching is done before drivers leave their origins.

TABLE 1: Summary of Notations

Notation	Definition
$G = (L, E)$	a road network with a point set L and a road set E.
$\mathcal{O}$, $\mathcal{S}$	a stream of ride orders, a set of *TkRS* queries.
o, s	a ride order, a *TkRS* query.
γ_s	the tolerable detour ratio of s.
$\kappa(o, s)$	the ranking score of o and s .
$\pi(\cdot, \cdot)$	the shortest path distance of two points.
$\delta(\cdot, \cdot)$	the Euclidean distance of two points.
$\mathcal{H}_s^k$, o_s^k	the top-k result of s, the k-th ride order in $\mathcal{H}_s^k$.
$\delta(\cdot, \cdot)$	the Euclidean distance of two points.
c	a grid cell.
Π_c	a set of *TkRS* queries crossing c.
$\mathcal{R}_c$	the reachable cells of the *TkRS* queries in Π_c.
$\mathcal{C}_s$	the representation of a *TkRS* query s.
t_s^c	the time range of s moving over c.
τ_c	the upper bound of maximum trip distance of s in Π_c.
t_c^-, t_c^+	the earliest time arriving at c, the latest time departing from c.
$\mathcal{O}_s^o$	the candidates of the expired ride order o in $\mathcal{H}_s^k$.
$\mathcal{B}_s^k$, $\widehat{\mathcal{B}_s^k}$	the k-skyband of $\mathcal{O}$ with regard to s, the partial k-skyband of $\mathcal{O}$ with regard to s.
κ_s^b	the matching ratio of o_s^k when invoking the top-k computation most recently.

3.2 Problem Definition

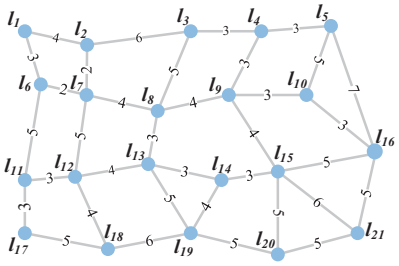
A *TkRS* query is defined over a road network $\mathcal{G} = (L, E)$ and a stream of ride orders $\mathcal{O}$. The road network $\mathcal{G}$ consists of a set of points L denoting road intersections and a set of edges $E \subseteq L \times L$ denoting road segments connecting a pair of road intersections. Each ride order $o \in \mathcal{O}$ is denoted by a tuple $o = (t_o^p, t_o^w, l_o^p, l_o^d)$ where t_o^p denotes the pick-up time, t_o^w the tolerable waiting time, l_o^p the pick-up point, and l_o^d the drop-off point. A formal definition of a *TkRS* query is given in Definition 1.

Definition 1. *(TkRS query) A TkRS query, denoted by a tuple $s = (t_s^p, l_s^p, l_s^d, \gamma_s, k)$ where t_s^p denotes the departure time, l_s^p the origin, l_s^d the destination, γ_s the tolerable detour ratio, and k the number of ride orders to be maintained, aims to continuously maintain the up-to-date top-k results over a stream of ride orders.*

The parameter γ_s in Definition 1 naturally derives from the observation that drivers usually expect their detour ratios not to exceed their tolerable thresholds in order to guarantee high sharing utility [17], [19], [27]. In practice, γ_s can be expressed as a set of options for users to select from. The parameter k means that the system recommends the top-k matched ride orders for one unoccupied seat of a driver.

For ease of description, in the sequel we use the terms rider and ride order or driver and *TkRS* query interchangeably. In addition, maintaining top-k riders will provide a driver more flexibility to select the best rider based on his/her personalized preference, e.g., the rider's rating and gender, which may affect ridesharing experience. Next, we formally define the matching between a ride order and a *TkRS* query below.

Definition 2. *(Matching) Given a set of ride orders $\mathcal{O}$ and a set of TkRS queries $\mathcal{S}$, we call $(o, s) \in \mathcal{O} \times \mathcal{S}$ a matching if the time constraints of pick-up time and tolerable waiting time of o and the distance constraint of tolerable detour ratio of s can be satisfied when s serves o.* ■



(a) Road network

S_1	$\langle l_1, l_5\rangle$
S_2	$\langle l_1, l_{15}\rangle$
S_3	$\langle l_{11}, l_{10}\rangle$

(b) *TkRS* queries

O_1	$\langle l_1, l_9\rangle$
O_2	$\langle l_1, l_{12}\rangle$
O_3	$\langle l_7, l_{10}\rangle$
O_4	$\langle l_{12}, l_{15}\rangle$
O_5	$\langle l_{11}, l_{16}\rangle$
O_6	$\langle l_2, l_{10}\rangle$

(c) Ride orders

	S_1	S_2	S_3
Top-1	O_1 (0.84)	O_1 (1)	O_4 (0.85)
Top-2	O_3 (0.76)	O_2 (0.94)	O_5 (0.81)

	S_1	S_2	S_3
Top-1	O_6 (0.8)	O_2 (0.94)	O_4 (0.85)
Top-2	O_3 (0.76)	O_4 (0.85)	O_5 (0.81)

(d) The maintenance of top-2 results of s_1, s_2 and s_3 when o_1 expires and o_6 arrives

Fig. 3: A running example of *TkRS* queries

To measure the trip closeness of a ride order o matching a *TkRS* query s, we adopt the following ranking function

$$\kappa(o, s) = \frac{\pi(l_s^p, l_s^d)}{\pi(l_s^p, l_o^p) + \pi(l_o^p, l_o^d) + \pi(l_o^d, l_s^d)}, \quad (1)$$

where $\pi(\cdot, \cdot)$ denotes the travel cost between two points in the road network and can be measured as a travel time or a travel distance [20], [23], [31]. The ranking function in Equation 1 essentially measures the trip closeness between a driver and a rider, and has been widely adopted in many existing works [12], [19], [23], [27]. Without loss of generality, any metric that measures the similarity of two trajectories can also be used to measure the trip closeness. Note that in this paper we follow the conventional assumption that a driver can serve at most one rider in a single trip [13], [27] for two reasons. First, drivers in a ride hitching service have their daily commutes, and prefer a short detour during their trips. Serving multiple riders normally results in a long detour. Second, having multiple riders share a ride may result in unfriendly user experience, especially for riders with long-distance trips, because the driver has to pick up/drop off other riders many times at different locations during their trips. Hence, in this paper we focus on how to efficiently maintain the up-to-date top-k results for each driver serving only one rider. Nevertheless, our proposed algorithms can also be extended to serve multiple riders, as discussed in Section 4.5.

Remark that we have considered the time and distance constraints in our matching model. To evaluate the matching between riders and drivers, we consider: i) *time constraint*, meaning that a driver should pick up a rider within the rider's expected pick-up time window, and ii) *detour constraint*, meaning that the detour of the driver incurred by picking up and dropping off a rider should be less than the driver's tolerable detour. Without violating the time and distance constraints, the rider with a smaller detour distance has a higher ranking score.

Based on Definitions 1 and 2, we provide a running example to show how to maintain the top-k results for all subscribed *TkRS* queries when new/expired ride orders appear in the system.

Example 2. *Fig. 3(a) shows a road network where the solid lines are road segments and the numbers indicate road distances, and Fig. 3(b) and Fig. 3(c) show the origins and destinations of ride orders and TkRS queries. Initially, there exists a set of TkRS queries* $\mathcal{S} = \{s_1, s_2, s_3\}$ *and a stream of ride orders* $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_5\}$*. The top-2 results of each TkRS query, along with their ranking scores, are shown in the left of Fig. 3(d). Now, assume that* o_1 *expires and* o_6 *arrives. Obviously, by computing the ranking scores, the top-2 results of* s_1 *are updated from* $\{o_1, o_3\}$ *to* $\{o_6, o_3\}$ *and those of* s_2 *are updated from* $\{o_1, o_2\}$ *to* $\{o_2, o_4\}$*, while the top-2 results of* s_3 *remain unchanged since* o_4 *and* o_5 *are still the top-2 matched. The updated top-2 results of* s_1*,* s_2*, and* s_3 *are shown in the right of Fig. 3(d).*

Problem statement. Given a large number of *TkRS* queries and a ride-order stream, the *TkRS* query problem aims to develop an efficient and scalable solution to continuously maintain the up-to-date top-k results for all subscribed *TkRS* queries over the ride-order stream.

3.3 Baseline Methods

To handle the *TkRS* query problem, several existing techniques can be adapted and are briefed below. They are also adopted as baseline algorithms in Section 5.

Inverted index based method. This inverted index based approach is modified from a basic approach to process the publish/subscribe problem [2], [16]. It dynamically maintains a ride-order list $\mathcal{L}$, in which each ride order $o \in \mathcal{L}$ is associated with a set of *TkRS* queries whose top-k results contain o. When an expired ride order appears, it can quickly identify the affected *TkRS* queries whose top-k results need updates and recompute the up-to-date top-k results accordingly. When a new ride order arrives, however, it has to compare the ranking score of the new ride order with those of the top-k results for all subscribed *TkRS* queries, resulting in poor runtime performance.

Grid index based method. Unlike the inverted index based approach, existing works [19], [22], [23] adopt a grid index to accelerate the search speed, which divides the entire road network into a series of adjacent cells. Each cell is associated with a set of *TkRS* queries whose pick-up points locate in the cell. The merit of this approach is that it can accelerate the search speed in a group manner to retrieve the *TkRS* queries whose top-k results are affected. However, this method also generates a large number of candidates and thus increases the time cost of maintaining top-k results.

To address these weaknesses, we present efficient algorithms with several optimizations to tackle the *TkRS* query problem in the following section.

4 Proposed Solution

In this section, we first present a top-k initialization method to initialize the top-k results for *TkRS* queries. Then we offer two efficient algorithms equipped with several effective pruning rules and a novel index to maintain the top-k results. Finally, we propose an effective buffer mechanism to support efficient top-k result updates.

4.1 Top-K Initialization

As shown in the framework of the *TkRS* query service, in the top-k initialization step, the server assigns the k best matching ride orders to each subscribed TkRS query as the initial

Algorithm 1 Top-k Initialization

Input: *a TkRS query s, a ride-order set $\mathcal{O}$*
Output: *the top-k results of s*

1: Initialize a priority queue qu and a sorted k-size list $\mathcal{H}_s^k$;
2: **for** each ride order o in $\mathcal{O}$ **do**
3: **if** the matching of o and s is valid **then**
4: $qu.add(o, \kappa_{ub}(o, s))$;
5: **while** $qu \neq \emptyset$ **do**
6: Ride order $o \leftarrow qu.dequeue()$;
7: **if** the size of $\mathcal{H}_s^k$ is less than k **then**
8: $\mathcal{H}_s^k.add(o)$;
9: **else**
10: **if** $\kappa(o, s) > \kappa(o_s^k, s)$ **then**
11: Update $\mathcal{H}_s^k$ with o;
12: **if** $\kappa(o_s^k, s) \geq \kappa_{ub}(s, o)$ **then**
13: **break**;
14: **return** $\mathcal{H}_s^k$;

top-k results. It can be solved by our previous work [19]. Since the top-k initialization is the basic component of the *TkRS* query service, we briefly explain it for the sake of completeness. The pseudo code is given in Algorithm 1. The input augments are a *TkRS* query s and a ride-order set $\mathcal{O}$. We first initialize a priority queue qu in decreasing order of the ranking upper bound $\kappa_{ub}(o, s)$ and a sorted k-size list $\mathcal{H}_s^k$ in decreasing order of the ranking score $\kappa(o, s)$ to store the top-k results of s (Line 1). Here, $\kappa_{ub}(o, s)$ is computed by using the Euclidean distance as the lower bound of the shortest path distance between any two points in a road network. For each ride order $o \in \mathcal{O}$, if there is a matching between o and s, we compute the ranking upper bound $\kappa_{ub}(o, s)$ and enqueue o into qu according to $\kappa_{ub}(o, s)$ (Lines 2–4). Afterwards, for each ride order o in qu, if the size of $\mathcal{H}_s^k$ is less than k, we add o into $\mathcal{H}_s^k$. Otherwise, if the ranking score $\kappa(o, s)$ of o is larger than that of the k-th ride order o_s^k in $\mathcal{H}_s^k$, o_s^k is replaced with o. Finally, we compare $\kappa(o_s^k, s)$ with the ranking upper bound $\kappa_{ub}(o, s)$. If $\kappa(o_s^k, s)$ is larger than or equal to $\kappa_{ub}(o, s)$, the top-k results of s are found (Lines 5–14). In what follows, we focus on how to efficiently maintain the top-k results for all subscribed *TkRS* queries over a stream of ride orders.

4.2 TkRSMonitor Algorithm

In this section, we propose an efficient *TkRSMonitor* algorithm to answer the subscribed *TkRS* queries. In general, when a new/an expired ride order appears, we check each *TkRS* query in a sequential fashion and determine whose top-k results are affected. As illustrated in Section 1, such a process is cost prohibitive. To accelerate the checking and updating processes, we present two kinds of pruning rules from the perspectives of *service-guaranteed pruning* and *matching-bounded pruning* as follows.

4.2.1 Service-Guaranteed Pruning

According to Definition 2, the matching of a ride order and a *TkRS* query should guarantee the basic service requirements such as the waiting time and detour ratio constraints. That is, the driver should pick up the rider at the pick-up point within the expected pick-up time window (i.e., from t_o^p to $t_o^p + t_o^w$), and the detour ratio for a *TkRS* query serving a ride order should be less than the driver's tolerable detour ratio, guaranteed by Lemma 1.

Lemma 1. *Given a ride order o and a TkRS query s, o may affect the top-k results of s if the two conditions hold: (i) $t_s(l_o^p) \in w_o$ where $t_s(l_o^p)$ denotes the time when s arrives at l_o^p, $w_o = [t_o^p, t_o^p + t_o^w]$ denotes the pick-up time window; and (ii) $\pi(l_s^p, l_o^p) + \pi(l_o^p, l_o^d) + \pi(l_o^d, l_s^d) \leq (1 + \gamma_s) \cdot \pi(l_s^p, l_s^d)$.* ■

Here we omit the proof of Lemma 1 since it can be easily derived from the tolerable waiting time and detour ratio constraints. Lemma 1 indicates that the total travel distance for a driver taking a new rider should be less than the maximum travel distance upper bound $(1 + \gamma_s) \cdot \pi(l_s^p, l_s^d)$, which can efficiently prune the search space. On a separate note, frequently computing the shortest path distance $\pi(l_s^p, l_o^p)$ in a road network is time consuming, and thus we can use the Euclidean distance $\delta(l_s^p, l_o^p)$ as the lower bound of the road network distance $\pi(l_s^p, l_o^p)$. If the condition $\delta(l_s^p, l_o^p) + \delta(l_o^p, l_o^d) + \delta(l_o^d, l_s^d) \geq (1 + \gamma_s) \cdot \pi(l_s^p, l_s^d)$ holds, the top-k results of s cannot be affected.

According to Lemma 1, we can avoid updating the top-k results for the *TkRS* queries which cannot match the new ride order, reducing the updating cost. However, there may still be some *TkRS* queries whose top-k results cannot be affected even if they meet Lemma 1, because their ranking scores may be less than those of the top-k results. Thus, we next present the effective *matching-bounded pruning*.

4.2.2 Matching-Bounded Pruning

Besides the service-guaranteed pruning rule, we also attempt to detect the unaffected *TkRS* queries by comparing the ranking score of a new ride order with those of the top-k results. Given a new ride order o and a *TkRS* query s, if the ranking score of o is less than or equal to that of the k-th ride order o_s^k in the top-k results, i.e., $\kappa(o, s) \leq \kappa(o_s^k, s)$, the top-$k$ results of s cannot be affected. We formally give the travel distance upper bound of a new ride order w.r.t. a *TkRS* query in Theorem 1 to further improve the pruning capability.

Theorem 1. *The travel distance upper bound of a new ride order o for a given TkRS query s, denoted by $\pi_{ub}^s(l_o^p, l_o^d)$, is*

$$\pi_{ub}^s(l_o^p, l_o^d) = \frac{\pi(l_s^p, l_s^d)}{\kappa(o_s^k, s)} - \delta(l_s^p, l_o^p) - \delta(l_o^d, l_s^d). \blacksquare \quad (2)$$

Proof. The proof can be found in Appendix A.1. □

Given a ride order o given a *TkRS* query s, the travel distance upper bound of o with regard to s means the maximum trip distance for o to be a ride order in the top-k results of s. By adopting the travel distance upper bound, we can derive an efficient pruning rule, guaranteed by Lemma 2, to filter out the *TkRS* queries whose top-k results do not need updates. The proof of Lemma 2 can be easily derived from Theorem 1 and thus is omitted here.

Lemma 2. *A new ride order o cannot affect the top-k results of a given TkRS query s if $\pi(l_o^p, l_o^d) \geq \pi_{ub}^s(l_o^p, l_o^d)$.* ■

Algorithm 2 TkRSMonitor Algorithm

Input: *a ride order o, a TkRS query set $\mathcal{S}$, an inverted ride-order list $\mathcal{L}$*
Output: *the updated top-k results of $\mathcal{S}$*

1: Initialize an empty *TkRS* query set $\mathcal{S}_o$;
2: **if** o is a new ride order **then**
3: **for** each *TkRS* query s in $\mathcal{S}$ **do**
4: **if** o and s satisfy Lemma 1 **then**
5: Compute $\pi^s_{ub}(l^p_o, l^d_o)$ based on Equation 2;
6: **if** $\pi(l^p_o, l^d_o) \leq \pi^s_{ub}(l^p_o, l^d_o)$ **then**
7: **if** $\kappa(o, s) > \kappa(o^k_s, s)$ **then**
8: Update $\mathcal{H}^k_s$ with o;
9: **if** o is an expired ride order **then**
10: $\mathcal{S}_o \leftarrow$ the *TkRS* queries of $\mathcal{S}$ in $\mathcal{L}$ associated with o;
11: **for** each *TkRS* query s in $\mathcal{S}_o$ **do**
12: Update $\mathcal{H}^k_s$ by Algorithm 1;
13: **return** the updated top-k results of $\mathcal{S}$;

4.2.3 TkRSMonitor Algorithm

Algorithm 2 shows the pseudo code of the *TkRSMonitor* algorithm. The input arguments include a ride order o, a *TkRS* query set $\mathcal{S}$, and an inverted ride-order list $\mathcal{L}$. It is worth noting that each ride order o in $\mathcal{L}$ is associated with a set of *TkRS* queries whose top-k results contain o. We continuously maintain $\mathcal{L}$ over a ride-order stream when new/expired ride orders appear in the system. At first, we initialize an empty *TkRS* query set $\mathcal{S}_o$ (Line 1) and then handle the input ride order o in two cases. If o is a new ride order, we check and filter out the *TkRS* queries which violate the trip constraints guaranteed by Lemma 1 (Lines 4–5) and the travel distance upper bound guaranteed by Lemma 2 (Lines 5–6). Thereafter, if the ranking score $\kappa(o, s)$ of o is larger than that of the k-th ride order o^k_s in the top-k results $\mathcal{H}^k_s$, we replace o^k_s with o (Lines 7–8). If o is an expired ride order, we retrieve the inverted ride-order list $\mathcal{L}$ and add the affected *TkRS* queries that are associated with o into $\mathcal{S}_o$ (Line 10). For each affected *TkRS* query $s \in \mathcal{S}_o$, Algorithm 1 is invoked to renew the top-k results (Lines 11-12).

Theorem 2. *Algorithm 2 correctly updates the top-k results of all subscribed queries in $\mathcal{S}$ when new/expired ride orders appear.* ■

Proof. The proof can be found in Appendix A.2. □

Cost analysis. In summary, the *TkRSMonitor* algorithm can efficiently and correctly keep the freshness of top-k results based on the proposed pruning rules. The time complexity of Algorithm 2 is $O(|\mathcal{S}| + |\mathcal{S}_o| \cdot \Theta(|\mathcal{O}|))$ where $|\mathcal{S}|$ is the total number of *TkRS* queries, $|\mathcal{S}_o|$ is the number of *TkRS* queries affected by an expired ride order, and $\Theta(|\mathcal{O}|)$ is the time cost of top-k initialization. However, the disadvantage is that we have to examine the top-k results for all subscribed *TkRS* queries, resulting in high maintenance cost.

4.3 Advanced TkRSMonitor

For the *TkRSMonitor* algorithm, if the number of affected *TkRS* queries is large, sequentially updating the top-k results of each *TkRS* query is still inefficient. To further enhance the processing performance, we propose an advanced *TkRSMonitor* algorithm where the unaffected *TkRS* queries can be pruned in a group manner. The advanced

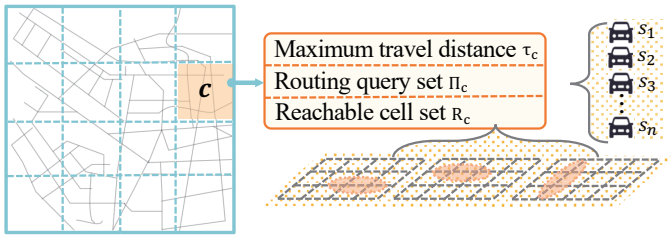


Fig. 4: The structure of the *HG* index

TkRSMonitor algorithm is achieved by designing an efficient and effective hybrid grid index. In what follows, we first illustrate the hybrid grid index structure and then describe how to build and update this index. Finally, we present the advanced *TkRSMonitor* algorithm by integrating the proposed index structure.

4.3.1 Efficient Hybrid Grid Index

Indexing is a natural and efficient approach for spatial data processing problems [7], [17]–[19], [23]. As such, we design a novel hybrid grid index structure denoted by *HG* to index all *TkRS* queries registered in the system. Fig. 4 shows the basic structure of an *HG* index, where each internal entry is represented as a cell or a square. It partitions the road network into a set of continuous cells. Each cell represents an area of the road network and is associated with a set of *TkRS* queries that may reach this cell. The basic usage of an *HG* index rests with the efficient filtering of *TkRS* queries which cannot arrive at riders' pick-up and drop-off locations in time. To further enhance the pruning capability, each cell in an *HG* index is coupled with three pieces of clustering information of the associated *TkRS* queries to support fast pruning. Specifically, given a grid cell c, the three pieces of information associated with c include: (i) a set of *TkRS* queries Π_c where the moving area of each *TkRS* query in Π_c contains c, (ii) a set of reachable cells $\mathcal{R}_c$ which are reachable for the *TkRS* queries in Π_c, and (iii) the upper bound of the maximum travel distance τ_c for the *TkRS* queries in Π_c. Before explaining these variables, we first introduce the definition of the representation of a *TkRS* query, which is used to calculate the values of these variables.

Due to the waiting time and detour ratio constraints, the moving range of a *TkRS* query is restricted to a small reachable area in the road network. Inspired by this observation, we represent each *TkRS* query as a set of time-marked cells in the grid. Fig. 5 illustrates the representations of two *TkRS* queries, where the time-marked cells covered by the ellipses denote the representations of the *TkRS* queries, and the arrows denote the driving directions of the drivers. A formal definition of the representation of a *TkRS* query is given below.

Definition 3. *(Representation of a TkRS query) The representation of a TkRS query s is a set of grid cells $\mathcal{C}_s$ where each cell $c \in \mathcal{C}_s$ is associated with a time range $t_c = [t^-_c, t^+_c]$ indicating the earliest time t^-_c of s arriving in c and the latest time t^+_c of s departing from c, and the combination of cells in $\mathcal{C}_s$ can minimally cover the ellipse $\mathcal{E}_s$ with two focal points l^p_s and l^d_s such that any point l on or within the curve of $\mathcal{E}_s$ satisfies*

$$\delta(l, l^p_s) + \delta(l, l^o_s) \leq (1 + \gamma_s) \cdot \pi(l^p_s, l^d_s). \blacksquare \quad (3)$$

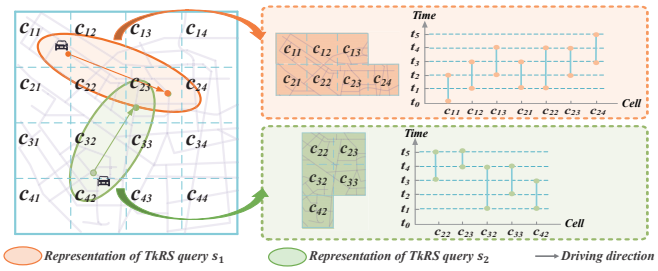


Fig. 5: The representation of a *TkRS* query

Definition 3 derives from the concept of ellipse, which denotes a plane curve surrounding two focal points, such that for all points on or within the curve, the sum of two distances to the focal points is less than or equal to a constant. Based on Definition 3, we can derive the spatial-temporal range that the driver of a *TkRS* query may reach. Thus, the three pieces of clustering information coupled with a grid cell c are detailed as follows.

(1) Routing query set Π_c. A routing query set Π_c is a set of *TkRS* queries that may move across c based on Definition 3. For example, in Fig. 5, the routing *TkRS* query set of cell c_{23} is $\Pi_{c_{23}} = \{s_1, s_2\}$, since both of the representations of s_1 and s_2 contain c_{23}. With Π_c, we can quickly identify the *TkRS* query which moves across the cells containing the pick-up/drop-off points of a new ride order. It is of great importance to improve the filtering efficiency.

(2) Reachable cell set $\mathcal{R}_c$. To further enhance the pruning capability, we track the reachable cells $\mathcal{R}_c$ of all subscribed *TkRS* queries moving across the cell c. The reachable cell set $\mathcal{R}_c$ can be calculated as $\mathcal{R}_c = \bigcup_{i=1}^{|\mathcal{S}_c|} \mathcal{C}_{s_i}$, where $|\mathcal{S}_c|$ denotes the number of *TkRS* queries across c. Note that a cell c' in $\mathcal{R}_c$ couples with several time ranges since there may be more than one *TkRS* query crossing c', resulting in maintenance overhead, e.g., storage and computation. In order to balance the efficiency and effectiveness, we recursively combine the time ranges coupled with c' if the gap of any two time ranges is less than a user-specified parameter Δ, e.g., 30 minutes. Furthermore, we equally divide the whole time range into a set of time bins $\Lambda_c = \{\lambda_1, \lambda_2, \cdots, \lambda_m\}$, e.g., one hour, then each cell $c' \in \mathcal{R}_c$ can be mapped into λ_i if the time range of c' overlaps λ_i. Therefore, $\mathcal{R}_c$ can be expressed as a set of cell subsets that are composed of $\mathcal{R}_{\lambda_i}$ $(i \in [1, m])$ where the time range of each cell in $\mathcal{R}_{\lambda_i}$ overlapping λ_i is not empty. The reachable cell set $\mathcal{R}_c$ provides a mechanism to pinpoint the *TkRS* queries whose top-k results need updates. Given a grid cell c and a new ride order o, if the pick-up point l_o^p locates in $c_1' \in \mathcal{R}_c$, the drop-off point l_o^d locates in $c_2' \in R_c$, and the pick-up time window of l_o^p overlaps the time bin associated with c_1', the top-k results of all *TkRS* queries in $\mathcal{R}_c$ need updates.

Note that we make use of travel speed bounds to compute the representation of a *TkRS* query. This is practical because the travel speed bounds are constant and predictable in the road network. We adopt the travel speed bounds to calculate the earliest arrival time and the latest departure time for each reachable grid cell. As such, we can easily precompute the variable $\mathcal{R}_c$ regardless of how the speed changes.

Example 3. *In Fig. 5, there are two TkRS queries* s_1 *and* s_2*. Each cell of* Π_{s_1} *and* Π_{s_2} *is marked with a time range. Assume that a new ride order* o *with pick-up time window* $w_o = [t_3, t_4]$ *and the pick-up point* $l_o^p \in c_{23}$ *arrives. It is obvious that* s_2 *cannot reach* c_{23} *in time since the time range of* s_2 *moving over* c_{23} *is* $t_{s_2}^{c_{23}} = [t_4, t_5]$*, so the top-k results of* s_2 *do not need updates.*

(3) Maximum travel distance τ_c. By definition, the tolerable detour ratio of a *TkRS* query dominates its maximum travel distance. Thus, given a new ride order o and a *TkRS* query s, if the maximum travel distance of s is less than the minimum travel distance of o, s cannot match o. To efficiently find out such *TkRS* queries, we use an aggregate variable τ_c to index the upper bound of the maximum travel distance of all *TkRS* queries stored in c, leading to the pruning rule in Lemma 3.

Lemma 3. *Given a new ride order* o *and a grid cell* c*, none of TkRS queries in* Π_c *needs to update the top-k results if* $\pi(l_o^p, l_o^d) > \tau_c$ *holds.* ■

Proof. The proof can be found in Appendix A.3. □

Note that the representation of a *TkRS* query (i.e., driver) can still be maintained even if a *TkRS* query shares a trip with other ride orders, because the representation is derived based on the driver's tolerable detour, limiting the driver's maximum reachable area.

Index construction and update. As stated above, the *HG* index can enhance the processing capability of order dispatch, and the improved performance will be verified in Section 5. Here we elaborate how to build and update an *HG* index. Specifically, we compute the routing query set and the reachable cell set for each grid cell by a union operation on the stored *TkRS* queries, and the maximum travel distance by selecting the upper bound of the maximum travel distance of each *TkRS* query stored in the routing *TkRS* query set. Based on the *HG* index, when receiving a new ride order, we can filter out the *TkRS* queries that do not need updates, which substantially cuts down the computation cost. Next, we discuss the issue of *HG* index update. An *HG* index needs to be updated only under the circumstances when a driver registers a new *TkRS* query or new/expired ride orders appear in the system. The updating processes of routing query set, reachable cell set, and maximum travel distance are the same as the building process.

Cost analysis. A critical task of building an *HG* index is to select a reasonable cell size of the grid, since the cell size determines the number of cells within the *HG* index and affects the processing performance. Here we discuss two extreme cases when the cell size equals the entire road network or minimally bounds a pick-up/drop-off point. Then the advanced *TkRSMonitor* algorithm will degrade into the *TkRSMonitor* algorithm, leading to worse performance. To address this issue, we propose a method to select a proper cell size of the grid for the best pruning capability of an*HG* index. We adopt a probabilistic model to estimate the expected dispatch cost for different cell sizes. When dispatching a new ride order o, the dispatch cost $cost_d(o)$ consists of the filtering cost $cost_f(o)$ and the verification cost $cost_v(o)$, i.e.,

$$cost_d(o) = cost_f(o) + cost_v(o). \tag{4}$$

The filtering cost $cost_f(o)$ is mainly determined by the number of *TkRS* queries, i.e.,

$$cost_f(o) = \nu_f \cdot \sum_{c \in g} p(c) \cdot ns(c), \tag{5}$$

where ν_f is the average cost of checking the matching of a ride order and a *TkRS* query by our proposed pruning techniques, $p(c)$ and $ns(c)$ denote the possibility of a ride order and the number of *TkRS* queries appearing in a cell c, respectively. We can easily derive $p(c)$ and $ns(c)$ from the historical data. For the verification cost $cost_v(o)$, we assume that the cost of verifying whether the top-k results of a *TkRS* query need to be updated is ν_v, then we have

$$cost_v(o) = \nu_v \cdot |\mathcal{S}'|, \tag{6}$$

where $|\mathcal{S}'|$ is the average number of remaining *TkRS* queries after the filtering step. Thus, given a grid g, the cost of processing a ride order o is

$$cost_d(o, g) = \nu_f \cdot \sum_{c \in g} p(c) \cdot ns(c) + \nu_v \cdot |\mathcal{S}'|. \tag{7}$$

The cost model of order dispatch guides how to measure the impact of a cell size on the performance of an *HG* index. We take a greedy approach to find the optimal cell size from the historical data. Similar to the quad-tree approach [25], we recursively subdivide the entire road network G into grid g_i with 4^i cells ($i \in \mathbb{N}$) until the conditions $cost_d(o, g_i) \leq cost_d(o, g_{i-1})$ and $cost_d(o, g_i) \leq cost_d(o, g_{i+1})$ hold. Then, the cell size of g_i is the optimal cell size of the *HG* index. Note that the optimal cell size can be computed offline, and only needs to be calculated once. Thus, it does not affect the online processing performance. Assume that the cell size of g_0 is $\mathcal{A}$ and that the algorithm is terminated forthwith when the road network is divided into 4^i cells. The optimal cell size of the *HG* index is $\mathcal{A}/4^i$.

4.3.2 Advanced TkRSMonitor Algorithm

The pseudo code of the advanced *TkRSMonitor* algorithm is given in Algorithm 3. It takes as input a ride order o, a set of *TkRS* queries $\mathcal{S}$, an inverted ride-order list $\mathcal{L}$, and an *HG* index *idx*. We first initialize a cell set *CS*, a *TkRS* query set $\mathcal{S}_o$, and a ride-order set $\mathcal{O}_s^o$ (Line 1). If o is a new ride order, we conduct order dispatch for all affected *TkRS* queries in two stages: *filtering* and *verification*. In the filtering stage, we select the cell c_1 containing the pick-up point l_o^p and the cell c_2 containing the drop-off point l_o^d (Line 3). After that, for each cell c of *idx*, if c_1 and c_2 are both in the reachable cell set $\mathcal{R}_c$ and $\pi(l_o^p, l_o^d) \leq \tau_c$, we add c into *CS* (Lines 5–7), i.e., only the *TkRS* queries in *CS* need to update their top-k results, guaranteed by Lemma 3. Now, for each *TkRS* query $s \in$ *CS*, we further check if the pick-up and drop-off points of o are both in the reachable cell set $\mathcal{C}_s$. If yes, we continue to check if s matches o without violating their trip constraints based on Lemma 1 (Lines 10–11). Next, we compute the trip distance upper bound $\kappa_{ub}(o, s)$ (Line 12). If the trip distance $\pi(l_o^p, l_o^d)$ of o is less than or equal to $\pi_{ub}^s(l_o^p, l_o^d)$, we compare the matching ratio $\kappa(o, s)$ of o with that of the k-th ride order o_s^k in $\mathcal{H}_s^k$ (Line 13). If $\kappa(o, s) > \kappa(o_s^k, s)$, we replace o_s^k with o (Lines 14–15).

Next, we explain how to maintain *TkRS* queries when o is an expired ride order. Comparing with the *TkRSMonitor* algorithm, the difference is that, for each affected *TkRS* query $s \in \mathcal{S}_o$, the candidates of the top-k results are in the ride-order set $\mathcal{O}_s^o$, guaranteed by Theorem 3, instead of the active order pool (Lines 18–20). Hence, making use of an *HG* index, we can substantially accelerate the top-k updating process for each affected *TkRS* query.

Algorithm 3 Advanced TkRSMonitor Algorithm

Input: *a ride order o, a TkRS query set $\mathcal{S}$, an inverted ride-order list $\mathcal{L}$, an HG index idx*
Output: *the updated top-k results of $\mathcal{S}$*

```
1: Initialize a cell set CS, a TkRS query set S_o, and a ride-order set O_s^o;
2: if o is a new ride order then
3:     Select the cells c_1 containing l_o^p and c_2 containing l_o^d;
4:     for each cell c in HG idx do
5:         if c_1 and c_2 are both in R_c then
6:             if π(l_o^p, l_o^d) ≤ τ_c then
7:                 CS.add(c);
8:     for each cell c in CS do
9:         for each TkRS query s in Π_c do
10:            if l_o^p and l_o^d are not both in C_s then
11:                if o and s satisfy Lemma 1 then
12:                    Compute π_ub^s(l_o^p, l_o^d) based on Equation 2;
13:                    if π(l_o^p, l_o^d) ≤ π_ub^s(l_o^p, l_o^d) then
14:                        if κ(o, s) > κ(o_s^k, s) then
15:                            Update H_s^k with o;
16: if o is an expired ride order then
17:     S_o ← the TkRS queries of S in L associated with o;
18:     for each TkRS query s in S_o do
19:         Compute O_s^o based on Theorem 3;
20:         Compute H_s^k from O_s^o by Algorithm 1;
21: return the updated top-k results of S;
```

Theorem 3. *Given a TkRS query s, the candidate ride-order set to replenish an expired ride order o in $\mathcal{H}_s^k$, denoted by $\mathcal{O}_s^o$, should satisfy $\forall o' \in \mathcal{O}_s^o$, $\exists\ c_1, c_2 \in \mathcal{C}_s$, s.t., $l_{o'}^p \in c_1$, $l_{o'}^d \in c_2$, and $w_{o'} \cap t_s^{c_1} \neq \emptyset$.* ■

The proof of Theorem 3 can be derived from Definition 3 and is omitted here. By Theorem 3, the search space of the top-k maintenance can be significantly reduced, leading to better processing performance. The correctness of Algorithm 3 is guaranteed by Theorem 4, whose proof is similar to that of Theorem 2.

Theorem 4. *Algorithm 3 can correctly update the top-k results of all subscribed TkRS queries when new/expired ride orders appear in the system.* ■

Cost analysis. Compared with Algorithm 2, the unqualified *TkRS* queries can be pruned in a batch manner by the *HG* index. The time complexity of the advanced *TkRSMonitor* algorithm is $O(max\{|\Pi_c| \cdot |CS|, |\mathcal{S}_o| \cdot \Theta(|\mathcal{O}_s^o|)|\})$ where $|\Pi_c| \cdot |CS|$ and $|\mathcal{S}_o| \cdot \Theta(|\mathcal{O}_s^o|)$ are the top-k maintenance costs when a new ride order arrives and an existing ride order expires, respectively. Due to the effective pruning capability of the *HG* index, numerous unmatched ride orders are pruned so that $|\Pi_c| \cdot |CS|$ is much smaller than $|S|$. Hence, the performance of the advanced *TkRSMonitor* algorithm is expected to be much better than that of the *TkRSMonitor* algorithm.

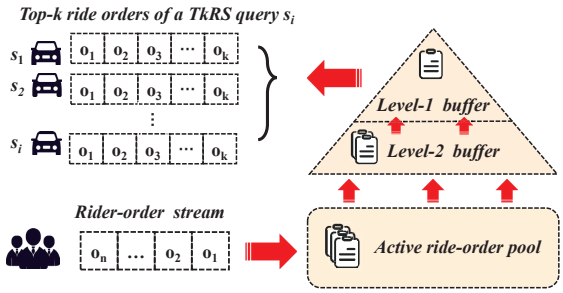


Fig. 6: The two-level buffer structure

4.4 Optimizations

By adopting the novel *HG* index structure, we can efficiently process the affected *TkRS* queries in a group manner. When an existing ride order expires, the top-k initialization module is invoked to search the top-k results for all affected *TkRS* queries in the candidate ride-order set $\mathcal{O}_s^o$ rather than in the active ride-order pool (i.e., the whole search space), which substantially accelerates the updating speed. To further reduce the top-k maintenance cost, we design a two-level buffer structure for each *TkRS* query, where the first-level buffer contains the ride orders whose pick-up and drop-off points are in the reachable cell set and the second-level buffer includes the ride orders that are obtained by the skyband computation [24] over the ride orders in the first-level buffer. Note that the skyband used in the work [24] is only one component of our proposed OPT solution, which is originally designed to process textual data and thus cannot support spatial data processing well. Fig. 6 shows the basic structure of the two-level buffer. When there is an expired ride order, we remove it from the top-k results of all affected *TkRS* queries and replenish it with the most matched ride order from the second-level buffer. If there is no matched ride order in the second-level buffer, we need to renew the top-k results from the first-level buffer and update the second-level buffer accordingly.

For the first-level buffer, given a *TkRS* query s, the ride orders with pick-up and drop-off points in the representation of s are contained the first-level buffer. Thus, we can easily build the first-level buffer by adopting the representation of the *TkRS* query, which is guaranteed by Theorem 3. Next, we focus on how to build the second-level buffer by computing the *k-skyband*. Here, we first introduce the definitions of *dominance* and *k-skyband* in the context of the *TkRS* query problem.

Definition 4. *(Dominance) Given a TkRS query s and two ride orders o_1 and o_2, o_1 dominates o_2 if $\kappa(o_1, s) \geq \kappa(o_2, s)$ and $t^p_{o_1} \geq t^p_{o_2}$.* ■

Consider a *TkRS* query s_1 and three ride orders o_1, o_2, and o_3. Assume $\kappa(o_1, s_1) \geq \kappa(o_2, s_1) \geq \kappa(o_3, s_1)$ and $t_{o_1} \geq t_{o_2} \geq t_{o_3}$. By Definition 4, o_2 and o_3 are dominated by o_1 over s_1. Next, we introduce *k-skyband* based on Definition 4.

Definition 5. *(k-skyband) Given a TkRS query s and a ride-order set $\mathcal{O}$, the k-skyband of $\mathcal{O}$ w.r.t. s, denoted by $\mathcal{B}_s^k$, is a ride-order set in which each ride order $o \in \mathcal{B}_s^k$ is dominated by less than k ride orders in $\mathcal{O} - \{o\}$.* ■

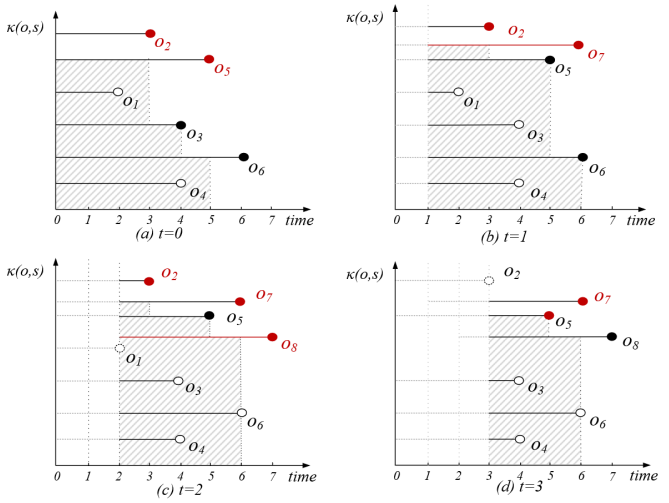


Fig. 7: An example of a 2-skyBand buffer

Theorem 5. *Given a TkRS query s and a ride-order set $\mathcal{O}$, the top-k results $\mathcal{H}_s^k$ must be in the k-skyband $\mathcal{B}_s^k$.* ■

Proof. The proof can be found in Appendix A.4. □

Theorem 5 tells that the top-k results are a subset of the k-skyband of s. The k-skyband provides a mechanism to cache the most possible candidates for those ride orders that are about to expire in the top-k results.

Example 4. *In Fig. 7, the x-axis and the y-axis denote the time t and the ranking score $\kappa(o, s)$, respectively. In the initial stage (i.e., at time $t = 0$ as illustrated in Fig. 7(a)), there is a TkRS query s and a ride-order set $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_5, o_6\}$ in the system. The ranking score $\kappa(o, s)$ and the arrival/expired time of each ride order $o \in \mathcal{O}$ are shown in Fig. 7(a). The top-2 results of s are $\mathcal{H}_s^k = \{o_2, o_5\}$, and the 2-skyband of $\mathcal{O}$ is $\mathcal{B}_s^k = \{o_2, o_5, o_3, o_6\}$.*

At time $t = 1$ (Fig. 7(b)), the new ride order o_7 arrives. Since $\kappa(o_7, s) > \kappa(o_5, s)$, the top-2 results are updated to $\mathcal{H}_s^k = \{o_2, o_7\}$. Now, o_3 is dominated by o_5 and o_7. Since the number of dominating orders of o_3 exceeds 1, o_3 is removed from the 2-skyband $\mathcal{B}_s^k$, and $\mathcal{B}_s^k$ is updated to $\mathcal{B}_s^k = \{o_2, o_7, o_5, o_6\}$;

At time $t = 2$ (Fig. 7(c)), the new ride order o_8 arrives. Although o_8 cannot affect the top-2 results $\mathcal{H}_s^k$, it incurs the updating of $\mathcal{B}_s^k$ because o_6 is dominated by o_7 and o_8. Thus, the updated 2-skyband is $\mathcal{B}_s^k = \{o_2, o_7, o_5, o_8\}$.

At time $t = 3$ (Fig. 7(d)), the existing ride order o_2 expires, and it is removed from $\mathcal{H}_s^k$. Then o_5 with the highest ranking score is selected to replenish the top-2 results $\mathcal{H}_s^k = \{o_7, o_5\}$, and the 2-skyband of s is also changed to $\mathcal{O}_s^k = \{o_7, o_5, o_8\}$ accordingly. ■

For the top-k maintenance, the critical operation is to maintain a small set of candidates in the second-level buffer. For instance, given a *TkRS* query, we continuously maintain the k-skyband which can efficiently replenish the expired ride orders in the top-k results. When an expired ride order appears, the system selects the ride order with the highest ranking score from the two-level buffer and substitutes it for the expired one in the top-k results. Hence, maintaining the candidates of the top-k results transfers to a task of k-skyband maintenance. However, maintaining the k-skyband over all ride orders in the first-level buffer is also

cost-prohibitive. To solve this issue, we define a partial k-skyband in Definition 6, which is used to reduce the candidate space for building the second-level buffer.

Definition 6. *(Partial k-skyband) A partial k-skyband, denoted by $\widehat{\mathcal{B}_s^k}$, is a subset of $\mathcal{B}_s^k$ such that each ride order $o \in \widehat{\mathcal{B}_s^k}$ satisfies $\kappa(o,s) \geq \kappa_s^b$, where κ_s^b is the ranking score of the k-th ride order in $\mathcal{H}_s^k$ when invoking the most recent top-k computation.* ■

According to Definition 6, we know that the top-k results $\mathcal{H}_s^k$ are also a subset of $\widehat{\mathcal{B}_s^k}$. Therefore, we have the following lemma.

Lemma 4. *The ride order replacing the expired ride order of $\mathcal{H}_s^k$ must be in $\widehat{\mathcal{B}_s^k}$ if the size of $\widehat{\mathcal{B}_s^k}$ is larger than k.* ■

The proof of Lemma 4 is omitted since it can be derived from Defintion 6. Lemma 4 suggests that a partial k-skyband can indeed reduce the candidate space.

Algorithm 4 gives the pseudo code of the optimized process to continuously maintain the two-level buffer and the top-k results. The input arguments include a *TkRS* query s whose top-k results should be maintained, a new/an expired ride order o, and a threshold κ_s^b used for ranking score comparison where the value of κ_s^b is the ranking score of the k-th ride order when invoking the most recent top-k computation. In Algorithm 4, the maintenance of the top-k results and two-level buffer of s is invoked in two cases: (1) When o is a new ride order, if the matching of o and s exists and the ranking score satisfies $\kappa(o,s) \geq \kappa_s^b$, we further check each ride order o' in $\widehat{\mathcal{B}_s^k} \cup \{o\}$. If the number of dominating ride orders of o' is larger than or equal to k, o' should be removed from $\widehat{\mathcal{B}_s^k}$. Otherwise, we add o into $\widehat{\mathcal{B}_s^k}$ (Lines 3–15). Next, if the ranking score of o is larger than that of the k-th ride order o_s^k, we add o into $\mathcal{H}_s^k$ (Lines 16–17). (2) When o is an expired ride order and $o \in \mathcal{H}_s^k$, we first remove o from $\widehat{\mathcal{B}_s^k}$ and $\mathcal{H}_s^k$. If the size of $\widehat{\mathcal{B}_s^k}$ is no less than k, we add the ride order o' with the highest $\kappa(o',s)$ from $\widehat{\mathcal{B}_s^k}$ into $\mathcal{H}_s^k$ as the supplement (Lines 20–22). Otherwise, if the size of $\widehat{\mathcal{B}_s^k}$ equals k-1, we need to recalculate the top-k results $\mathcal{H}_s^k$ from $\mathcal{O}_s^o$, i.e., the first-level buffer, by invoking the top-k initialization algorithm and refresh the corresponding threshold κ_s^b (Lines 25–26). By adopting the proposed two-level buffer, we can quickly update the top-k results for all affected *TkRS* queries when expired ride orders appear in the system.

Cost analysis The two-level buffer indeed accelerates the maintenance of top-k results for each *TkRS* query. The time complexity of Algorithm 4 is $O(|\mathcal{B}_s^k| + \Theta(|\mathcal{O}_s^o|))$. Compared with the advanced *TkRSMonitor* algorithm, the optimized algorithm can substantially improve the processing performance at the cost of slightly more memory usage, which will be experimentally verified in Section 5.

Algorithm 4 Optimized Processing Approach

Input: *a TkRS query s, a ride order o, and a threshold κ_s^b*
Output: *the partial k-skyband $\widehat{\mathcal{B}_s^k}$*

```
1: if o is a new ride order then
2:     if the matching of o and s is valid then
3:         if κ(o,s) ≥ κ_s^b then
4:             for each ride order o' ∈ B̂_s^k do
5:                 if κ(o,s) ≤ κ(o',s) and t_o ≤ t_o' then
6:                     o.num ← o.num + 1;
7:                     if o.num ≥ k then
8:                         break;
9:             if o.num < k then
10:                for each ride order o' ∈ B̂_s^k do
11:                    if κ(o',s) ≤ κ(o,s) and t_o' ≤ t_o then
12:                        o'.num ← o'.num + 1;
13:                    if o'.num ≥ k then
14:                        B̂_s^k.remove(o');
15:                B̂_s^k.add(o);
16:            if κ(o,s) ≥ κ(o_s^k, s) then
17:                Update H_s^k with o;
18: if o is an expired ride order in B̂_s^k then
19:     Remove o from B̂_s^k and H_s^k;
20:     if |B̂_s^k| ≥ k then
21:         Select the ride order o' ∈ B̂_s^k with the best κ(o',s);
22:         H_s^k.add(o');
23:     else
24:         Compute H_s^k from O_s^o by Algorithm 1;
25:         B̂_s^k ← H_s^k;
26:         κ_s^b ← κ(o_s^k, s);
27: return B̂_s^k;
```

4.5 Serving Multiple Ride Orders

It is easy to extend our algorithms to serve multiple ride orders in a single *TkRS* query. In general, when a driver serves multiple riders, the driver can select the riders independently in a round-robin approach, i.e., the driver selects the riders one after another. At first, when a driver has not started to serve any rider, the system notifies the driver the first batch of top-k matched riders. Since now there is only one moving pattern for the driver, the ranking score is calculated according to Equation 1. After the driver selects a rider from the first batch of the top-k matched riders, the system will compute and notify the driver of the second batch of top-k matched riders. In this case, we need to plan a route for the pick-up and drop-off points of the already onboarded riders and the new rider such that the driver's actual total detour is minimum. Therefore, the ranking function to rank a new rider o is defined as $\kappa(o,s) = \frac{\pi(l_s^p, l_s^d)}{MinD(s,o)}$, where *MinD(s, o)* denotes the minimum travel distance when the driver s serves the new rider o and its already onboarded riders. Here, we adopt the method of point insertion presented in [9], [31] to plan the order of the pick-up and drop-off points for all onboarded riders and the new rider o. Using the above method, the drivers can select the riders to share the remaining seats.

In addition, there is no difference in building or updating an *HG* index and a two-level buffer between serving only one ride order and serving multiple ride orders. This is because no matter how many ride orders a *TkRS* query serves, the pick-up/drop-off points and the pick-up time windows of the eligible ride orders should satisfy the constraints of the representation of a given *TkRS* query.

5 Performance Evaluation

In this section, we evaluate the performance of different algorithms on two real datasets. As explained in Section 3.3, the inverted index based method (referred to as *IM*) and the grid index based method (referred to as *GM*) are considered the baselines. The *TkRSMonitor Algorithm* presented in Section 4.2 is referred to as *TA*, the *Advanced Algorithm*

TABLE 2: Dataset properties

Items	*ChengDu*	*NewYork*
total # of ride orders	*200K*	*200K*
total # of TkRS queries	*50K*	*50K*
total # of intersections	*36,630*	*264,346*
total # of roads	*50,786*	*366,923*
total # of trips	*200K*	*200K*

TABLE 3: Parameter settings

Parameters	*Value*	*Default*
k	*1, 2, 3, 4, 5*	*3*
# of ride orders	*30K, 50K, 100K, 150K, 200K*	*100K*
# of TkRS queries	*10K, 20K, 30K, 40K, 50K*	*30K*
maximum waiting time (mins)	*3, 5, 10, 15, 20*	*10*
tolerable detour ratio	*0.1, 0.2, 0.3, 0.4, 0.5*	*0.3*
capacity of vehicle	*1, 2, 3, 4, 5*	*3*

introduced in Section 4.3 is referred to as *ADV*, and the two-level buffer based *Optimized Algorithm* presented in Section 4.4 is referred to as *OPT*. We evaluate the above methods in terms of average elapsed time, memory cost, and throughput, which are critical to a ride hitching service.

5.1 Experimental Settings

We evaluate the algorithms over two real trajectory datasets collected from *NYCTaxi*[1] and *DiDi*[2], which are widely used to evaluate ridesharing services. Table 2 shows the properties of these two datasets. The *NYCTaxi* dataset contains one month's trajectories, including 1,445,285 trips in March, 2017 in *New York*, and trajectories released by *DiDi* include 853,156 trips in *Cheng Du* over one month starting in January, 2016. We map the starting and ending points of each trip to their nearest road intersections on the road network according to the latitude and longitude coordinates. We select the mapped starting point and ending point as the pick-up point (origin) and drop-off point (destination) of the ride order (*TkRS* query), respectively. Meanwhile, we treat the timestamp of starting point as the pick-up time (departure time) of the ride order (*TkRS* query). We randomly generate a certain number of *TkRS* queries and a stream of ride orders based on the released trajectories. Note that the data is retrieved from the time range from 7:00am to 9:00am (i.e., one of the peak ordering periods) in normal weekdays. In the default setting, 13.8 ride orders are issued per second and 30K *TkRS* queries are registered in the system. We extract the road network information from *OpenStreetMap*[3] to construct the underlying road networks for the cities of *New York* and *Cheng Du*. The experimental parameter settings are summarized in Table 3, where k denotes the number of returned ride orders for each *TkRS* query. All the algorithms are implemented in *Java* and evaluated on a PC with an *Intel i7-8700 @ 3.60HZ CPU* and *16GB DDR4 RAM*.

5.2 Experimental Results

Effect of the number of ride orders. The first set of experiments is designed to examine the performance of different algorithms under varying numbers of ride orders. Fig. 8 shows the average elapsed time of different algorithms under different numbers of ride orders. We can observe that the proposed algorithms *ADV* and *OPT* perform better than the algorithms *TA*, *IM* and *GM*, and that they are less sensitive to the increase of the number of ride orders. The performance of *ADV* and *OPT* is still reasonably good even when the number increases to $200k$, demonstrating the effective pruning capability of *HG* indexes. It is worth noting that, due to the effective two-level buffer, the candidates to the expired ride orders in the top-k results are cached, and thus many repeated computations are reduced in the process of top-k maintenance, leading to the best performance of *OPT*.

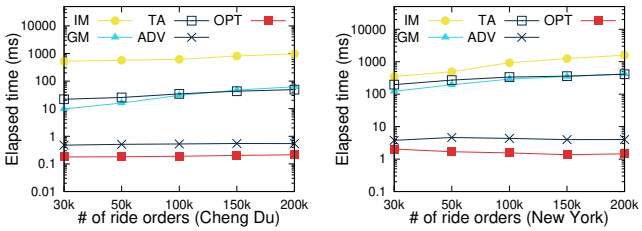


Fig. 8: Elapsed time vs. # of ride orders

Effect of k. In this set of experiments, we evaluate the performance of all algorithms under various k values (i.e., the number of top-k results maintained for a *TkRS* query). As plotted in Fig. 9, with the increase in k, all algorithms spend more time to maintain the top-k results, because the greater the value of k, the more candidates need to be maintained. It can be observed that *ADV* and *OPT* achieve the best update performance under all k values due to the good pruning capability of *HG* indexes. Moreover, compared with *ADV*, *OPT* just needs to retrieve all the ride orders from the two-level buffer instead of the reachable cell set of each *TkRS* query, and thus requires less time than *ADV* when updating the top-k results for all affected *TkRS* queries.

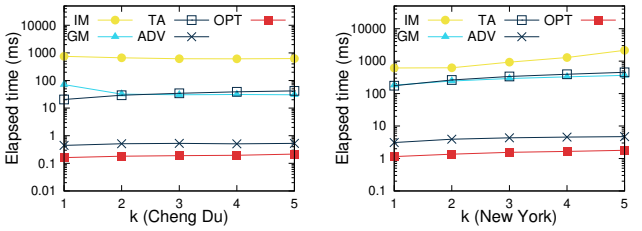


Fig. 9: Elapsed time vs. the value of k

Effect of the number of TkRS queries. Next, we evaluate the performance of different algorithms by varying the number of *TkRS* queries. Intuitively, a larger number of *TkRS* queries results in a larger candidate space for the top-k maintenance. From Fig. 10, it can be seen that the time cost of all algorithms increases when the number of *TkRS* queries becomes larger. As expected, thanks to the strong pruning capability of *HG* indexes and the two-lever buffer, *ADV* and *OPT* achieve better performance than that of *TA*, *IM*, and *GM*, with up to nearly 100X performance improvement.

Effect of the waiting time. In this set of experiments, we investigate how the waiting time affects the performance of the algorithms. Fig. 11 plots the performance pattern when varying the length of the waiting time. It can be observed that all algorithms need more time to examine and

[1]http://www.nyc.gov
[2]https://gaia.didichuxing.com
[3]https://www.openstreetmap.org

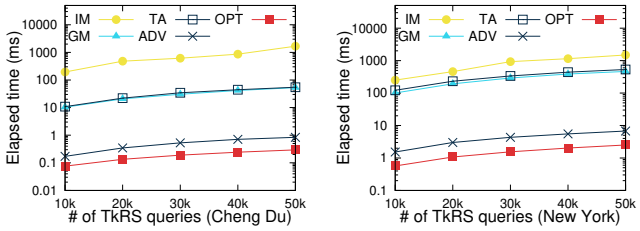


Fig. 10: Elapsed time vs. # of *TkRS* queries

update the top-k results of all affected *TkRS* queries with the increase in the waiting time. The reason is that, when the waiting time gets longer, there are more *TkRS* queries becoming eligible to serve a given ride order. It follows that the number of affected *TkRS* queries increases a lot when a new or an expired ride order with longer waiting time appears in the system, resulting in more updating cost. Again *ADV* and *OPT* have better performance due to the good pruning capability of *HG* indexes and the two-level buffer.

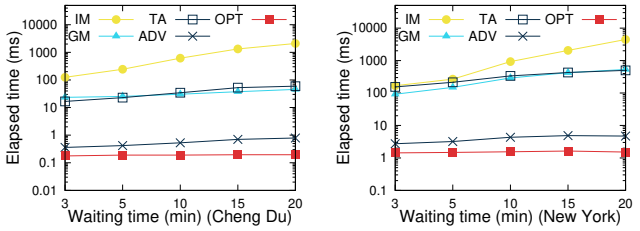


Fig. 11: Elapsed time vs. the waiting time

Effect of the detour ratio. We next consider the performance of the algorithms under different values of detour ratios. Intuitively, a larger detour ratio offers a *TkRS* query more chance to serve more eligible ride orders, which in turn enlarges the search space. In Fig. 12, we can observe that the results are consistent with our theoretical analysis that the time cost of all algorithms considerably degrades with the increase in the detour ratio. In all settings, *ADV* and *OPT* still outperform the other three algorithms due to better pruning capability.

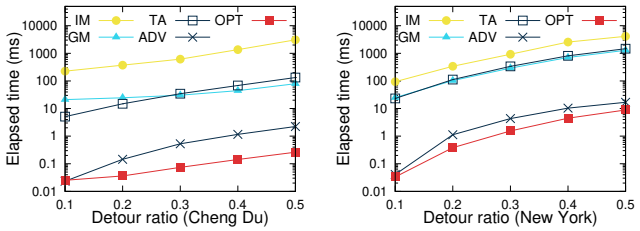


Fig. 12: Elapsed time vs. the detour ratio

Memory costs of different methods. As discussed earlier, indexes and buffers play an important role in improving the processing performance of our proposed algorithms. Thus, in the last set of experiments, we examine the memory costs of different methods under the default parameter settings. Table 4 gives the memory costs of different methods. *GM*, *TA*, *ADV*, and *OPT* require more memory costs than that of *IM*, while the benefit is that the additional memory cost greatly improves the processing performance of our proposed algorithms. Compared with *GM* and *TA*, *ADV* and *OPT* achieve two orders of magnitude smaller running time at the expense of just about 3 times space cost, proving the effectiveness of our proposed methods.

TABLE 4: Memory costs of different methods

Datasets	IM	GM	TA	ADV	OPT
New York	866MB	1,453MB	1,196MB	2,253MB	2,446MB
Cheng Du	308MB	1,244MB	426MB	1,434MB	1,450MB

6 Conclusion and Future Work

In this paper, we proposed a novel *TkRS* query problem in a ride hitching system. To maintain the freshness of the top-k results for each *TkRS* query, we proposed several efficient algorithms with effective pruning techniques. Moreover, we also proposed an efficient *HG* index and an effective two-level buffer to further improve the processing performance.

As for future work, we plan to extend our current work in the following directions. First, we attempt to study the cross matching problem for ride hitching to integrate drivers and riders in different platforms. Second, we plan to study the data persistence or distributed computing techniques to solve the problem when the number of *TkRS* queries and ride orders cannot fit in memory.

Acknowledgments

This work is supported by NSFC Grants 61972362, 61602420, 62072136, and 62036010, HNSF Grant 202300410378, Fundamental Research Funds for the Central Universities Grant 3072020CFT2402, Guangdong Basic and Applied Basic Research Foundation Grant 2019B1515130001, and HK RGC Grants 12200817 and 12201018. Data source: Didi Chuxing.

References

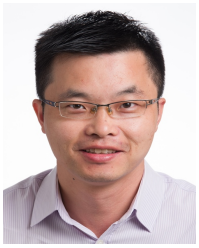
[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM PODS*, pages 1–16, 2002.
[2] L. Chen and G. Cong. Diversity-aware top-k publish/subscribe for text stream. In *ACM SIGMOD*, pages 347–362, 2015.
[3] L. Chen, G. Cong, and X. Cao. An efficient query indexing mechanism for filtering geo-textual data. In *ACM SIGMOD*, page 749–760, 2013.
[4] L. Chen, G. Cong, X. Cao, and K. Tan. Temporal spatial-keyword top-k publish/subscribe. In *IEEE ICDE*, pages 255–266, 2015.
[5] L. Chen, Y. Cui, G. Cong, and X. Cao. Sops: A system for efficient processing of spatial-keyword publish/subscribe. *PVLDB*, 7(13):1601–1604, 2014.
[6] L. Chen, Y. Gao, Z. Liu, X. Xiao, C. S. Jensen, and Y. Zhu. Ptrider: A price-and-time-aware ridesharing system. In *PVLDB*, pages 1938–1941, 2018.
[7] L. Chen, Y. Li, J. Xu, and C. S. Jensen. Towards why-not spatial keyword top-k queries: A direction-aware approach. *IEEE TKDE*, 30(4):796–809, 2018.
[8] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen. Price-and-time-aware dynamic ridesharing. In *IEEE ICDE*, pages 1061–1072, 2018.
[9] P. Cheng, H. Xin, and L. Chen. Utility-aware ridesharing on road networks. In *ACM SIGMOD*, pages 1197–1210, 2017.
[10] B. Cici, A. Markopoulou, E. Frias-Martinez, and N. Laoutaris. Assessing the potential of ride-sharing using mobile and social data: A tale of four cities. In *ACM UbiComp*, page 201–211, 2014.
[11] Didi. http://www.didiglobal.com.
[12] X. Fu, J. Huang, H. Lu, J. Xu, and Y. Li. Top-k taxi recommendation in realtime social-aware ridesharing services. In *SSTD*, pages 221–241, 2017.

[13] X. Fu, C. Zhang, H. Lu, and J. Xu. Efficient matching of offers and requests in social-aware ridesharing. In *IEEE MDM*, pages 197–206, 2018.
[14] Grab. http://www.grab.com.
[15] Y. Huang, F. Bastani, R. Jin, and X. Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 7(14):2017–2028, 2014.
[16] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *ACM SIGKDD*, pages 802–810, 2013.
[17] Y. Li, R. Chen, L. Chen, and J. Xu. Towards social-aware ridesharing group query services. *IEEE TSC*, 10(4):646–659, 2017.
[18] Y. Li, R. Chen, J. Xu, Q. Huang, H. Hu, and B. Choi. Geo-social k-cover group queries for collaborative spatial computing. *IEEE TKDE*, 27(10):2729–2742, 2015.
[19] Y. Li, J. Wan, R. Chen, J. Xu, X. Fu, H. Gu, P. Lv, and M. Xu. Top-k vehicle matching in social ridesharing: A price-aware approach. *IEEE TKDE*, pages 1–1, 2019.
[20] Z. Liu, Z. Gong, J. Li, and K. Wu. Mobility-aware dynamic taxi ridesharing. In *IEEE ICDE*, pages 1–14, 2020.
[21] Lyft. https://www.lyft.com.
[22] S. Ma and O. Wolfson. Analysis and evaluation of the slugging form of ridesharing. In *ACM GIS*, pages 64–73, 2013.
[23] S. Ma, Y. Zheng, and O. Wolfson. Real-time city-scale taxi ridesharing. *IEEE TKDE*, 27(7):1782–1795, 2015.
[24] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *ACM SIGMOD*, pages 635–646, 2006.
[25] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
[26] A. Shraer, M. Gurevich, M. Fontoura, and V. Josifovski. Top-k publish-subscribe for social annotation of news. *PVLDB*, 6(6):385–396, 2013.
[27] N. Ta, G. Li, T. Zhao, J. Feng, H. Ma, and Z. Gong. An efficient ride-sharing framework for maximizing shared route. *IEEE TKDE*, 30(2):219–233, 2018.
[28] Uber. https://www.uber.com.
[29] X. Wang, Y. Zhang, W. Zhang, X. Lin, and Z. Huang. Skype: Top-k spatial-keyword publish/subscribe over sliding window. *PVLDB*, 9(7):588–599, 2016.
[30] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *IEEE ICDE*, pages 1107–1118, 2015.
[31] Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu, and W. Li. An efficient insertion operator in dynamic ridesharing services. In *IEEE ICDE*, pages 1022–1033, 2019.
[32] M. Yu, G. Li, T. Wang, J. Feng, and Z. Gong. Efficient filtering algorithms for location-aware publish/subscribe. *IEEE TKDE*, 27(4):950–963, 2015.
[33] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 11(8):853–865, 2018.
[34] L. Zheng, P. Cheng, and L. Chen. Auction-based order dispatch and pricing in ridesharing. In *IEEE ICDE*, pages 1034–1045, 2019.

Yafei Li received the PhD degree in computer science from Hong Kong Baptist University, in 2015. He is currently a professor in the School of Information Engineering, Zhengzhou University, China. His research interests span data management and location-based services. He has authored more than 30 journal and conference papers in these areas, including IEEE TKDE, IEEE TSC, ACM TWEB, ACM TIST, PVLDB, IEEE ICDE, WWW, etc.

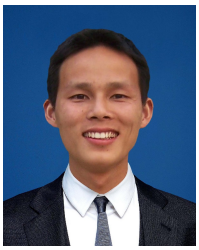
Hongyan Gu received the BEng degree in computer science and technology from Zhengzhou University, China, in 2017. She is currently working toward the MEng degree at the School of Information Engineering, Zhengzhou University. Her research interests include temporal and spatial data management, location-based services, and urban computing.

Rui Chen received the Ph.D. degree in Computer Science from Concordia University. He is a professor in the College of Computer Science and Technology, Harbin Engineering University. His research interests include machine learning, data privacy and databases. He has published more than 40 technical papers in top venues, including CSUR, VLDBJ, PVLDB, IEEE TKDE, IEEE TDSC, ACM KDD, ACM CCS and IEEE ICDE, and won CIKM 2015 Best Paper Runner Up, the Best Papers of ICDE 2016 and the Best Papers of ICDM 2018. He has served as program committee member for leading conferences, including ACM KDD, IEEE ICDM, and ACM CIKM, and as reviewer for numerous flagship journals, including VLDBJ, PVLDB, IEEE TKDE, IEEE TDSC, IEEE TIFS, and IEEE TOPS.



Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, and the PhD degree in computer science from the Hong Kong University of Science and Technology. He is a professor in the Department of Computer Science, Hong Kong Baptist University. He held visiting positions with Pennsylvania State University and Fudan University. His research interests include big data management, mobile computing, and data security and privacy. He has published more than 150 technical papers in these areas. He has served as a program cochair/vice chair for a number of major international conferences including IEEE ICDCS 2012, IEEE CPSNA 2015, and APWeb-WAIM 2018. He is an associate editor of the IEEE Transactions on Knowledge and Data Engineering and the Proceedings of the VLDB Endowment 2018.

Shangwei Guo received the PhD degree in computer science from Chongqing University, Chongqing, China. He worked as a postdoctoral research fellow at Hong Kong Baptist University from 2018 to 2019. He is now a research fellow in the School of Computer Science and Engineering, Nanyang Technological University. His research interests include secure cloud and edge computing and database security.



Junxiao Xue received the PhD degree from the School of Mathematical Sciences, Dalian University of Technology, China, in 2009. He is currently an associate professor in School of Software, Zhengzhou University, China. His research interests include big data management, virtual reality, and virtual simulation.

Mingliang Xu received the PhD degree from the State Key Lab of CAD&CG, Zhejiang University, China. He is a professor in the School of Information Engineering, Zhengzhou University, China. His current research interests include multimedia big data, computer graphics, and artificial intelligence. He has authored more than 60 journal and conference papers in these areas, including ACM TOG, IEEE TPAMI/TIP/TCSVT/TKDE, ACM SIGGRAPH (Asia)/MM, ICCV, etc.