Designing an On-Line Ride-Sharing System

Blerim Cici, Athina Markopoulou
University of California, Irvine, USA
{bcici, athina}@uci.edu

Nikolaos Laoutaris
Telefonica Research, Spain
nikolaos.laoutaris@telefonica.com

ABSTRACT

Ride-sharing systems have the potential to match travelers with similar itineraries and time schedules, and to bring significant benefits to individual users and the city as a whole. However, this is a challenging task, since users’ requests are not known in advance and they become available a few minutes before departure. In this paper, we design an online ride sharing system, where drivers and passengers send their requests for a ride in advance, possibly on a short notice. Our design is efficient and optimal. This is achieved by dividing the system into two components: the constraint satisfier and the matching module. The constraint satisfier takes as input the spatio-temporal constraints of drivers and passengers and provides feasible (driver, passenger) pairs in real time, and the matching module takes as input the feasible pairs and provides a maximum cardinality matching of drivers and passengers. Our preliminary evaluation shows that the constraint satisfier can resolve the most expensive queries (matching of passenger to en-route drivers) in 2 seconds (on average), while the matching module can achieve a matching ratio of 78% when the offline upper-bound is 80%.

Categories and Subject Descriptors
H.2.8 [Database Management]: Database Application—data mining, spatial databases

Keywords
Online ride-sharing, Intelligent transportation systems

1. INTRODUCTION

The car has been for some time one of the most heavily used ground transportation vehicles, and in the US, it is the dominant one. According to American Community Survey Reports currently there are more than 130 million commuters, and almost 80 of them drive alone when commuting [1]. This has many negative consequences: pollution, traffic, high car expenses, and loss of productivity.

Ride-sharing is a promising approach for reducing the number of vehicles on the streets in order to address both individual and city-wide issues. Ride-sharing refers to a mode of transportation in which individuals share a vehicle for a trip and split travel costs, and it combines the flexibility and speed of private cars with the reduced cost of shared public transportation.

An anticipated breakthrough in ride-sharing is the ability to satisfy on-demand requests that do not require participants to schedule their trips in advance [2]. Such an online system will provide a participant the reassurance that they would still be serviced if their travel-needs change unexpectedly; when asked how far ahead of time participants would like to organize a shared ride, 43% desired organizing their ride 15-60 minutes before departure [3].

In this paper, we are interested in such an online ride-sharing system: requests arrive dynamically, possibly with a short notice, rides are on-recurrent basis, trajectories and times need to be updated as cars move, and the system has to find a matching (or declare that no ride exists) in real-time. We design and implement an online ride sharing system for matching users that could share a ride. Our goal is to provide rides to as many users as possible (or equivalently to minimize the total numbers of cars), while respecting users’ spatiotemporal constraints (e.g. to not incur excessive waiting, delay or detour).

The system consists of two main components: (1) the constraint satisfier and (2) the matching module. The constraint satisfier takes as input the spatio-temporal constraints of drivers and passengers and provides feasible (driver, passenger) pairs. We design this module with scalability in mind. The matching module takes as input the feasible pairs (which form a bipartite graph between drivers and passengers) and responds to the users with their suggested ride. We formulate the problem as maximum cardinality matching, and we propose an optimal yet efficient algorithm for online matching (i.e. it finds the maximum cardinality matching, while performing efficient incremental updates upon arrival/expiration of requests). By decoupling the system into two components that interact only through the (dynamically updated) bipartite graph of feasible pairs, we build a system that is modular, easy to parallelize and to make online.

We evaluate our system using home/work locations and trajectories extracted from spatio-temporal datasets from the city of New York [4]. Our preliminary evaluation shows that the constraint satisfier can resolve the most expensive queries (matching of passenger to en-route drivers) in 2 seconds (on average), while the matching module can achieve a matching ratio of 78% when the offline upper-bound is 80%.
2. RELATED WORK

This section summarizes the most relevant work in the area of ride-sharing and in related areas.

Commercial Ride-Sharing. Ride-sharing is an important part of the sharing economy and provides economical, societal and environmental benefits by utilizing “empty” car seats. Relevant ride-sharing startups include Zimride and Scoop [5], whose focus, however, is to facilitate ride-sharing between employees of large corporations. This makes the problem less challenging than in the general case (and is thus a popular strategy for bootstrapping carpooling [6]), since users have the same destination (the company they all work for) and the system considers only the home locations of drivers and passengers, thus reducing the number of potential pairs. Their algorithms are proprietary and the problem they address is a special case of the framework developed in this paper.

Academic Research provides valuable insights into ride-sharing, primarily through surveys on ride-sharing optimization [7] and small scale ride-sharing demos [3]. The work in [3] reports how far in advance ride-sharing participants want to schedule their trips. Other studies characterized the behavior of carpoolers [8], identified the individuals who are most likely to carpool and explained what are the main factors that affect their decision [9]. Prior work quantified the potential of ride-sharing [6, 4] using offline analysis of datasets. For example, in our prior work [4], we evaluated the potential of carpooling considering end-point and en-route ride-sharing, and using mobile datasets from four cities to extract home and work locations. Excellent surveys on the formulation and optimization of ride-sharing and on the key computational challenges include [7] and [2]. The focus of this paper is online ride-sharing (as opposed to offline analysis of its potential) and system design.

3. SYSTEM OVERVIEW

3.1 System Requirements

We define ride-sharing as a one-time trip shared between a single driver and a single passenger, according to spatiotemporal constraints that both parties specify. Drivers and passengers submit their requests before their desired departure time; this ahead-of-time notification can be, for example, a few minutes before departure or the evening before the trip, and in general a parameter that affects performance. When ride requests are submitted, a search for potential matches takes place in real time. If a suitable match is found, the participants are notified immediately. An overview of the system is shown on Fig. 1.

3.2 Driver and Passenger Requests

Let S denote the set of all users, D denote the set of drivers and P denote the set of passengers. Clearly D ⊆ S, P ⊆ S and D ∪ P = S. A location is described with its coordinates (lat, long).

For every passenger p ∈ P, the request entered in the system consists of the following information, also depicted on Fig. 2:

- Source location: coordinates (lat\_p, long\_p).
- Earliest departure time: p is ready to leave at \( t_p^{(0)} \).
- Destination location: coordinates (lat\_p, long\_p).
- Latest arrival time: \( t_p^{(0)} \) is the latest acceptable time to arrive at the destination. Based on that, we can compute the latest departure time \( t_p^{(w)} \), i.e. the latest time she can leave from the source location in order not to be late; we refer to \( \Delta t_2 = t_p^{(b)} - t_p^{(w)} \) as the delay tolerance of the passenger.
- Request time: \( t_p^{(w)} \) is the time the passenger sends its request to the system, and it must be before the desired departure time. We refer to \( \Delta t_1 = t_p^{(b)} - t_p^{(s)} \) as the ahead-of-time notification and \( \Delta t_2 = t_p^{(h)} - t_p^{(b)} \) as delay tolerance (how much can p wait to be picked up).

![Figure 2: Request by passenger p: The vertical axis represents space (source and destination locations), the horizontal axis represents time and the dashed lines are example trajectories. At \( t_p^{(s)} \) the passenger sends his request in the system. The passenger is ready to leave from the source location at \( t_p^{(b)} \) and wants to arrive at the destination by \( t_p^{(h)} \); \( t_p^{(b)} \) is the latest time the passenger can leave and not to late. We refer to \( \Delta t_1 = t_p^{(b)} - t_p^{(s)} \) as ahead-of-time notification and \( \Delta t_2 = t_p^{(h)} - t_p^{(b)} \) as delay tolerance (how much can \( p \) wait to be picked up).](image)

The driver needs only to provide the source and destination and the system will present a list of available paths, e.g. obtained using Google Maps, and will ask her to choose one.

3.3 Driver and Passenger Constraints

Ride-sharing needs to be convenient for both the driver and the passenger: they shouldn’t deviate too much from their routine and they shouldn’t experience excessive delay or inconvenience.

Let us consider a given driver-passenger pair, \( d \) and \( p \), depicted on Fig. 3. We assume that the driver does not change trajectory or departure time; however, he is willing to deviate from his trajectory to pickup/dropoff the passenger, as depicted in Fig. 3.

- Request time: \( t_p^{(w)} \) the time the driver’s request was sent to the system, and it must be before the departure time; ahead-of-time notification is defined similarly.

The drivers and passengers are willing to deviate from their routes and destination locations, the horizontal axis represents time and the dashed lines are example trajectories. At \( t_p^{(s)} \) the passenger sends his request in the system. The passenger is ready to leave from the source location at \( t_p^{(b)} \) and wants to arrive at the destination by \( t_p^{(h)} \); \( t_p^{(b)} \) is the latest time the passenger can leave and not to late. We refer to \( \Delta t_1 = t_p^{(b)} - t_p^{(s)} \) as ahead-of-time notification and \( \Delta t_2 = t_p^{(h)} - t_p^{(b)} \) as delay tolerance (how much can \( p \) wait to be picked up).
Fig. 1: Online Ride-Sharing System Overview. Drivers and passengers enter their requests. The "constraint satisfier module" finds candidate pairs and builds a bipartite graph. The matching module takes the bipartite graph as an input, produces a matching. Finally, drivers and passengers are notified.

Figure 3: Example of Spatio-Temporal constraints when matching a passenger $p$ with a driver $d$. The passenger $p$ leaves from her source location (home $H$) and is going to a destination (work $W$). The driver has a fixed trajectory and departure time. However, the driver considers deviating at different points on his trajectory and do a detour (indicated in dashed line) to pickup and dropoff the passenger, as long as these detour distances do not exceed his distance tolerance: i.e., $\text{dist}_H(p, d, i) \leq \delta$ and $\text{dist}_W(p, d, j) \leq \delta$. In exchange, the passenger may wait until his latest departure time.

passenger and driver constraints are satisfied, i.e.:

$$w(d, p) = \begin{cases} 
1, & \text{if } t_p^{(h)} > t_d^{(i)} + \text{delay}_H(p, d, j) \\
& \text{and } t_p^{(w)} < t_d^{(j)} + \text{delay}_W(p, d, j) \\
& \text{and } \max(\text{dist}_H(p, d, i), \text{dist}_W(p, d, j)) < \delta \\
0, & \text{otherwise}
\end{cases}$$

A core challenge in ride sharing is to find feasible passenger-driver pairs and points for pick-up drop-off on the driver's trajectory, that meet all constraints. The response to such search queries must be fast, for the ride-sharing system to be real-time and scale with the number of users.

3.4 System Architecture

Fig. 1 shows an overview of the architecture of the system, which consists of two main components. The first component is the constraint satisfier that takes as input the passengers and drivers' requests and produces feasible passenger-driver pairs, which can be summarized in the bipartite graph, shown in the middle of Fig. 1. The second component is the matching module that takes as input the bipartite graph of feasible pairs and finds a maximum cardinality matching (as described in Section 5).

An important aspect of our system is that it is online: requests arrive dynamically (at times $t_d^{(i)}$, $t_d^{(j)}$, respectively) and also can expire (when a driver arrives at the destination $t_d^{(n,i)}$, or after the latest departure time of a passenger $t_p^{(h)}$). When arrival/expiration events happen, the two modules need to do incremental updates.

More specifically, the first module needs to update the records in the database (driver’s trajectory points and passengers’ source, destination and constraints) and the bipartite graph of feasible pairs. The second module needs to update the matching solution, based on the changes in the bipartite graph.

4. CONSTRAINT SATISIFIER

This component of the online ride-sharing system receives the queries from the drivers and the passengers, and will generate the feasible matching combinations, i.e. for every passenger, it will generate the set of drivers who meet her spatio-temporal constraints.

Choosing MongoDB. The speed and scalability are our primary concerns, and we do not require complex join queries that traditional databases offer. We considered different options for the constraint-satisfier and we decided to implement it using a NoSQL database. More specifically, we choose MongoDB [10], which is very popular in the industry (e.g. used by Foursquare [11]). We choose MongoDB due to its support for spatial data. More specifically, MongoDB offers the following:

- **Spatial Indexing**: MongoDB offers supports for queries that calculate geometries on an earth-like sphere, through the 2dsphere index – a grid based geohashing scheme – [12].
- **Fast data Insertion**: MongoDB is designed for fast insertion speed; it employs the cache of the operating system, which significantly reduces write costs. Fast insertion of data is very important for a real-time system, since while data are being inserted in the database, the process that is doing the insertion will lock the data – via a write lock – and during that time no other process can read or write anything. This means that one cannot take advantage of parallelization of insertion queries (which can be easily done for read queries, because they use a read-lock that allows other processes to read from the dataset).

Implementation details: In order to take advantage of the 2dsphere and the proximity queries that come with it, we build and store our data in such a way that all <latitude, longitude> points – both for coordinates that represent source/destination points and coordinates that represent points in a route.

- **Home/Work Collections**: We store the data of the passengers in two collections: 1) Home collection for the source points, and 2) Work collection for the destination points.
- **Trajectory Collection**: We store the trajectory points of the driver in a different Trajectory collection. Each trajectory point is an individual entry in the collection, indexed by the id of the user and its position in the trajectory (e.g. first, second, third, ..., or last point).
- **Spatio-temporal queries**: When we query one of the collections (either a diver seeking for passengers using the Home and Work collections, or a passenger seeking for drivers using the Trajectory
Figure 4: Home/Work locations and route points for NY.

5. MATCHING DRIVERS-PASSENGERS

In this section, we focus on the matching module in Fig. 1. This takes as input the bipartite graph of feasible driver-passenger pairs and provides a matching.

Maximum Cardinality Matching (MCM.) Consider the bipartite graph of feasible pairs: $G = (D \cup P, E)$ where $E = \{(d, p) : d \in D, p \in P\}$ such that the constraints of $d, p$ as defined in Section 3.3 are satisfied. Finding the Maximum Cardinality Matching (MCM) on this bipartite graph is a classic problem that can be solved efficiently (in $O(\min(|D|, |P|) \cdot |E|)$ time) and optimally using augmenting paths [13].

This classic (Ford-Fulkerson) algorithm lends itself naturally to an online version that can handle arrivals and departures of requests. Indeed, arrival of driver/passenger requests lead to edges appearing/disappearing from the bipartite graph, which can be handled by efficient incremental updates. Indeed, every time a request arrives, this results in one or more edges appearing in the bipartite graph. All we need to do is to find an augmenting path in the new auxiliary graph and update the existing matching (in $O(|E|)$), as opposed to solving the problem from scratch.

6. PRELIMINARY RESULTS

<table>
<thead>
<tr>
<th>City</th>
<th>NY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>61K</td>
</tr>
<tr>
<td>Inter-point distance</td>
<td>100m</td>
</tr>
<tr>
<td>Average distance</td>
<td>16.1 km</td>
</tr>
<tr>
<td>Median distance</td>
<td>8.0 km</td>
</tr>
<tr>
<td>Average gps points per trajectory</td>
<td>78.8</td>
</tr>
</tbody>
</table>

Table 1: Data set summary

In order to assess the performance of our on-line ride-sharing algorithm using real data, we used datasets that provide the home and work locations, as well as trajectories, of a large number of users in metropolitan areas. The data are summarized in Tab. 1 and was obtained from [4].

- **Constraint Satisfier:** We add all the users – their trajectories and their home/work locations – in MongoDB database (version 2.6.7).

Then we measure the average query speed for the most expensive operation, i.e. queries to the Trajectory collection – passengers looking for drivers. Our results indicate that the average query speed is around 2 seconds. Assuming a 32 parallel processes, then we can answer 16 queries per second and we would need an hour and 5 minutes for all the users in our database (assuming their queries arrived in our system uniformly at random).

- **Matching Algorithm:** To evaluate the matching algorithm we do a discrete time simulation where all the events appear in a simulated timeline based on the order of their arrival. Assuming an ahead of time notification of 20 minutes, our algorithm has a matching ration of 78% when the offline upper bound is 80%.

7. CONCLUSION AND FUTURE WORK

Ride-sharing has a great potential for reducing the number of cars in the streets of a city according to recent studies [4]. In this paper, we designed an online ride-sharing system. In future work, we plan to do a thorough and in depth evaluation of our system. We will explore better the relationship between dataset size and speed query speed for the constraint satisfier, and we will show how our matching algorithm is doing when compare to a greedy heuristics. Finally, we plan to explore how to speed up queries, e.g. by having sparser trajectories, and how does that affect the performance of the algorithm.

8. REFERENCES

[12] “Scaling mongodb at foursquare.”