SORS: A Scalable Online Ridesharing System

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

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

ABSTRACT

Ridesharing systems match travelers with similar trajectories and have the potential to bring significant benefits to individual users as well as to the city as a whole. In this paper, we design and evaluate SORS– a scalable online ridesharing system, where drivers and passengers send their requests for a ride in advance, possibly on a short notice. SORS is modular and consists of two main, loosely coupled, components: the Constraint Satisfier and the Matching Module. The Constraint Satisfier takes as input information about the desired trajectories and spatio-temporal constraints of drivers and passengers and it returns a list of feasible (driver, passenger) pairs. We use a road networks data structure, optimized for the specific spatio-temporal queries in the context of ridesharing, and we show that our Constraint Satisfier has a 4.65x more scalable query time than a general-purpose database. We represent the feasible pairs of drivers and passengers as a weighted bipartite graph with edge weight being the length of the shared trip of the pair, which captures the revenue in real-world ridesharing systems, such as Lyft Carpool. The Matching Module then takes as input this weighted bipartite graph and returns the maximum weighted matching (MWM), using an algorithm that solves the problem online and efficiently, by incrementally updating the matching solution in real-time. We show that our algorithm achieves 51% larger weight (*i.e.*, total revenue) compared to greedy heuristics used by many real systems today. We also evaluate the SORS system as a whole, using mobile datasets to extract driver trajectories and passenger locations in urban environments. We show that SORS can provide a ridesharing recommendation to individual users within a sub-second query response time, even at high workloads.

1. INTRODUCTION

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

Ridesharing is a promising approach for reducing car usage in the spirit of sharing economy: individuals share a vehicle for a trip and split travel costs. Ridesharing combines the flexibility and speed of private cars with the reduced cost of public transportation. Recently, companies such as Uber and Lyft announced ridesharing pilots targeted to commuters. Uber recently announced uberHOP and uberCOMMUTE [2]. Lyft announced Lyft Carpool [3]. We would like to emphasize that these ridesharing services are different from the taxi-like services typically provided *e.g.*, by uberPOOL [4], which allows taxi passengers to share a similar route. These pilot services came out recently, and independently of our work, and validate the importance of *ridesharing* (as opposed to taxi-like) systems, which are the focus of this paper.

In this paper, we design and evaluate such a *scalable online ridesharing system* – SORS. Requests for sharing a ride arrive over time, possibly within a short notice, and expire after a certain time or when users find a ride. Our focus is also on *online* ridesharing, where participants do not need to schedule their trips well in advance [5], since studies have shown that late trip scheduing is an important feature for the users; for example, when asked how far ahead of time they would like to organize a shared ride, 43% of people answered 15-60 minutes before departure [6]. Our system dynamically matches drivers and passengers and provides ridesharing recommendations accordingly, so as to both meet individual user constraints and to maximize a global objective, namely the total revenue for the system. We design SORS as a modular system consisting of two main, loosely connected, components: Constraint Satisfier and Matching Module, as depicted on Fig. 1.

The first component, the Constraint Satisfier, takes as input the itineraries and spatio-temporal constraints of individual drivers and passengers and provides a list of feasible (driver, passenger) pairs that satisfy those constraints. A key challenge in this module lies in its scalability. To that end, we use a road networks data structure, specifically optimized for the spatio-temporal queries in the context of ridesharing. We show that our Constraint Satisfier system is more scalable than what can be built using state-of-art but generic spatio-tempotal databases: *e.g.*, query time increases with the number of users, but 4.65x slower when compared to MongoDB. We represent the feasible pairs found by Constraint Satisfier as bipartite graph between passengers and drivers, with edge weights representing the length of the pair's shared trip. The latter is motivated by the revenue model that today's commercial systems (such as Lyft Carpool) use to charge for shared rides.

The second component, the Matching Module, takes as input the weighted bipartite graph (which already satisfies individual users' constraints) and returns the maximum weighted matching (MWM) that maximizes a global objective (the total weight represents the systems' revenue). A key challenge in the design of this component is the inherent computational complexity of the MWM problem, which is NP-hard. We propose an efficient algorithm, Online MWM, and we show that it achieves a total weight 51% higher than greedy heuristics used by many real systems today.

A common challenge that both components face is the *online* na-

IWCTS'16, October 31-November 03 2016, Burlingame, CA, USA
© 2016 ACM. ISBN 978-1-4503-4577-4/16/10…$15.00
DOI: http://dx.doi.org/10.1145/3003965.3003971

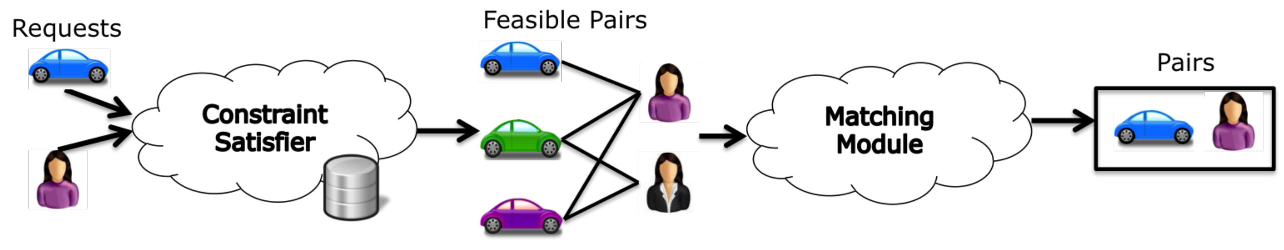

Figure 1: SORS Overview. Drivers and passengers enter their requests: the driver provides her trajectory, the passenger provides her source and destination, both provide their tolerance in deviating from their trajectory and timeline. The Constraint Satisfier finds feasible (driver,passenger) pairs that satisfy individual users' constraints. It builds a weighted bipartite graph; the weight of an edge corresponds to the length of the shared trip, which translates to revenue. The Matching Module takes the bipartite graph as an input and produces a matching that maximizes the total weight (revenue for the system). Finally, drivers and passengers are notified with ridesharing recommendations.

ture of problem. SORS's modular design can handle efficiently the requests that arrive or expire in an online fashion, by utilizing efficient queries of feasible pairs, dynamically updating the weighted bipartite graph, and incrementally updating the matching solution accordingly (Online MWM uses augmenting path methods to update the current optimal matching). We evaluated the entire SORS system using mobile datasets to extract realistic mobility patterns (namely driver trajectories and passenger locations) in urban environments. Our evaluation shows that SORS can provide a ridesharing recommendation to individual users with a sub-second query response time, even at high workloads.

The structure of the paper is as follows. Section 2 summarizes related work. Section 3 provides a system overview. Section 4 and 5 describe the two main system components: the Constraint Satisfier and the Matching Module, respectively. Section 6 presents evaluation results and Section 7 concludes the paper.

2. RELATED WORK

Commercial Ridesharing. Ridesharing startups include *Zimride* and *Scoop* [7] facilitate ridesharing between employees of large corporations. This makes the problem less challenging than in the general case, since users have the same destination (the company they all work for) and the same arrival time, and the system has to match only the home locations of drivers and passengers, thus reducing the number of potential pairs. Recently, Uber and Lyft, which are primarily on-demand taxi companies, announced ridesharing services for commuters: Uber announced uberHOP and uberCOMMUTE [2] and Lyft announced Lyft Carpool [3]. The algorithms used in those services are proprietary and it is not possible to compare against them. However, they serve as a guideline for our SORS, and we formulate our ridesharing problem to be in line with them.

On-demand Taxi-sharing (*e.g.*, the main services of Uber and Lyft) use smartphone devices to schedule trips between drivers and passengers. They employ dedicated drivers who have no strict spatial or temporal constraints. Taxi-sharing [8, 9] resembles online ridesharing, but since it does not consider spatial and temporal constraints for the driver, requires different algorithms. Both Uber and Lyft offer cheaper versions of their on-demand taxi services that include ridesharing: Uber offers uberPOOL and Lyft offers LyftLine. These two services facilitate ridesharing for taxi passengers who ride along a similar route, and they do not takes into account spatial and temporal constraints for the driver. Therefore, on-demand taxi-sharing is very different than ridesharing for commuters.

Academic Research provides valuable insights into ridesharing, primarily through surveys on ridesharing optimization [10] and small-scale ridesharing demos [6]. The work in [6] reports how far in advance ridesharing participants want to schedule their trips. Other studies characterized the behavior of ridesharing participants [11], identified the individuals who are most likely to share a ride and explained what are the main factors that affect their decision [12]. Prior work quantified the potential of ridesharing [13, 14] using *offline* analysis of datasets. Excellent surveys on the formulation and optimization of ridesharing and on the key computational challenges include [10] and [5]. The focus of this paper is online ridesharing (as opposed to offline analysis of its potential) and system design (design and evaluation of SORS that needs to run in real time).

Comparison to Our Prior Work: A preliminary version of this work appeared as short paper/poster in SigSpatial 2015 [15]. It presented the high-level idea of splitting the functionality into two loosely coupled modules: the Constraint Satisfier and the Matching Module. However, the design of each module was off-the-shelf and there was only toy evaluation. In this long paper, we build on and improve over [15], but there are clear new contributions. First, we design a specialized Constraint Satisfier: it uses a state-of-the-art data structure specialized to our problem (the road network) to store latitude and longitude coordinates of users moving in a city. Compared to SatisfierDB in [15], which was built using MongoDB, Constraint Satisfier is more scalable: the query time grows 4.65 slower with the number of users than with MongoDB. Second, we changed the formulation of the Matching Module that selects pairs of drivers-passengers to share a ride. In [15], the goal was to maximize the total number of passenger-driver pairs (maximum cardinality matching), while the goal here is to maximize the total distance drivers and passengers travel together (maximum weight matching). This new objective is closer to real-world systems (*e.g.*uberHOP and uber COMMUTE charge proportionally to the shared trip length) but the MWM problem becomes computationally hard, while MCM is known to be solved efficiently. To solve MWM efficiently and online, we designed a new Online MWM that is 14.4% better than our MCM 0-1 Algorithm in [15]. Finally, this paper provides a comprehensive evaluation of the whole system using a realistic workload of user requests extracted from spatio-temporal datasets. We show that SORS can provide a ridesharing recommendation within a few seconds from receiving a request, even under heavy workload, while increasing the system revenue by 51% compared to current commercial systems; we study the sensitivity of the system to various parameters, such as *ahead-of-time notification*, and we provide useful insights for designing and deploying real-world ridesharing systems.

3. SYSTEM OVERVIEW

3.1 System Requirements

We define ridesharing as a one-time trip shared between one driver

Notation	Definition
$(lat_p^{(h)}, lng_p^{(h)})$	Source location for p (“S” or ”H”)
$(lat_p^{(w)}, lng_p^{(w)})$	Destination location for p (“D” or ”W”)
$t_p^{(h)}$	Earliest acceptable departure time for p
$t_p^{(w)}$	Latest acceptable arrival time for p
$t'^{(h)}_p$	Latest acceptable departure time for p (it depends on $t_p^{(w)}$)
$\Delta t_2 = t'^{(h)}_p - t_p^{(h)}$	Delay Tolerance of p
$t_p^{(r)}$	Time when p sends request to SORS
$\Delta t_1 = t_p^{(h)} - t_p^{(r)}$	*ahead-of-time notification*
δ	Distance Tolerance.
T_d	Trajectory for driver d: ordered list of (location, time) points
$(lat_d^{(0)}, lng_d^{(0)}, t_d^{(0)})$	Starting point of d’s trajectory
$(lat_d^{(n_d)}, lng_d^{(n_d)}, t_d^{(n_d)})$	Ending point of d’s trajectory

Table 1: Notation for passenger p and driver d.

and one passenger, which is the most common case.[1] The driver specifies a trajectory and the passenger specifies a source (“S”) and a destination (“D”).[2] The driver can pick-up the passenger along his way and drop-off closer to the passenger’s destination. Notice the asymmetry in our definition of ridesharing: we want to match a driver’s trajectory with a passenger’s source and destination locations,[3] subject to certain spatiotemporal constraints. 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. Finally, the passenger pays the driver according to the length or duration of the shared trip (*e.g.*, to cover fuel expenses) and the ridesharing system keeps a percentage of that payment. The longer the shared part of the trip the higher the revenue. Therefore, from SORS’s perspetive, it is desirable to match drivers to passengers so as to maximize the total revenue.

Notation. Let S denote the set of all users, D denote the set of drivers and P denote the set of passengers. Clearly $D \subseteq S$, $P \subseteq S$ and $D \cup P = S$. A location is described with its coordinates (lat, lng). For every passenger $p \in P$, the request entered in the system consists of the following information. For every driver $d \in D$, the request entered in the system consists of the following information, also depicted on Fig. 2. Drivers’ inputs, passengers’ inputs, and ridesharing parameters are summarized in Tab. 1.

3.2 Driver and Passenger Constraints

Ridesharing 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. 2. We assume that the driver does not change trajectory or departure time; however, he is willing to do a small detour to pickup and drop off the passenger, as long as that detour does not exceed his *distance tolerance* δ. Let i be the pick up (*i.e.* closest point of d’s trajectory to the home of p), and j be the drop off (*i.e.* closest

[1]Lyft Carpool allows only one extra passenger so everyone gets to their destination as quickly as possible. 77% of nationwide carpools in 2000 involved one driver and one passenger, [13].

[2]In the evaluation section, we consider a use case where the source is the home (“H”) and the destination is the work (“W”), thus the notation H/W is used interchangeably with S/D in *some* figures. However, we would like to emphasize that our framework can address arbitrary source/destination locations and trajectories.

[3]Of course, SORS can use the source and destination to infer the trajectory of the passenger’s trajectory as well, *e.g.*, by using Googlemaps.

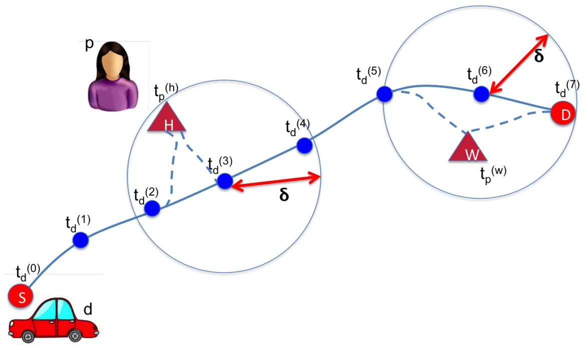

Figure 2: Example of spatio-temporal constraints when matching a passenger p with a driver d. The passenger p leaves from her source location (home H) and is going to a destination (work W). The driver d has a fixed trajectory (indicated in solid line) and departure time; each point of the trajectory contains an arrival time. The driver d can deviate from her trajectory (indicated in dashed line) to pick-up/drop-off the passenger p, as long as the deviation does not exceed her *distance tolerance*: *i.e.*, $dist_H(p, d, i) \leq \delta$ and $dist_W(p, d, j) \leq \delta$. In exchange, the passenger may wait until his latest departure time.

point of d’s trajectory to the work of p) location, and $i < j$. The passenger conveniently gets a ride, picked up at his source (*e.g.* home H) and dropped off at his destination (*e.g.* work W) location. In exchange, he may have to wait and delay his departure up to his latest departure time $t'^{(h)}_p$ in order to arrive by the latest arrival time $t_p^{(w)}$. Let $delay_H(p, d, i)$ and $delay_W(p, d, j)$ be the pick-up and drop-off delays respectively.A pair $(d, p) \in E$ is feasible iff both the passenger and driver constraints are satisfied, *i.e.*:

$$w(d,p) = \begin{cases} dist(i,j), \quad \text{if} \quad t_p^{(h)} > t_d^{(i)} + delay_H(p,d,j) \\ \quad \text{and} \quad t_p^{(w)} < t_d^{(j)} + delay_W(p,d,j) \\ \quad \text{and} \quad max(dist_H(p,d,i), dist_W(p,d,j)) < \delta \\ 0, \quad \text{otherwise} \end{cases}$$

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

3.3 System Architecture

Fig. 1 shows an overview of the system architecture, which consists of two main components. The first is the Constraint Satisfier, which takes as input the passengers’ and drivers’ requests and produces feasible passenger-driver pairs.The second is the Matching Module, that takes as input the bipartite graph of feasible pairs and finds a maximum weighted matching. The two components are loosely connected through the weighted bipartite graph, which is the output of the first and the input to the second. An important aspect of our system is that it is *online*: requests from drivers and passengers can arrive dynamically (at times $t_d^{(r)}, t_r^{(r)}$, respectively) and also can expire (when a driver arrives at the destination $t_d^{(n_d)}$, 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. The decomposition of the problem into two parts, is key for enabling a modular, fast, online, optimal system.

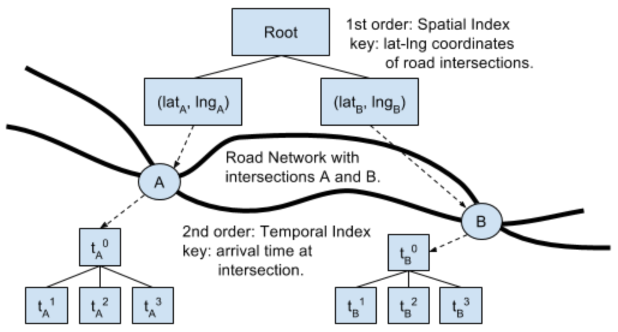
Root
1st order: Spatial Index
key: lat-lng coordinates of road intersections.
(lat_A, lng_A)
(lat_B, lng_B)
Road Network with intersections A and B.
A
B
2nd order: Temporal Index
key: arrival time at intersection.
t_A^0
t_A^1
t_A^2
t_A^3
t_B^0
t_B^1
t_B^2
t_B^3

Figure 3: Storing spatio-temporal data in our specilized (road network-based) `Constraint Satisfier`. The intersections of the road network are stored in a tree data structure for fast accessing. That is the 1^{st} order data structure that stores intersections using as keys their (lat, lng) coordinates. Each key is paired with a value, and the values (of the 1^{st} order data structure) are symbol tables that contain $< key, value >$ pairs, refereed to as the 2^{nd} order data structure. In the 2^{nd} order data structure the keys are the times when user appear at the specific intersection and as a value the id of the user.

4. CONSTRAINT SATISFIER

This component receives the queries from the drivers and the passengers, and generates the feasible (driver, passenger) pairs. The Constraint Satisfier needs to support fast insertion queries (*e.g.* when new requests arrive), and fast searches queries (when matching passengers and drivers) in real time.

4.1 SatisfierDB

We initially built the SatisfierDB, using off-the-shelf components. We chose MongoDB because speed and scalability are our primary concerns, and we do not require complex join queries that relational databases offer. Moreover, *MongoDB* [16] is very popular in the industry (*e.g.* used by Foursquare [17]). MongoDB is a document-oriented database that stores data in collections made out of individual documents; each document is big JSON file with no particular format or schema. Finally, MongoDB supports fast and accurate spatio-temporal proximity queries.

Spatial Indexing: MongoDB offers supports for queries that calculate geometries on an earth-like sphere, through the *2dsphere index* – a grid based geohashing scheme [18].

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 do a write lock, during which, 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).

4.2 Our Specialized Constraint Satisfier

Next, we build a specialized Constraint Satisfier, which is tailored to the needs of our problem and it takes into account the unique characteristics of our data. More specifically, our specialized system takes into account the underlying road network of the trajectories. To that end, our Constraint Satisfier utilizes and fine-tunes a state-of-the-art spatio-temporal data structure: the *road network* [19, 20], as depicted on Fig. 3.

Road Network: The Constraint Satisfier uses a road network to store the (lat, lng) coordinates of the users. The road network contains intersections, in the form of (lat, lng) coordinates. The (lat, lng) coordinates are stored in a tree structure (1^{st} order structure in Fig. 3) with the coordinates being the keys. The values that the keys (coordinates) are paired with are symbol tables that contain $< key, value >$ pairs (2^{nd} order structure in Fig. 3). In the 2^{nd} order data structure the keys are the times when user appear at the specific intersection and as a value the id of the user. When a new driver request arrives, the (lat, lng) points of her trajectory are mapped to their closest intersections; from there (the closest intersections) they are mapped to the 2^{nd} order structure (a $< key, value >$ symbol tables) where they are stored based on their times (of arrival at the intersections). When a passenger wants to find the driver that can pick her up, the Constraint Satisfier (1) will look at the intersections that are within δ of her home, and for each one of this intersections (2) it will get the drivers that are within time constraints; then, the Constraint Satisfier (3) will do the same for the work locations, and (4) the drivers that can both pick-up and drop-off the passenger will be returned. A similar procedure will be followed for when a driver is trying to find passengers; for each one of the points of her trajectory the Constraint Satisfier will find the passengers that are withing spatio-temporal constraints and can be both picked-up and dropped-off.

Implementation Details: We used a KDTree for the 1^{st} order structure, and MultiMaps[4] for the 2^{nd} order structure. A KDTree is a space-partitioning data structure for organizing points in a k-dimensional space; for our (lat, lng) coordinates k is 2. We use road intersections, instead of road segments due to practical reasons: our road network (which will be described in the Evaluation Section) is dense and the distances between them are short [21] [5]. Moreover, our (1^{st} order structure is a static road network and we don't have to update or delete nodes; all nodes are added when a priori. We use MultiMaps for the 2^{nd} order data structure, which we want to be dynamic and support fast add/remove operations.

5. MATCHING DRIVERS-PASSENGERS

In this section, we describe the Matching Module that takes as input the bipartite graph of feasible driver-passenger pairs and provides a matching. Recall that the edges are weighted, with a weight that corresponds to the length of the shared trip and is proportional to the revenue made by SORS. We formulate the problem as maximum weight matching (MWM), and we provide and algorithm (referred to as Online MWM) that is efficient, and online (*i.e.* it continuously updates the matching in the presence of arrival/expiration of requests). As a baseline for comparison, we compare it to the 0-1 Algorithm, described in [15] and the offline solution of MWM.

5.1 Online MWM Algorithm

Consider the bipartite graph of feasible pairs: $G = (D \cup P, E, W)$ where $E = \{(d, p) : d \in D, p \in P\}$ s.t. that the constraints of d, p, as defined in a previous section, are satisfied. Each edge $(d, p) \in E$ is associated with a weight $W(d, p)$ indicating the length of the shared trip between the driver and the passenger.

Our Online MWM algorithm breaks the graph of feasible pairs into multiple sub-graphs based on the weight of their edges (*e.g.*, the 1^{st} sub-graph contains the top 1% of the edges, the 2^{nd} sub-graph contains the top 2%, *etc.*,) and for each one of the sub-graphs applies Max Cardinality Matching (MCM), giving priority to higher edges.MCM can be solved efficiently and very fast; our bipartite graph that has tens of thousands of nodes and a few million edges can be solved in seconds. 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)

[4]A Hash Table where the same key can map to multiple user ids. To be more specific a key maps to a set of users.

[5]For a sparser road network large distances between intersections, using road segments and an R-Tree would be better.

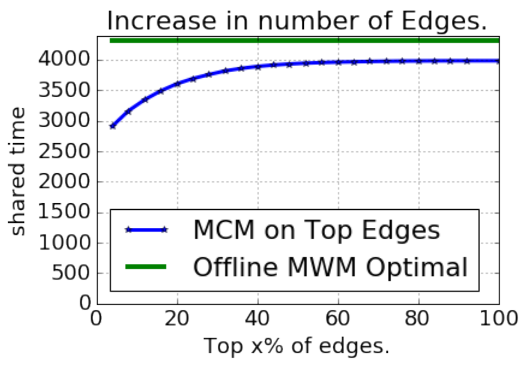

Figure 4: This figure shows how well our MCM-based algorithm approximate the MWM offline. Online MWM **breaks the graph into multiple sub-graphs based on the weight of their edges, e.g. sub-graph-1 contains the top 1% of the edges, while sub-graph-2 contains the top 2% of the edges, e.t.c and then it applies an augmenting path algorithm to find the MCM in each sub-graph. The combined solution of all sub-graphs is the approximation solution to the MWM problem.**

and optimally using augmenting paths [22]. 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 be handled by efficient incremental updates. 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 $O(|E|)$), instead of solving the problem from scratch. Finally, Fig. 4 shows how our Online MWM compares to the offline MWM solution (how well the algorithm approximates the offline optimal solution).

5.2 Greedy Algorithm

Many on-demand taxi services have emerged recently, which act as a broker between a taxi and a passenger, the primary example being Uber. These companies use proprietary matching algorithms, which are however widely believed to be simpler: *e.g.* typically match a passenger "greedily" with the closest driver. As a baseline for comparison, we define a Greedy Algorithm as follows: each request is matched with its best available choice, at the time of its arrival, e.g. a passenger is matched to the best unmatched driver at the time her request arrives in the system. We are interested in understanding how our global optimization (Online MWM) compares to these simpler, greedy heuristics, in terms of the global objective (*i.e.* total joint distance traveled).

6. EVALUATION

In this section we evaluate our ridesharing system using spatio-temporal data from the city of New York. The data are summarized in Tab. 2 and was obtained in our prior work [14]. More specifically, we evaluate the Online MWM in terms of the global objective (matching rations and sum of total shared trips, which translates to revenue). Then, we compare our Constraint Satisfier with the baseline SatisfierDB in turns of scalability.

6.1 Experimental Setting

Given the home/work locations of the users, their departure times and the drivers' routes, we compute the performance of the algorithm for different values of *ahead-of-time notification*. We do a discrete time simulation where all the events appear in a simulated time-line based on the order of their arrival. In our simulation, there are two types of events :(1) *request arrival*, and (2) *request expiration*.We show the speed of our system, as well as the matching ratio and the sum of all shared trips (total shared trips). Also, we apply the following spatio-temporal constraints: (i) a spatial constraint of 1 km $\delta = 1$ km and (ii) a delay tolerance of 10 minutes $\Delta t_1 = 10$. Finally, our Constraint Satisfier uses the NY road network from [21]; the road networked is represented by a weighted directed graph. The graph contains 264K nodes and 734K edges, and weight of each edge represents the distance between the nodes.

City	NY
Users	61K
Inter-point distance	100m
Average distance	16.1 km
Median distance	8.0 km
Average gps points per trajectory	78.8

Table 2: Summary of Twitter-NY Dataset.

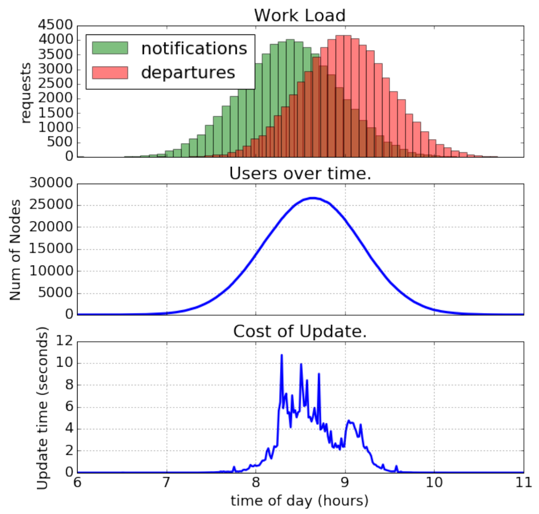

Figure 5: Performance under expected work load. We assume that the requests will arrive randomly before the departure. The probabilistic distribution to generate how long before departure users will notify the system is a uniform distribution in range [15 minutes, 60 minutes]. According to the figure, the highest stretch for our system occurs when the number of users (or nodes in the bipartite graph) is at its peak. At that time each update – that contains multiple new request – can required up to 12 seconds.

6.2 Results

	Shared Trip Sum	Comparison to Offline MWM (%)
Offline MWM	4320	–
Online MWM	3957	-8.4
0-1 Algorithm	3460	-20
Greedy Algorithm	2623	-39

Table 3: Offline Result Comparison for a graph with size 61K nodes (out of which only 48054 had a neighbor), and 1.5M edges. Online MWM **is 14.4% better than** 0-1 Algorithm **and 51% better than** Greedy Algorithm**.**

In Fig. 5 you can see the end-to-end experiment. According to the figure, the highest stretch for our system occurs when the number of users (or nodes in the bipartite graph) is at its pick. At that time each update – that contains multiple new request – can required up to 12 seconds. When compared to the offline optimal, which takes more

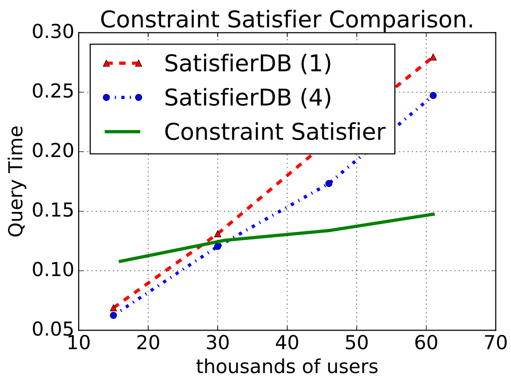

Figure 6: Comparing the scalability of Constraint Satisfier and SatisfierDB. Both Constraint Satisfier and SatisfierDB scale linearly, but the slope of the SatisfierDB is 4.65 times greater than the slope of Constraint Satisfier. Therefore, we say that the specialized Constraint Satisfier is 4.65 times more scalable when compared to the SatisfierDB. When there are a few users SatisfierDB tends to be faster; this happens because Constraint Satisfier is using a road network, with hundred of thousands of intersection, and when the number of users is low all this intersections are a burden. However, as the number of users grows the query time of Constraint Satisfier grows at a much slower rate than SatisfierDB; the road network of Constraint Satisfier pays of as the number of users keeps growing.

than two days to compute, Online MWM is 8.4% worse, but it's 14.4% better than 0-1 Algorithm (see Tab. 3).

Fig. 6 show the query speed of three different implementations of the constraint satisfier: (1) SatisfierDB with one process, (2) SatisfierDB with four parallel process, and (3) the specialized Constraint Satisfier. We see that parallelization can improve the speed of SatisfierDB, but the Constraint Satisfier is still faster and much more scalable. Both Constraint Satisfier and SatisfierDB scale linearly, but the slope of the SatisfierDB in (fig:query-speed) is 4.65 times greater than the slope of Constraint Satisfier. Therefore, we can say that the specialized Constraint Satisfier is 4.65 times more scalable when compared to the SatisfierDB that is build using off the shelf components.

7. CONCLUSION

An anticipated breakthrough in ridesharing is the ability to satisfy on-demand requests that do not require participants to schedule their trips in advance [5]. This will provide a participant the reassurance that they would still be serviced if their travel-needs change unexpectedly. In this paper, we design and evaluate an *scalable on-line ridesharing system (*SORS*)* that handles dynamic ridesharing requests, possibly with a short notice.

We break SORS into two main components: the constraint satisfier and the matching module. The constraint satisfier, a.k.a Constraint Satisfier, takes as input the itineraries and spatio-temporal constraints of drivers and passengers and provides feasible (driver, passenger) pairs. We achieve scalability by designing a constraint satisfier using a road networks data structure, specifically optimized for our spatio-temporal queries. Our Constraint Satisfier system is more scalable than what can be built using state-of-art off the shelf components: query time increases 4.65x slower with the number of users when compared to MongoDB.

We use the feasible pairs found by Constraint Satisfier to define a bipartite graph between possible drivers and passengers, with edge weights representing the length of the shared trip of a pair. The matching module takes as input the weighted bipartite graph and returns the maximum weighted matching (MWM), which captures the objective of real-world ridesharing systems (such as `Lyft Carpool`). We propose an efficient algorithm to solve the MWM problem, which is 51% better than greedy heuristics used by many real systems. Furthermore, the system is designed to handle efficiently requests that arrive on-line, via efficient queries of feasible pairs and incremental updates of the matching solution. We evaluate the entire SORS system using real mobile datasets to extract driver trajectories and passenger locations in urban environments. We show that SORS can provide a ridesharing recommendation to individual users with a sub-second query response time, even at high workloads.

8. ACKNOWLEDGMENTS

We would like to thank Kyriakos Mouratidis, an expert in spatio-temporal databases, for his feedback on our constraint satisifer.

9. REFERENCES

[1] B. McKenzie and M. Rapino, "Commuting in the united states: 2009." American Community Survey Reports, 2009.
[2] "More people in fewer cars." `https://newsroom.uber.com/us-washington/more-people-in-fewer-cars/`, 2016.
[3] "Meet lyft carpool: A new way to commute." `http://blog.lyft.com/posts/meet-lyft-carpool`, 2016.
[4] "Announcing uberpool." `https://newsroom.uber.com/announcing-uberpool/`, 2014.
[5] M. Furuhata, M. Dessouky, F. Ordóñez, M.-E. Brunet, X. Wang, and S. Koenig, "Ridesharing: The state-of-the-art and future directions," *Transportation Research Part B: Methodological*, vol. 57, no. 0, pp. 28 – 46, 2013.
[6] H. S., "Implementing Real-Time Ridesharing in the San Francisco Bay Area. ," Master's thesis, Mineta Transportation Institute, San Jose State University, CA, USA, 2010.
[7] "Scoop." `https://www.takescoop.com/`, 2015.
[8] S. Ma, Y. Zheng, and O. Wolfson, "T-Share: A Large-Scale Dynamic Taxi Ridesharing Service.," in *Proc. of ICDE*, 2013.
[9] J. F. Cordeau and G. Laporte, "The Dial-a-Ride Problem (DARP): Variants, modeling issues and algorithms," *4OR*, vol. 1, 2003.
[10] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang, "Optimization for dynamic ride-sharing: A review," *European Journal of Operational Research*, vol. 223, no. 2, pp. 295 – 303, 2012.
[11] R. Teal, "Carpooling: Who, how and why," *Transportation Research*, vol. 21A, no. 3, pp. 203–214, 1987.
[12] K. D., "Carpooling: Status and potential." Final Report U.S. Department of Transportation, DOT-TSC-OST-75-23, 1975.
[13] A. M. Amey, "Real-Time Ridesharing: Exploring the Opportunities and Challenges of Designing a Technology-based Rideshare Trial for the MIT Community," Master's thesis, MIT, 2010.
[14] 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 *Proc. of UbiComp (Best Paper Nominee Award)*, 2014.
[15] B. Cici, A. Markopoulou, and N. Laoutaris, "Designing an On-Line Ride-Sharing System," in *Proc. of SIGSPATIAL (short paper)*, 2015.
[16] "Mongodb." `http://www.mongodb.org/`.
[17] "Scaling mongodb at foursquare." `http://www.10gen.com/presentations/mongonyc-2012-scaling-mongodb-foursquare`.
[18] "Geospatial indexes in mongodb." `http://docs.mongodb.org/manual/core/geospatial-indexes/`.
[19] E. Frentzos, "Indexing objects moving on fixed networks," in *Proc. of SSTD*, 2003.
[20] K. Mouratidis and M. L. Yiu, "Anonymous query processing in road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, 2010.
[21] "9th dimacs implementation challenge - shortest paths." `http://www.dis.uniroma1.it/challenge9/download.shtml`, 2015.
[22] C. H. Papadimitriou and K. Steiglitz, "Algorithms for matching," in *Combinatorial Optimization, Algorithms and Complexity*, ch. 10, pp. 221–226, 1998.