Fast and Scalable Ridesharing Search

James Jie Pan and Guoliang Li

***Abstract*—In the next few decades, it is estimated that a quarter of all trips worldwide will be served by shared mobility supported in part by lower carbon footprint compared to private mobility. In particular, on-demand ridesharing is appealing due to its convenience, matching passengers needing rides to vehicles in real time while optimizing the matching. While this matching problem is computationally challenging, the state-of-art greedy search algorithm assigns passengers one at a time to the locally best vehicle and has been shown to perform well in practice. However, in order to scale the algorithm, how to parallelize searches for multiple requests remains challenging due to contention for vehicle tours. Moreover, the request latency may still be too high for on-demand requests. In this paper, we give several techniques to speed up and scale out ridesharing search. To deal with data contention while scaling out greedy search, we introduce a "map-release" and ticketing system that sacrifices read-write consistency to achieve high concurrency, even under high contention, and while avoiding expensive aborts incurred by optimistic approaches. To address high request latency, we give a caching technique to speed up the tour expansion subroutine of greedy search, and we also give a pruning technique to reduce the tour candidates even further compared to existing techniques. Together, these techniques deliver around 7x the throughput and order of magnitude lower latency on a real instance compared to the "embarassingly parallel" parallelized map approach and with better scalability.**

***Index Terms*—Ridesharing, concurrency control, tour expansion.**

I. Introduction

Shared mobility, such as buses, subways, and shared vehicles, represents around 2% of the global mobility market and is estimated to reach 24% worldwide by 2040 and 36% in China.[1] This trend may be attributed to better environmental friendliness compared to private mobility [1]. In particular, *on-demand ridesharing*, which works by matching passengers needing rides to participating vehicles located throughout a service region, has appeal due to its convenience and low fares. The service can act as a marketplace, where riders post requests to the platform and drivers choose from a list which ones to serve [2], but in most cases it centrally assigns passengers to vehicles. Doing so can lead to better matchings but suffers from combinatorial explosion of not only assignments but also *vehicle tours* as the platform must additionally decide the best tour for serving the passengers once assignments are decided [3], as illustrated below.

Example 1 (Tour Combinations): In Fig. 1, Vehicle 1 is following tour AB, and Vehicle 2 is following tour CDE. The request departing from X and arriving at Y is unassigned. For Vehicle 1, there are 6 ways that it can insert the new request into its tour, keeping departures ($+$) in front of arrivals ($-$) and preserving the order of existing locations: $XYAB$, $XAYB$, $XABY$, $AXYB$, $AXBY$, and $ABXY$, and for Vehicle 2, there are 10 ways: $XYCDE, CXYDE, CDXYE, CDEXY$, $XCYDE$, $CXDYE$, $CDXEY$, $XCDYE$, $CXDEY$, and $XCDEY$. If order is not preserved, e.g. C can be visited after D or E, then there are a total of 30 ways to expand Vehicle 2's tour. Without preserving order, the number of possible tours grows in $(2n)!/2^n$ or $O(n!)$, where n is the number of requests in the expanded tour. With preserving order, the number grows in $n(n+1)/2$ or $O(n^2)$, where n is the number of locations.

Real-world services of this type, such as UberX Share[2] and Lyft Shared,[3] already struggle with the computational burden of this problem, leading to compromises such as artificially limiting vehicle capacities and incentivizing issuing requests far in advance instead of on-demand. In the literature, the state-of-art *greedy search* algorithm assigns passengers on a first-come first-serve basis by finding the best vehicle per passenger, as shown in Algorithm 1 [4]. The *tour expansion* operator on line 3, ExpandTour, determines the cost of a potential assignment on line 4. As finding optimal tours is famously NP-hard [5], a key technique is to approximate it using an *insertion* heuristic, where the departure and arrival locations of a request are inserted into an existing tour without changing the ordering of other locations in the tour [6]. The naive algorithm requires $O(n^2)$ time, where n is the length of the tour, but a dynamic programming (DP) approach improves this to $O(n)$ [4].

Limitations: Nevertheless, (1) how to parallelize Greedy-Search *across requests* remains challenging due to contention for vehicle tours by the parallel request processing workers, as illustrated by the following examples.

Example 2 (Read-Write Conflict): Workers W_1 and W_2 are processing requests $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$, respectively, in parallel. At a certain time, W_1 reads tour $\boldsymbol{s_1}$ and evaluates ExpandTour$(\boldsymbol{s_1}, \boldsymbol{r})$ to obtain a cost for $\boldsymbol{s_1}$. Immediately after, W_2 updates the tour. The cost obtained by W_1 is now incorrect because it does not account for the update.

Manuscript received 31 October 2023; revised 28 April 2024; accepted 17 June 2024. Date of publication 24 June 2024; date of current version 27 September 2024. This work was supported in part by the National Key R&D Program of China under Grant 2023YFB4503600, in part by the NSF of China under Grant 61925205, Grant 62232009, and Grant 62102215, in part by Zhongguancun Lab, in part by Huawei, in part by TAL Education, and in part by Beijing National Research Center for Information Science and Technology (BNRist). Recommended for acceptance by S. Salihoglu. *(Corresponding author: Guoliang Li.)*

The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100190, China (e-mail: liguoliang@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TKDE.2024.3418433

[1]DiDi Form F-1 SEC filing, 2021

[2]http://www.uber.com/us/en/ride/uberx-share/

[3]http://www.lyft.com/blog/posts/shared-rides-return-to-more-cities

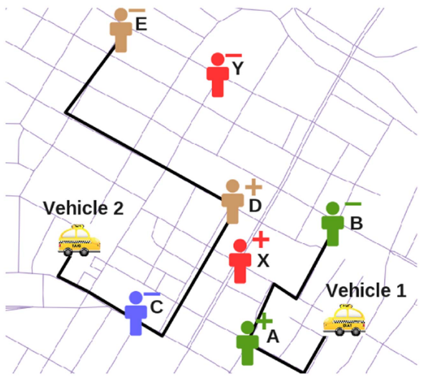


Fig. 1. Example ridesharing scenario with two vehicles visiting locations A–E and a new request aiming to depart from X and arrive at Y.

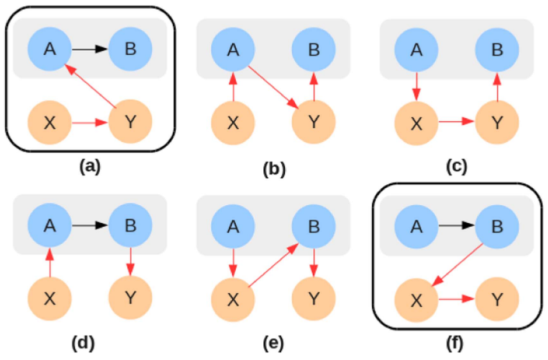


Fig. 2. All possible insertion-based tour expansions for an initial tour AB of length 2 and request departing from X and arriving at Y. Here, A and X are departure locations and they respectively must precede B and Y. In the worst case, DP tour expansion issues 15 SPSP queries, indicated by the red arrows. In (a) and (f), the blue and orange requests are "chained" one after another, with requests being served in succession.

Example 3 (Write-Write Conflict): Continuing the example above, if W_1 proceeds to update $\boldsymbol{s}_1$ without reading it again following the write by W_2, then the write by W_2 will be lost.

Moreover, (2) many objectives rely on calculating shortest-paths (SPs) through a graph model of the road network as a way to estimate the cost of an assignment [7]. Irregardless of whether the paths represent least distance, least estimated travel time, revenue, or other metric, SP queries are still expensive, even when an index is used. The DP algorithm issues $O(n)$ number of single-pair shortest-path (SPSP) queries, which may be too expensive for large tours such as those belonging to high-capacity vehicles or for long "chains" on small capacity vehicles, as shown in Fig. 2. Chains can be arbitrarily long, irregardless of vehicle capacity. Finally, (3) a common technique for the Prune routine on line 2 of Algorithm 1 is to use a request arrival deadline to prune vehicles by distance [8], [9]. Naturally it is possible to prune vehicles and tours by considering the arrival deadlines of *all* existing requests in a tour, but it is unclear how to do this without first performing expensive tour expansion. For example in [4], tour expansion is necessary to determine the new arrival times which are then checked against the deadlines.

Algorithm 1: GreedySearch$(S, \boldsymbol{r})$.

```
   Input: vehicle tours $S$, request $\boldsymbol{r}$
1: $\boldsymbol{s}^* \leftarrow$ null, $c^* \leftarrow \infty$
2: for $\boldsymbol{s} \in$ Prune$(S, \boldsymbol{r})$ do            $\triangleright$ Read phase
3:     $\boldsymbol{s}' \leftarrow$ ExpandTour$(\boldsymbol{s}, \boldsymbol{r})$
4:     $c \leftarrow$ Cost$(\boldsymbol{s}')$
5:     if $c < c^*$ then
6:         $\boldsymbol{s}^* \leftarrow \boldsymbol{s}'$, $c^* \leftarrow c$
7: Update$(S, \boldsymbol{s}^*)$                          $\triangleright$ Write phase
```

Our Contributions: In this paper, we give several techniques to address these limitations. (1) We start by giving concurrency control techniques to deal with contention during parallel request processing. To the best of our knowledge, while one work [10] parallelizes the inner loop of GreedySearch using a map-reduce framework, parallel request processing for centralized ridesharing platforms has not been previously studied. Speculative approaches such as MVCC [11] abort requests when inconsistencies are encountered, but aborts may not be acceptable for on-demand requests that demand low latency. Instead, we give a variation of two-phase locking (2PL) called "map-release" (MR) that more flexibly acquires and releases locks to improve concurrency. The price of MR is the loss of read-write consistency because locks are not held until the end of processing. But as tour expansion is approximated anyway and read-write errors are small, quality is not greatly impacted. We then give a "ticket-locking" (TL) mechanism on top of MR that can flexibly avoid deadlocks, further increasing concurrency. (2) To reduce the number of SP queries, we give an "insertion cache" technique that requires only 2 or $2 + n$ single-source queries if the road network graph is directed. (3) To improve the pruning power, we give a "slack bubble" technique that considers the arrival deadlines on all requests in a tour without needing to perform tour expansion, taking advantage of fast lower bounding using the euclidean metric. Experimental results show that these techniques together can process hundreds of requests per second on a commodity server even without an SP index, representing about 7x improvement in throughput over the parallelized map approach.

The rest of this paper is organized as follows. In Section II, preliminary definitions are given. In Section III, related work is presented. Then in Section IV, the locking schemes are given, and in Section V, the caching and pruning techniques are given. Experimental results are presented in Section VI, followed by concluding remarks in Section VII.

II. Problem Statement

Table I lists the notation used throughout this paper.

Definition 1 (Ridesharing Request): A ridesharing request consists of a departure location u, called the *origin*, and an arrival location v, called the *destination*. This is represented by an ordered pair, (u, v).

Definition 2 (Vehicle Tour): A vehicle tour consists of a sequence of locations, each belonging to a unique request.

TABLE I
NOTATION USED IN THIS ARTICLE

Symbol	Description
W_i	The ith worker thread.
$\boldsymbol{s} = s_1 s_2 ... s_n$	A vehicle tour of length n.
$\boldsymbol{r} = (u, v)$	A request with origin u and destination v.
$D(\boldsymbol{s})$	The cost of tour $\boldsymbol{s}$.
$\lvert\boldsymbol{s}\rvert$	The length of tour $\boldsymbol{s}$.
$d(u, v)$	The cost (*e.g.* shortest-path distance) between segment (u, v) of a tour.
$\lvert\lvert u - v\rvert\rvert_2$	The Euclidean distance between u and v.
$\delta(u, v)$ or $/uv/$	The travel duration between u and v.
$T(i)$	The arrival time at location s_i of tour $\boldsymbol{s}$.
$\xi(i)$	The slack time at location s_i of tour $\boldsymbol{s}$.
t_s	The earliest acceptable departure time for a request.
t_e	The latest acceptable arrival time for a request.
ν	The vehicle speed.
κ	The maximum seating capacity of a vehicle.
τ	The delay tolerance.
R	The set of all requests.
S	The set of all vehicles.
E_i	The slack bubble on the tour segment (s_{i-1}, s_i).

In the real world, passengers cannot be dropped off before being picked up, imposing an ordering constraint on tours. If origin u appears in a tour, then the corresponding destination v must appear after it. A tour does not need to start and finish at the same location.

Many optimization objects, including distance or duration [8], [12], revenue [13], or utility [14], are calculated based on segments along the tours. We call this function d, so that the cost of a tour, $\boldsymbol{s} = s_1 s_2 \ldots s_n$, becomes $D(\boldsymbol{s}) = \sum_{i=1}^{n-1} d(s_i, s_{i+1})$. In these studies, function d fundamentally performs an SPSP query. The road network is modeled as a directed graph using existing map-matching techniques such as [15], and the edges are usually set as travel duration or distance along the matched roads. The edges can also be dynamic to represent traffic, and existing techniques can be used to answer SP queries on dynamic road networks [16]. All request and tour locations belong to the vertices in the graph.

Service providers aim to optimize ridesharing assignments across a planning period, for example one business day. This can be modeled as an *online matching problem:* Given a set of ridesharing requests, R, that are revealed online, and a set of vehicles with tours, S, that may be initially empty, assign requests to vehicles such that the cost across the tours is a minimum once all the requests have been revealed, subject to constraints. When request (u, v) is assigned to a vehicle, the vehicle tour is expanded to contain u and v.

The GreedySearch algorithm solves the online matching problem by greedily assigning each request one at a time. This is formalized as the following *local search problem:*

Definition 3 (Ridesharing Search Problem): Given a set of vehicles with tours, S, and a ridesharing request $\boldsymbol{r}$, assign $\boldsymbol{r}$ to the vehicle with minimum change in tour cost, subject to constraints.

Typically, there are two constraints on the tours. The capacity constraint represents the physical seating limitations of ridesharing vehicles, and the deadline constraint represents limits on travel duration.

Definition 4 (Capacity Constraint): Associate each tour with a capacity, κ, and each request with a load. For each location in a tour, add the load of the corresponding request to a running sum if it is an origin, or subtract the load if it is a destination. The sum must never exceed κ.

Definition 5 (Deadline Constraint): Associate each tour with a timestamp, t_0, indicating the start of vehicle service, and associate each request with a pair of deadlines, t_s and t_e. For each location s_i in tour $\boldsymbol{s}$, label it with $T(i) = t_0$ if $i = 1$ or $T(i) = T(i-1) + \delta(s_{i-1}, s_i)$ otherwise, where $\delta(u, v)$ returns the travel duration between u and v. For any i, and where t_s and t_e are the deadlines of the request corresponding to s_i: if s_i is an origin, then $T(i) \geq t_s$ must be true, otherwise if s_i is a destination, then $T(i) \leq t_e$ must be true.

The T value represents the time that a vehicle visits a particular location. The "start time", t_s, is the earliest possible time that a passenger can depart, and the "end time", t_e, is the latest acceptable time that a passenger can arrive at the destination. Together, they form the delivery *time window* [8]. The duration $\delta(u, v)$ can be found using travel time estimation techniques such as [17]. It is impossible to guarantee that real vehicles will visit tour locations exactly at the estimated times. Even so, the deadline constraint is a useful model for limiting passenger travel times which can lead to greater customer satisfaction.

To explain the later techniques, we introduce the concept of "insertion pairs". Insertion pairs denote where a new request should be inserted into a tour.

Definition 6 (Insertion Pair): Given tour $\boldsymbol{s} = s_1 s_2 \ldots s_n$, an insertion pair for this tour is any pair (i, j) where $1 \leq i \leq n+1$ and $i \leq j \leq n+1$.

Example 4 (Insertion Pairs): Let $\boldsymbol{s} = s_1 s_2 s_3 s_4$. Given request $\boldsymbol{r} = (u, v)$ and insertion pair (2,4), the expanded tour is $s_1 u s_2 s_3 v s_4$.

III. Related Work

A. Ridesharing Algorithms

Ridesharing services can take many forms. In the Dial-A-Ride Problem (DARP), rides are requested in advance instead of on-demand [18]. In the *ridehailing* problem, each vehicle serves one passenger at a time like a taxi, modeled as a bipartite mathing problem [19]. For centralized ridesharing as studied in this paper, existing works can be classified as search-based or join-based algorithms [20]. But as join-based algorithms are generally slower than search-based algorithms, in this paper we focus on search and its specific subroutines, namely tour expansion and tour pruning. For completeness, we also mention related work on vehicle pruning.

Search Algorithms: By far, the most popular search-based algorithm is GreedySearch which assigns each request to its locally optimal vehicle [4], [7], [8], [21], [22], [23]. Several variants have also been studied, including [22], [23], [24], [25] which searches for Pareto optimal solutions for a dual-objective variant, [7], [14], [26] which aim to increase the quality of the search via local minima escape mechanisms or via demand-aware mechanisms, and [10], [27] which aim to increase throughput by parallelizing the search. In this paper, we focus on parallelizing the core GreedySearch algorithm *across requests*, not just parallelizing the search itself. Note that there is no guarantee GreedySearch provides an optimally competitive solution, but it is believed that there is no algorithm for centralized ridesharing that has constant competitive ratio [4].

Tour Expansion: In [12], a tree-based index is used to speed up exact tour expansion, later used in [24], [25]. But most works use insertion-based tour expansion, which appears in [6] for DARP and used in many later works. The $O(n)$ DP algorithm appears in [4] and is the basis for [7], [22], [23] and used in [28]. A speed-up technique involving precomputing parts of the tour cost appears in [22], [23], [26].

Physical tour construction is usually performed by concatenating SPSPs across each segment of the tour. To avoid repeating individual SP queries, several works make use of *SP caches*. For example in [10], [27], an all-pairs SP cache is constructed so that subsequent queries can be answered via $O(1)$ lookup. But this approach is memory intensive, and other works use a fixed-size least-recently used (LRU) cache [4], [7], [12], [13], [26]. In this paper, we propose an *insertion cache* that can be quickly primed and relies on a more fine-tuned eviction mechanism compared to LRU.

Tour Pruning: Infeasible tours can be pruned during tour expansion. A naive “admittance” strategy is used in [8], [21], where each candidate tour is checked for feasibility only after it is already constructed. The “early termination” strategy used in [4], [7], [10], [12], [14], [22], [23], [24], [25], [26], [27] works by terminating tour expansion once a certain stopping condition is met, thereby pruning all subsequent expansion candidates. For example, one particular condition states that if $T(i) + \delta(s_i, u)$ for request (u, v) exceeds the request deadline, then i cannot be an insertion position. This prunes all the insertion pairs $(i, i), (i, i + 1), \ldots, (i, n)$. Even so, early termination still requires at least partially evaluating a number of insertion pairs. In this paper, we propose a slack bubbles technique that aims to yield all feasible insertion pairs upfront.

Vehicle Pruning: Proximity-based pruning prunes all vehicles outside a particular distance to the request origin, typically based on the request deadline. In [8], [9], [21], [24], vehicle and road network indexes are introduced to quickly prune vehicles based on this principle. Another approach is cost-based pruning, where the estimated tour cost of a vehicle is used as the pruning condition. In [4], vehicles are processed in order of lower-bound cost so that once the lower-bound cost of a vehicle exceeds the true cost of an incumbent, then all vehicles lower in the order can be safely pruned. This technique is also used in [7], [26].

B. Concurrency Control

In [10], a map-reduce framework is used to parallelize the vehicle search loop of GreedySearch (lines 2–6 of Algorithm 1). But this framework cannot be used to parallelize the *requests* as it lacks any means of concurrency control. Instead, ridesharing assignment can be viewed analogously to a *transaction* as it results in an update to the state of the system, and various techniques are available from the database community for transaction concurrency control including partitioning [29], optimistic or pessimistic methods [30], [31], [32], [33], and methods based on multi-versioning [11], [33]. Partitioning assigns worker threads to distinct non-overlapping partitions, but this approach cannot be adapted to the ridesharing problem because the vehicle tours are hard to effectively partition. Pessimistic techniques such as 2PL use locks to prevent conflicts from arising in the first place but suffer from reduced concurrency. Optimistic techniques speculatively execute transactions in a lock-free manner, relying on a pre-commit verification phase to detect data conflicts and using aborts to avoid data corruption. But for ridesharing, each search is expensive, making even a single abort hard to tolerate. Multi-version schemes such as MVCC [11] store multiple versions of the same data record, allowing it to loosen the conditions that cause abort while still retaining strong isolation guarantees. But in addition to storage and maintenance overhead, they may still exhibit frequent aborts under high contention. For these reasons, we propose a pessimistic technique but with more flexibility compared to 2PL.

IV. Map-Release and Ticket Locking

The search loop of Algorithm 1 starting on line 2 is embarrassingly parallel on condition that the reduction is performed atomically. This can be achieved under a local lock. But parallelizing multiple instances of the search algorithm is not as trivial as there can be contention for vehicle tours.

For ridesharing, write-write conflicts cannot be tolerated. While a read-write conflict merely leads to a potentially suboptimal assignment, a write-write conflict erases a previous assignment, causing an assigned customer to be suddenly unassigned. For read-write conflicts, tour expansions are approximated anyway and the error in the tour cost due to the conflict is usually small. Hence, it may be worth it to sacrifice read-write consistency for the sake of concurrency, if by doing so the concurrency can be increased.

The “map-release” (MR) and “ticket locking” (TL) schemes are both designed towards this aim, differing in the way that deadlocks are managed. In MR, locks are acquired in a global sort order, just as in many modern in-memory transactional databases [33], [34], [35], [36]. This is a well-known deadlock-avoidance technique, but it can lead to long locking periods. In TL, worker threads acquire locks on a first-come first-serve basis, and there is no global sort requirement. Free from the need for read-write consistency, this simple technique is sufficient to avoid deadlocks while reducing locking periods due to more flexible lock acquisition. The basis for these techniques is the “exclusive” locking scheme based on 2PL, which is able to provide both read-write and write-write consistency. Fig. 3 illustrates the workflows for these three schemes.

A. Exclusive Locking

Exclusive locking shown in Fig. 3(a) is characterized by its use of *exclusive locks* that block all access to the locked data record, except for the owner of the lock. To maintain read-write consistency during parallel vehicle search, all the locks must be exclusive because the tour to update is not known at the time of the request. Before the read phase, a lock for each of the tours

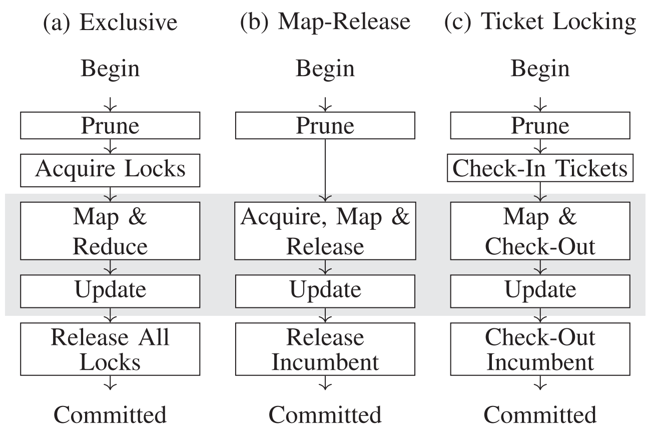


Fig. 3. Workflows for different locking schemes.

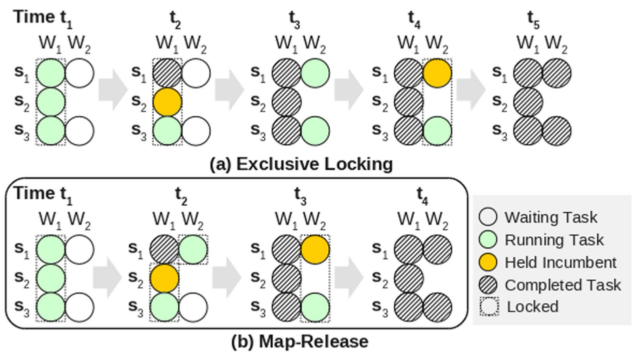


Fig. 4. Task evolution under exclusive locking (a) and map-release (b) on an instance with two request processing workers, W_1 and W_2, and three tours, $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, and $\boldsymbol{s_3}$. Each circle represents an ExpandTour task.

in the candidate set is acquired. Then following the write phase, all the locks are released at once. To prevent deadlocks, locks are acquired in a global sort order.

Exclusive locking has marginal overhead but it can cause long locking periods when there is high contention.

B. *Map-Release*

Map-release (MR) allows a larger number of tours to be simultaneously accessed by acquiring and releasing locks *along the way* instead of all at once. The workflow is shown in Fig. 3(b). Note that absence of a pre-acquisition phase compared to exclusive locking. As soon as a tour is determined to not be the reduction incumbent, the lock on the tour is immediately released, allowing other workers to access the tour. This is highlighted by the following example.

Example 5 (Map-Release): In Fig. 4 , each request processing worker can perform an unlimited number of parallel ExpandTour tasks. At time t_1, worker W_1 obtains locks on all three tours, $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, and $\boldsymbol{s_3}$, and at time t_2, worker W_1 decides that $\boldsymbol{s_2}$ is the reduction incumbent for the request it is processing. Under exclusive locking (Fig. 4(a)), W_1 holds the lock on $\boldsymbol{s_1}$ until the end of request processing, even though this tour is no longer needed, but under MR (Fig. 4(b)), it can release this lock, allowing W_2 to acquire a lock on $\boldsymbol{s_1}$ and begin working on it immediately.

Algorithm 2: GreedySearch$(S, \boldsymbol{r})$ With Map-Release.

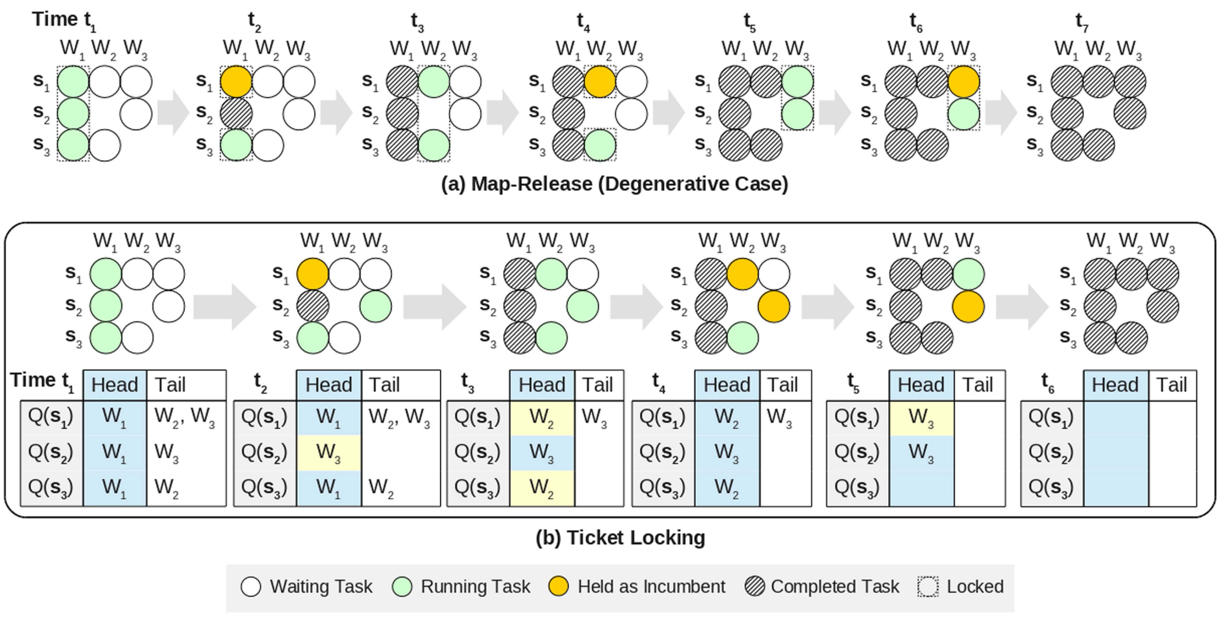
```
      Input: vehicle tours $S$, request $\boldsymbol{r}$
 1: $\boldsymbol{s}^* \leftarrow$ null, $c^* \leftarrow \infty$
 2: for $\boldsymbol{s} \in$ Prune$(S, \boldsymbol{r})$ do
 3:     Acquire lock on $\boldsymbol{s}$                              ▷
 4:     $\boldsymbol{s}' \leftarrow$ ExpandTour$(\boldsymbol{s}, \boldsymbol{r})$
 5:     $c \leftarrow$ Cost$(\boldsymbol{s}')$
 6:     if $c < c^*$ then
 7:         Release lock on $\boldsymbol{s}^*$                       ▷
 8:         $\boldsymbol{s}^* \leftarrow \boldsymbol{s}$, $c^* \leftarrow c$
 9:     else                                                          ▷
10:         Release lock on $\boldsymbol{s}$                         ▷
11: Update$(S, \boldsymbol{s}^*)$
12: if $\boldsymbol{s}^* \neq$ null then                              ▷
13:     Release lock on $\boldsymbol{s}^*$                           ▷
```

Algorithm 3: GreedySearch$(S, \boldsymbol{r})$ With Ticket Locking.

```
      Input: vehicle tours $S$, request $\boldsymbol{r}$, ticket queues $Q$
 1: $\boldsymbol{s}^* \leftarrow$ null, $c^* \leftarrow \infty$
 2: $P \leftarrow$ processer identifier                              ▷
 3: $S' \leftarrow$ Prune$(S, \boldsymbol{r})$
 4: Push a ticket into $Q[\boldsymbol{s}]$ for each $\boldsymbol{s} \in S'$    ▷
 5: for $\boldsymbol{s} \in S'$ do
 6:     while Ticket neither first nor notified do                   ▷
 7:         Wait until notified                                       ▷
 8:     $\boldsymbol{s}' \leftarrow$ ExpandTour$(\boldsymbol{s}, \boldsymbol{r})$
 9:     $c \leftarrow$ Cost$(\boldsymbol{s}')$
10:     if $c < c^*$ then
11:         Pop ticket for $P$ from $Q[\boldsymbol{s}^*]$ and notify next   ▷
12:         $\boldsymbol{s}^* \leftarrow \boldsymbol{s}$, $c^* \leftarrow c$
13:     else                                                          ▷
14:         Pop ticket for $P$ from $Q[\boldsymbol{s}]$ and notify next      ▷
15: Update$(S, \boldsymbol{s}^*)$
16: if $\boldsymbol{s}^* \neq$ null then                              ▷
17:     Pop ticket for $P$ from $Q[\boldsymbol{s}^*]$ and notify next       ▷
```

Algorithm 2 shows how MR is applied to request processing. The new lines compared to Algorithm 1 are indicated by the triangle symbol (▷) on the right-hand side, showing the release mechanism on lines 7, 10, and 13. For parallelizing ExpandTour tasks inside the search loop on line 2, the task threads need to be coordinated so that locks are acquired in the global sequence. This can be done by moving lock acquisition out of the parallel portion.

Under MR, read-write consistency is not guaranteed. If worker W_1 working on $\boldsymbol{r}$ decides that tour $\boldsymbol{s}$ is not the reduction incumbent, the lock on $\boldsymbol{s}$ is released and is never reclaimed. If $\boldsymbol{s}$ is subsequently updated by another worker while $\boldsymbol{r}$ is still being processed by W_1, worker W_1 will not reevaluate $\boldsymbol{s}$ as a candidate for $\boldsymbol{r}$, even if $\boldsymbol{s}$ ends up being the best tour for $\boldsymbol{r}$ as a result of the update.



Time t_1	Head	Tail
$Q(s_1)$	W_1	W_2, W_3
$Q(s_2)$	W_1	W_3
$Q(s_3)$	W_1	W_2

t_2	Head	Tail
$Q(s_1)$	W_1	W_2, W_3
$Q(s_2)$	W_3	
$Q(s_3)$	W_1	W_2

t_3	Head	Tail
$Q(s_1)$	W_2	W_3
$Q(s_2)$	W_3	
$Q(s_3)$	W_2	

t_4	Head	Tail
$Q(s_1)$	W_2	W_3
$Q(s_2)$	W_3	
$Q(s_3)$	W_2	

t_5	Head	Tail
$Q(s_1)$	W_3	
$Q(s_2)$	W_3	
$Q(s_3)$		

t_6	Head	Tail
$Q(s_1)$		
$Q(s_2)$		
$Q(s_3)$		

Fig. 5. Task evolution under map-release (a) and ticket locking (b) on an instance with three request processing workers, W_1, W_2, and W_3, and three tours, $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, and $\boldsymbol{s_3}$.

C. Ticket Locking

When the reduction incumbent for all requests is always the first evaluated tour, map-release degenerates into exclusive locking because of its reliance on the global lock order, as shown in Fig. 5(a). To overcome this issue, ticket locking (TL) arranges tour expansion tasks in a more fine-grained way in order to improve concurrency while still providing write-write consistency. To achieve this, the global lock order is replaced with a "ticket order" that is *based on the needs* of the individual requests. The workflow is shown in Fig. 3(c).

As shown in Algorithm 3, prior to calling GreedySearch, a "ticket queue" is constructed for each tour. Then during the search, a "ticket" is placed into the corresponding queue for each vehicle candidate (line 4). If a ticket for $\boldsymbol{s}$ is at the head of its queue, then tour expansion is performed on $\boldsymbol{s}$ (line 8); otherwise, the algorithm enters a wait cycle, exiting the cycle once the ticket is "notified" (lines 6–7). On line 10, the ticket for $\boldsymbol{s}$ is guaranteed to be at the head of its queue. If this line evaluates to false, then the ticket is popped from the queue, and any tickets remaining in the queue move up by one spot. If there is a new ticket at the head of the queue, it is notified, causing the worker holding this ticket to advance (line 14). But if this line evaluates to true, the ticket remains in the queue, and any ticket associated with the reduction incumbent is popped instead (line 11). If $\boldsymbol{s}^*$ is not null once all tours have been evaluated, then the ticket associated with $\boldsymbol{s}^*$ is released after this tour is updated.

Example 6 (Ticket Locking): In Fig. 5, at time t_1, worker W_1 works on all three tours, $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, and $\boldsymbol{s_3}$. Under MR (Fig. 5(a)), these tasks are performed under a lock, whereas under TL (Fig. 5(b)), these tasks proceed because the ticket for W_1 is at the head of the queues for each of the tours, $Q(\boldsymbol{s_1})$, $Q(\boldsymbol{s_2})$, and $Q(\boldsymbol{s_3})$. At time t_2, worker W_1 decides that tour $\boldsymbol{s_1}$ is the reduction incumbent for the request it is processing. Under MR, while $\boldsymbol{s_2}$ is no longer needed by W_1 and there is a task on $\boldsymbol{s_2}$ for worker W_3, worker W_3 cannot acquire the lock on $\boldsymbol{s_2}$ before acquiring the lock on $\boldsymbol{s_1}$ due to the global lock order. But under TL, it can freely work on $\boldsymbol{s_2}$ because it is the next worker in the $Q(\boldsymbol{s_2})$ queue.

D. Discussion

Exclusive locking provides serializable isolation as it is essentially 2PL. On the other hand, while both MR and TL provide a serializable write order to avoid errors such as overloading a vehicle, they do nothing to prevent non-repeatable or phantom reads, and there are also no mechanisms for preventing dirty reads. Hence, they can only offer read uncommitted isolation.

V. Caching and Pruning

The insertion cache exploits a pattern in the insertion pairs to reduce the SP query burden while slack bubbles exploit the deadline constraint to prune these pairs. An insertion cache reduces the SP query burden to just 2 or $2 + n$ single-source (SS) queries, depending on if the road network graph is undirected or directed. Slack bubbles reduce the cost of tour expansion by pruning the insertion pairs that are guaranteed to lead to a deadline constraint violation. Each bubble is easy to construct, is updated only when $\boldsymbol{s}$ changes, and each of the pairs is checked against the pruning condition in $O(1)$ time.

A. Insertion Cache

Insertion cache stores useful SP segments in local memory so they can be reused instead of re-queried. The key observation for the insertion cache is that all SPSP queries needed by insertion-based tour expansion are known at the time of the call, and they all can be found using a small number of SS queries, as shown in

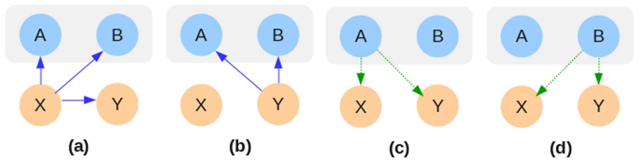


Fig. 6. Forward (blue) and backward (green) SS queries for finding all segments in Fig. 2.

Fig. 6. Answering SS queries to a number of targets is trivially guaranteed to be just as fast, and in practice many times faster, than answering multiple single-pair queries to each of the targets.

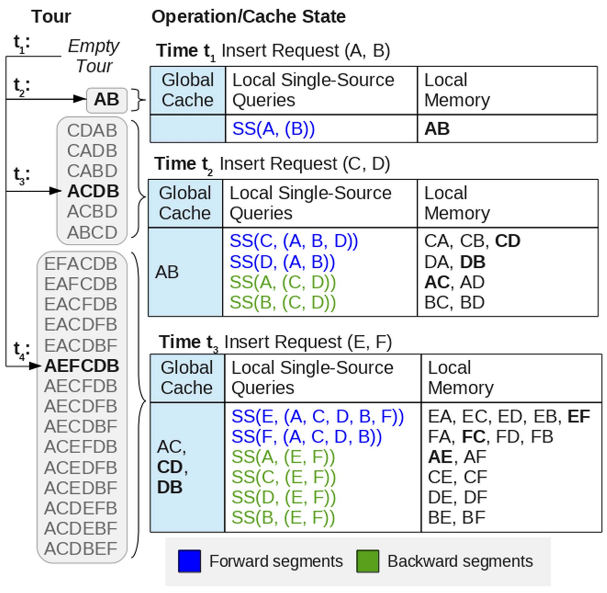
The cache stores $d(s_1, s_2), d(s_2, s_3), \ldots, d(s_{n-1}, s_n)$ for each tour in global shared memory accessible by all workers. When ExpandTour$(\boldsymbol{s}, \boldsymbol{r})$ is called, it finds the answers to the new segments involving the departure u and arrival location v and stores these in local memory. Any single-source SP algorithm can be used to prime the cache, including an SP index. If the search operation returns $\boldsymbol{s}$ as the distance-minimizing tour, then the new segments for $\boldsymbol{s}$ that are created by inserting $\boldsymbol{r}$ into $\boldsymbol{s}$ are added to the permanent store. In any case, the answers in local memory are discarded.

There is a minimum number of segments needed to find the cost of all tour expansion candidates. In addition to the permanently cached segments, all tour candidates require only a few new segments in order to evaluate their cost. These are the "forward segments," $us_1 \ldots us_n$ and $vs_1 \ldots vs_n$, and the "backward segments," $s_1u \ldots s_nu$ and $s_1v \ldots s_nv$. Clearly, these segments can be found by a small number of SS queries. If the road network graph is undirected, only two SS queries are needed, one from u with each $s_i \in \boldsymbol{s}$ as a target and also including v as a target, and one from v with the same targets. If the graph is directed, n additional queries are needed, one from each s_i and with u and v as the targets.

Example 7: In Fig. 7, requests (A, B), (C, D), and (E, F) are successively inserted into an initially empty tour. At time t_1, to insert (A, B), there is only a single tour expansion candidate AB, and the cost $d(A, B)$ is found and stored in the global cache. At time t_2, to insert (C, D), there are 9 new segments across all the possible tour candidates. The cost of each segment is found by issuing two single-source (SS) queries, one originating from C and the other from D, to get the cost of forward segments, and then two additional SS queries to get the cost of backward segments. The minimum-cost tour is found using the costs in local memory in addition to the costs in the global cache, and it turns out to be ACDB. The segments AC, CD, and DB are stored into the global cache. At time t_3, to insert (E, F), there are 17 new segments across all the possible tour candidates. Once again, the cost of each segment is found using $2 + n = 6$ SS queries.

B. Slack Bubbles

Slack bubbles aim to prune infeasible insertion pairs. As each location in the tour must be visited within a certain time window, a proximity-based pruning rule can be extended across *all the segments* in the tour. As each segment consists of two endpoints, the proximity range is elliptical. The departure and arrival locations of a new request can only be inserted between the segments where the ellipse of the segment covers the location. This allows discarding all insertion pairs corresponding to segments that do not meet this condition, as shown in Fig. 8.



Time t_1 Insert Request (A, B)

Global Cache	Local Single-Source Queries	Local Memory
	SS(A, (B))	**AB**

Time t_2 Insert Request (C, D)

Global Cache	Local Single-Source Queries	Local Memory

AB	SS(C, (A, B, D))
	SS(D, (A, B))
	SS(A, (C, D))
	SS(B, (C, D))
		CA, CB, **CD**
		DA, **DB**
		AC, AD
		BC, BD

Time t_3 Insert Request (E, F)

Global Cache	Local Single-Source Queries	Local Memory

AC,
CD,
DB
	SS(E, (A, C, D, B, F))
	SS(F, (A, C, D, B))
	SS(A, (E, F))
	SS(C, (E, F))
	SS(D, (E, F))
	SS(B, (E, F))
		EA, EC, ED, EB, **EF**
		FA, **FC**, FD, FB
		AE, AF
		CE, CF
		DE, DF
		BE, BF

🟦 Forward segments 🟩 Backward segments

Fig. 7. Tour expansion workflow for a directed road network using insertion cache over requests (A, B), (C, D), and (E, F).

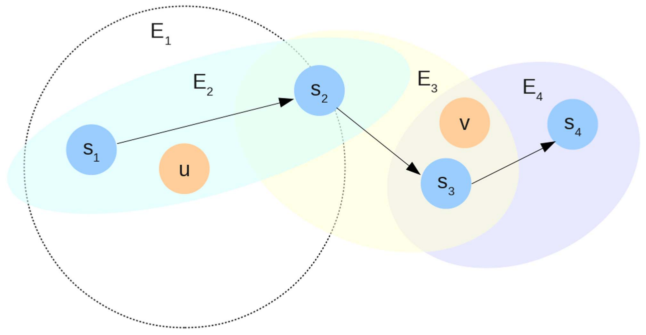


Fig. 8. Example slack bubbles. Request (u, v) is being inserted into tour $s_1s_2s_3s_4$. As u is covered only be the E_2 bubble, it can only be inserted between s_1s_2. Likewise as v is covered by both the E_3 and E_4 bubbles, it can only be inserted between the second and third segments. The feasible insertion pairs are thus (2, 3) and (2, 4). The E_1 bubble shows the convention proximity range around u for proximity pruning.

We derive the widths of each ellipse, or "bubble", based on the deadline constraint and using euclidean distance as a lower bound for travel distance, combined with an estimate of vehicle speed, to yield a lower bound on travel duration.

Definition 7 (Slack Bubble): The slack bubble E_i for segment (s_{i-1}, s_i) of tour $\boldsymbol{s}$, where $2 \leq i \leq |\boldsymbol{s}|$, is the ellipse $E_i = \{p \in \mathbb{R}^2 : ||s_{i-1} - p||_2 + ||p - s_i||_2 \leq 2\alpha\}$ where $2\alpha = d(s_{i-1}, s_i) + \zeta(i)\nu$.

Here, $\mathbb{R}$ are the real numbers, $||\cdot||_2$ is euclidean distance, and ν is the estimated vehicle speed. Function d returns a distance in this case to give the spatial region of the ellipse.

To find the radius 2α, we use the slack array, ζ, from [4]. Briefly, given tour $\boldsymbol{s} = s_1 s_2 \dots s_n$, let (u_i, v_i) be the request associated with s_i, and let $(t_\mathrm{s})_i$ and $(t_\mathrm{e})_i$ be the deadlines associated with this request. We restate the definition of the slack array, ζ, using the concept of slack time.

Definition 8 (Slack Time [6]): The "slack time," $\xi(i)$, of location $s_i \in \boldsymbol{s}$ is $\xi(i) = T(i) - (t_\mathrm{e})_i - \delta(u_i, v_i)$ if $s_i = u_i$, or $\xi(i) = (t_\mathrm{e})_i$ if $s_i = v_i$.

The "slack array", $\zeta(i)$, associated with $\boldsymbol{s}$ is $\zeta(i) = \min_{i \le k \le n}(\xi(k))$. This array is constructed in $O(n)$. The array and bubbles are constructed only when a tour is updated.

Correctness: To show the correctness of slack bubbles, we first restate the conditions that cause the deadline constraint to be violated respective to the insertion pair, (i, j). These conditions are well-known. From [4] and elsewhere, given $\boldsymbol{s} = s_1 s_2 \dots s_n$, insertion pair (i, j), and the slack array for $\boldsymbol{s}$, the deadline constraint is violated:

- If $i = j$ and

$$T(n) + \Delta D > (t_\mathrm{e})_j \qquad \text{if } j = n+1, \text{ or} \qquad (1)$$

$$\Delta D > \zeta(j) \qquad \text{otherwise;} \qquad (2)$$

- If $i \neq j$ and

$$\Delta D_i > \zeta(i) \qquad \text{or} \qquad (3)$$

$$T(n) + \Delta D_j > (t_\mathrm{e})_j \qquad \text{if } j = n+1, \text{ or} \qquad (4)$$

$$\Delta D_j > \zeta(j) \qquad \text{if } j < n+1. \qquad (5)$$

Here are the ΔD equations, also restated from [4]. For brevity, let $/s_1 s_2 \dots s_n/$ mean $\delta(s_1, s_2) + \delta(s_2, s_3) + \cdots + \delta(s_{n-1}, s_n)$. First, if $i = j$, then $\Delta D = /uvs_j/$ if $j = 1$, or $/s_{j-1}uv/$ if $j = n+1$, or $/s_{j-1}uvs_j/$ otherwise. But if $i \neq j$, then $\Delta D = \Delta D_i + \Delta D_j$, where

$$\Delta D_i = \begin{cases} /s_{i-1}us_i/ - /s_{i-1}s_i/ & \text{if } i > 1, \text{ or} \\ /us_i/ & \text{if } i = 1, \end{cases} \qquad (6)$$

$$\Delta D_j = \begin{cases} /s_{j-1}vs_j/ - /s_{j-1}s_j/ & \text{if } j < n+1, \text{ or} \\ /s_{j-1}v/ & \text{if } j = n+1. \end{cases} \qquad (7)$$

Lemma 1: A vehicle is traveling along the tour $s_1 s_2 \dots s_n$ at speed ν. If while traveling from s_{i-1} to s_i, for any i where $2 \le i \le n$, it exits the slack bubble E_i, then it cannot possibly arrive at s_i before the deadline on s_i.

Proof: We just need to show that any of (2), (3), or (5) is true when the vehicle exits the bubble.

Suppose the vehicle passes through two points, u and v, while traversing s_{i-1} to s_i. The minimum travel distance occurs when $u = v$. In this case, the deadline on s_i is violated if $d(s_{i-1}, v) + d(v, s_i) > d(s_{i-1}, s_i) + \zeta(i)\nu$ by combining (6) and (3). If this condition is violated, then any case where $d(u, v) > 0$ will also violate this condition. The right-hand side of this expression equals the width of the slack bubble and the proof is complete. □

Pruning: To find the feasible insertion pairs for request (u, v), we scan all the bubbles twice, once to check which bubbles cover u (by substituting coordinates of u for p in the condition of Definition 7) and again to check which bubbles cover v. While this is $O(n)$, each check uses euclidean distance and is fast. Then, we join the indices of the bubbles to get the insertion pairs. All other pairs are safely pruned.

VI. Experiments

We first evaluate the caching and pruning techniques, then evaluate the locking schemes, and finally evaluate the performance on real-world instances.

Implementation: We implement all algorithms in Julia 1.8.5 on a Ubuntu 20.04.4 machine with eighty Intel Xeon Gold 6242R@3.10 GHz logical cores. Julia is a dynamic high-level programming language capable of C language performance due to its compiler. For SP queries, we use index-free Dijkstra's algorithm with a priority queue. It is possible to use an SP index or time-dependent routing index to speed up these queries. But some smaller real-world platforms may use a third-party provider such as Google Maps to answer SP queries instead of maintaining accurate road network indexes themselves, leading to expensive queries. Hence to better illustrate the magnitude of savings offered by our caching and pruning techniques, we elect to use an index-free approach.

Data Sets: We derive realistic problem instances by using real-world taxi trips taken in Chengdu, China. Our instances are comparable to those in [4]. In total, there are around 500,000 raw taxi trips occurring within the period of one day. We chronologically sample from these trips to build instances with varying numbers of requests and with realistic spatiotemporal distributions. To set the starting locations of the tours, we randomly sample from the trip origins. To set travel deadlines, we use a unitless factor, which we call the "delay tolerance," τ. For request (u, v), the deadline is set to the time that the request is released plus the amount $\tau \cdot \frac{d(u,v)}{\nu}$, where $\tau \ge 1$. The speed ν is set to 10 meters per second.

The road network is available from [15]. There are 18,300 nodes and 52,224 directed edges in the graph. The average time to answer a single-pair query on this graph using our Dijkstra implementation is around 10 ms.

A. Caching and Pruning

The running time of tour expansion depends on the length of the given tour. To evaluate the insertion cache and slack bubbles, we design two special test instances. The first instance ($n = 100$, $m = 100$, $\tau = 100$, $\kappa = \infty$) used to evaluate the cache elicits a wide range of tour lengths, simulating large-capacity vehicles and long chains. The second instance ($n = 1000$, $m = 100$, $\tau = 1.4$, $\kappa = \infty$) used to evaluate slack bubbles is a more realistic instance designed to produce infeasible insertion pairs across a range of tour lengths.

Insertion Cache: We compare against three differently sized LRU caches, LRU100, LRU1k, and LRU10 k, sized with 100, 1,000, and 10,000 cache slots per vehicle, respectively. On our test instance, we observed no difference beyond around 10,000 cache slots. Normally, one LRU serves as a global cache. But we observed that a global cache experiences a high rate of evictions as certain long-tour vehicles contend for the limited slots. Hence to make the experiments more comparable, we attach a local LRU per vehicle instead of using a global cache.

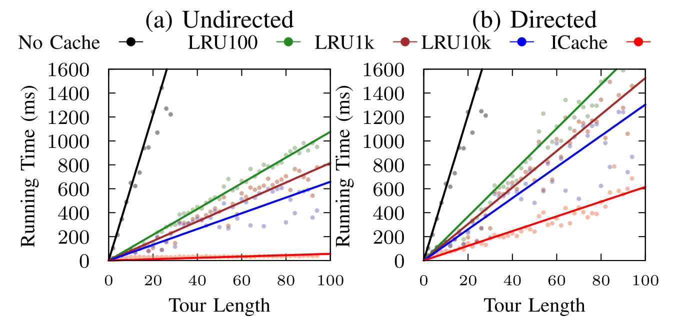


Fig. 9. Comparison of SP caching strategies on an (a) undirected and (b) directed road network.

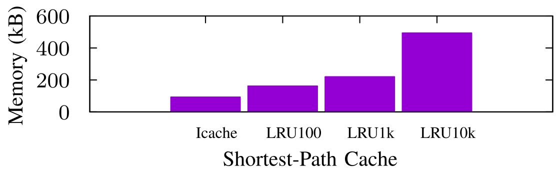


Fig. 10. Memory usage of SP caches.

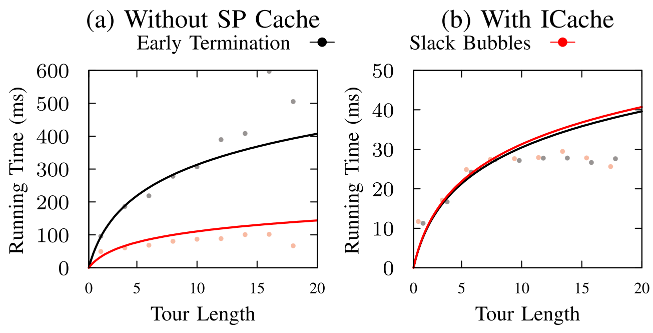


Fig. 11. Early termination and slack bubble insertion pair pruning strategies with (b) and without (a) SP cache.

Fig. 9 shows the running time of tour expansion along with best-fit lines. The observed linearity agrees with the $O(n)$ complexity of DP tour expansion. For insertion cache (ICache), the time includes the time for priming the cache. On the undirected graph, ICache achieves over 10x speed-up for long tours compared to the next best, LRU10 k (Fig. 9(a)). On the directed graph, ICache delivers about 5x speed-up for long tours compared to LRU10 k (Fig. 9(b)). With LRU, a cache miss as a result of a never-before seen query or as a result of eviction leads to a new SP query. But with ICache, there are no cache misses, and the cache itself is primed using fast SS queries. *On our machine, one SS query to 100 targets takes only 30 ms compared to 1,000 ms for 100 single-pair queries to the same targets.*

Fig. 10 shows memory usage measured during execution. Over time, LRU retains many SPs that are no longer necessary. But ICache discards unnecessary SPs after each tour expansion, leading to a small memory footprint.

Slack Bubbles: Fig. 11 shows the running time of tour expansion under early termination and under slack bubbles, along with regression curves to the nearest logarithm. We expect long tours to yield fewer feasible insertion pairs, hence the log shape. When there is no SP cache (Fig. 11(a)), slack bubbles achieves about 5x speed-up for long tours compared to early termination. Early termination requires entering the tour expansion loop and performing SP queries to decide whether or not to continue evaluating an insertion pair, but slack bubbles yields all the feasible pairs upfront, avoiding these queries. Interestingly, when there is a cache (Fig. 11(b)), the two techniques yield equivalent running times, indicating that the main bottleneck is by far the cost of SP queries.

B. Locking Schemes

We introduce a realistic test instance ($n = 1000$, $m = 4000$, $\kappa = 3$) to evaluate the locking schemes. Locking schemes are affected by the degree of contention, so to elicit different levels of contention, we vary τ between 1.2 and 1.8. During vehicle search, we use conventional proximity pruning to obtain vehicle candidates. The number of candidates is small at low τ, so the chance that any two requests share the same candidates is also small, leading to low contention. At high τ, the chance of shared candidates is much greater, leading to high contention. We observe that when $\tau = 2.0$, around 95% of the vehicles are candidates, leading to full contention. To evaluate the scalability of different locking schemes, we vary the number of processors. We conduct all experiments in batch mode, where all requests are released at the same time, and we report throughput as the number of requests in a batch divided by the total time.

We compare exclusive, MR, and TL in addition to MVCC and a single-worker baseline. For all approaches, we perform vehicle search serially or in parallel. Request assignment is performed serially for single-worker and in parallel for all other approaches. For all approaches, we use DP tour expansion, early termination tour pruning, and an LRU cache. In other words, we do not use any new techniques for tour expansion, pruning, and caching, and the only difference is the locking scheme. Fig. 12 shows the results, along with regression curves to the nearest logarithm. For MVCC, we observed that some requests with many candidates repeatedly abort, leading to "livelocks". To avoid these situations, we cap the amount of aborts to up to three. In other words, if a request is not assigned after three aborts, we drop the request.

Effect of Contention: Under low contention, when vehicle search is conducted serially (Fig. 12(a)), MR yields the highest throughput followed by MVCC. But when vehicle search is conducted in parallel (Fig. 12(b)), TL yields the highest throughput, followed by exclusive locking. There are several reasons for this reversal. First, MR acquires locks in a global order, which requires synchronizing across vehicle processing threads. Second for MVCC, parallelizing vehicle search introduces thread-level contention, increasing the risk of aborts. Third, exclusive locking performs well under low contention because the locks rarely block other workers.

Under high contention, the same general observations hold, except that exclusive locking and MVCC perform significantly

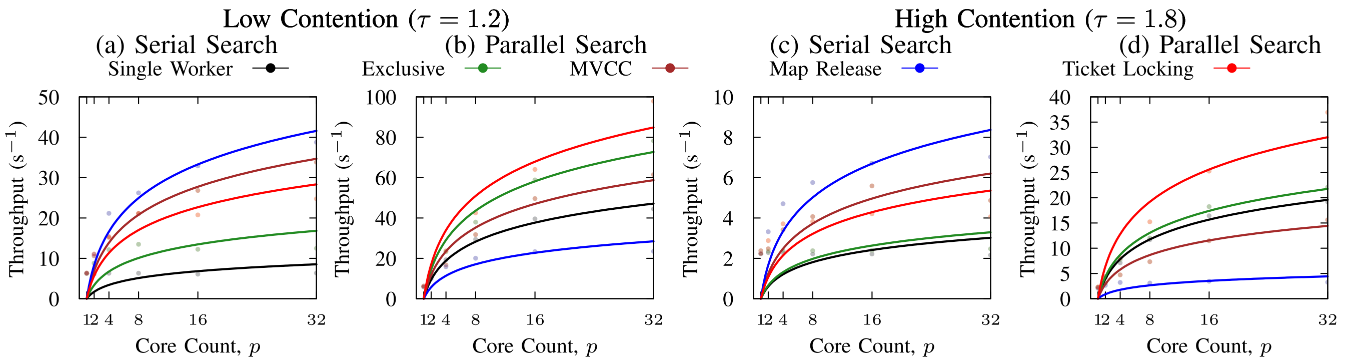


Fig. 12. Scalability under low (a, b) and high (c, d) contention.

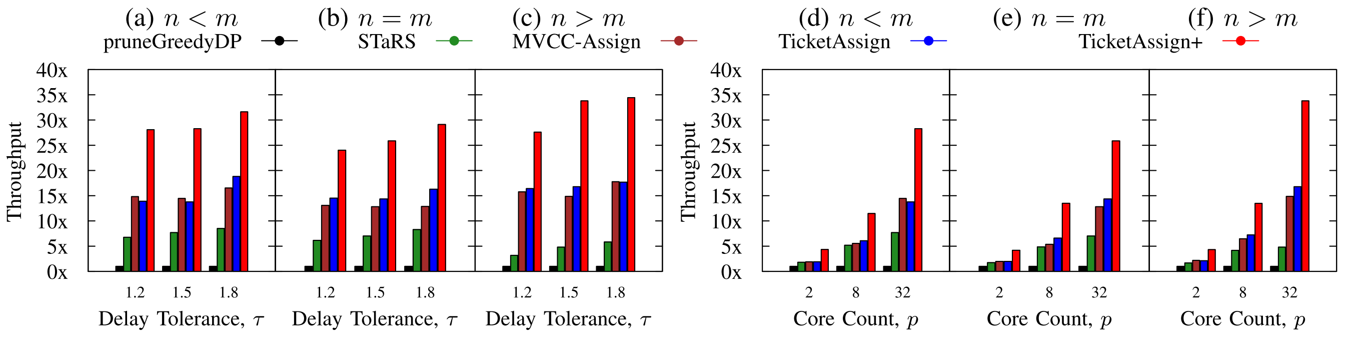


Fig. 13. Effect of delay tolerance (a, b, c) and core count (d, e, f) on throughput across various request and vehicle ratios.

TABLE II
SCENARIO PARAMETERS (DEFAULTS IN BOLD)

Parameter	Range
Vehicle Ratio, $n : m$	1000:3000, 3000:3000, **3000:1000**
Delay Tolerance, τ	1.2, **1.5**, 1.8
Capacity, κ	**3**, 6, 12
Processor Cores	2, 8, **32**

worse, especially when vehicle search is conducted in parallel (Fig. 12(d)). This is due to the high amount of blocking in the case of exclusive locking and the high amount of aborts in the case of MVCC. A side effect of high τ is that more vehicles and tours become feasible, explaining the overall lower throughput.

Scalability: Under serial search, scalability comes solely from the multiple workers. As shown in Fig. 12(a)(c), MR, TL, and MVCC all achieve logarithmic scaling while exclusive and single-worker algorithms do not scale. Under parallel search, scalability also comes from parallelizing vehicle search. In this case, exclusive locking and single-worker algorithhms do exhibit logarithmic scaling. The lack of scalability for MR is caused by lock synchronization across vehicle threads.

C. Real-World Performance

Our real-world baselines consist of the state-of-art single-worker algorithms pruneGreedyDP [4] and STaRS [10], and we include an algorithm called MVCC-Assign to serve as a multiple-worker baseline, retaining the hard limit of three aborts as before. We compare these baselines against a multiple-worker algorithm, TicketAssign, that is the same as MVCC-Assign except using TL instead of MVCC, and TicketAssign+, that is the same as TicketAssign but replaces early termination and LRU with slack bubbles and insertion cache. Table III summarizes these algorithms. We consider three scenarios, one with many vehicles and few requests ($n < m$), one with equal numbers of vehicles and requests ($n = m$), and one with few vehicles but many requests ($n > m$). Other parameters are shown in Table II.

Figs. 13 and 14 report performance and quality characteristics of the algorithms. For performance, we report throughput as a multiple of that achieved by pruneGreedyDP, and we report the average request latency in Fig. 14(g),(h),(i). As all requests are released at once in a batch, the latency for one request is the duration between the start of the batch and when the request completes processing. For quality, we report the number of assignments as a percentage of n, and we report the normalized distance as the sum tour distances divided by the number of assignments. We use normalized distance as an indicator of tour characteristics. For example, for the same number of assignments, a larger normalized distance indicates more distance traveled per passenger meaning costlier tours.

Effect of Number of Requests and Vehicles: When $n < m$ there is less contention for tours, and as Fig. 13(a) shows, MVCC-Assign is able to achieve slightly greater throughput than TicketAssign in some cases. But this result highly depends on the abort limit. In any case, when n grows, MVCC-Assign throughput begins to drop (Fig. 13(b)). Tours also get longer, slowing down tour expansion. This effect can be seen from STaRS throughput becoming closer to pruneGreedyDP in Fig. 13(c). But even in this case, TicketAssign+ vastly outperforms both

TABLE III
COMPARED ALGORITHMS. ALGORITHMS IN BOLD USE TECHNIQUES INTRODUCED IN THIS ARTICLE

Algorithm	Number of Workers	Vehicle Search	Concurrency Control	Tour Expansion	Tour Pruning	SP Cache
pruneGreedyDP [4]	Single	Serial	—	DP Insertion	Early Termination	LRU
STaRS [10]	Single	Parallel	—	Naive Insertion	Early Termination	LRU
MVCC-Assign	Multiple	Parallel	MVCC	DP Insertion	Early Termination	LRU
TicketAssign	Multiple	Parallel	Ticket Locking	DP Insertion	Early Termination	LRU
TicketAssign+	Multiple	Parallel	Ticket Locking	DP Insertion	Slack Bubbles	Insertion Cache

* The original StaRS uses an all-pairs SP cache over the Manhattan road network. For larger road networks, the sheer size of the road network may make memory requirements prohibitively large, so we use an LRU cache in our experiments.

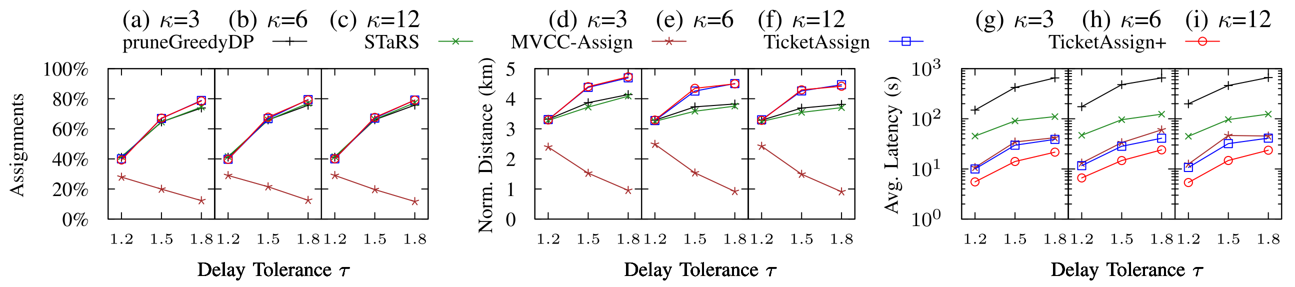


Fig. 14. Effect of delay tolerance on assignments (a, b, c), distance (d, e, f), and latency (g, h, i) across various capacities.

STaRS and pruneGreedyDP due to the insertion cache and slack bubbles.

Effect of Delay Tolerance (τ): Large τ leads to more vehicle candidates and longer tours, culminating in greater latency as shown in Fig. 14(g), (h), (i). Large τ also leads to higher contention. Fig. 13(a), (b), (c) show that the throughput of all algorithms stay at similar levels relative to pruneGreedyDP, except for TicketAssign+ which increases its relative advantage to pruneGreedyDP due to more efficient tour expansion.

Fig. 14(a), (b), (c) show that assignments increase as τ increases due to more feasible tours, while Fig. 14(d), (e), (f) show that normalized distance also increases due to longer tours and more chains. MVCC-Assign reaches its abort limit more frequently under high τ, leading to decreased assignments and decreased normalized distance, as it can only manage to produce short tours. Interestingly, TicketAssign and TicketAssign+ yield greater normalized distance compared to pruneGreedyDP and STaRS while yielding the same number of assignments, indicating that the tours they produce are costlier, perhaps due to read-write errors. The worst-case difference is around 1 extra kilometer per passenger.

Scalability: As shown in Fig. 13(d), (e), (f), STaRS stops scaling after about 8 cores while the multiple-worker algorithms all scale beyond 8 cores.

Effect of Tour Capacity (κ): Larger κ increases the possibility of sharing. As shown in Fig. 14(d), (e), (f), increasing κ from 3 to 12 causes the normalized distance to decrease slightly, by about 0.3 kilometers per passenger, for all algorithms except MVCC-Assign.

D. Discussion

We offer the following takeaways. (1) Reducing SP queries is more useful for lowering latency than tour pruning, and a cache can avoid issuing new queries. If memory permits, an all-pairs SP cache, as in STaRS, is most effective. Otherwise, insertion cache is targeted at ridesharing and is preferred over LRU. (2) For low contention workloads, exclusive locking achieves high throughput while offering serializability. If serializability is not necessary, TL can increase throughput but produces costlier tours. (3) For high contention workloads, MVCC suffers from too many aborts. Instead, TL can substantially increase throughput while achieving high assignments, but again with costlier tours. (4) If the request rate is not too high, then pruneGreedyDP or STaRS can offer better tours while also achieving high assignments.

Future Work: We mention three areas for future work. First, insertion cache assumes that the cost of a segment is independent of the ordering of the segment in the tour. This precludes it from order-dependent cost functions such as for time-dependent road networks [37], requiring a new priming mechanism. Second, MR and TL cannot be used for applications with a need for strong read-write consistency. However, they may be other applications where this requirement can be relaxed, for example other matching and planning problems such as those listed in [38]. Third, as mentioned in Section III, how to extend our techniques to other algorithms including demand-aware search algorithms such as [7], [26], join-based algorithms such as [13], [28], [39], and other tour expansion techniques such as [23] remains another area of future work.

VII. Conclusion

In this paper, we give techniques to speed up and scale out ridesharing search. The insertion cache uses fast cache priming to reduce the burden of shortest-path queries, while slack bubbles can be used to prune infeasible tour expansions. The map-release and ticket locking schemes sacrifices read-write consistency in order to increase the concurrency of request assignment. Ticket locking scales well with the number of processors, and together, these techniques outperform existing approaches across a wide range of instances.

References

[1] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Quantifying the benefits of vehicle pooling with shareability networks," in *Proc. Nat. Acad. Sci. USA*, vol. 111, no. 37, pp. 13 290–13 294, 2014.

[2] R. Mehrotra, J. McInerney, H. Bouchard, M. Lalmas, and F. Diaz, "Towards a fair marketplace: Counterfactual evaluation of the trade-off between relevance, fairness & satisfaction in recommendation systems," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 2243–2251.

[3] B. Shen, Y. Huang, and Y. Zhao, "Dynamic ridesharing," *SIGSPATIAL Special*, vol. 7, no. 3, pp. 3–10, 2016.

[4] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," in *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1633–1646, 2018.

[5] C. Moon, J. Kim, G. Choi, and Y. Seo, "An efficient genetic algorithm for the traveling salesman problem with precedence constraints," *Eur. J. Oper. Res.*, vol. 140, no. 3, pp. 606–617, 2002.

[6] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson, "A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows," *Transp. Res. Part B*, vol. 20, no. 3, pp. 243–257, 1986.

[7] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, and K. Xu, "Unified route planning for shared mobility: An insertion-based framework," *ACM Trans. Database Syst.*, vol. 47, no. 1, 2022, Art. no. 2.

[8] S. Ma, Y. Zheng, and O. Wolfson, "T-Share: A large-scale dynamic taxi ridesharing service," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 410–421.

[9] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metrewar, N. Annamaneni, and K. Chattopadhyay, "Xhare-a-Ride: A search optimized dynamic ride sharing system with approximation guarantee," in *Proc. IEEE Int. Conf. Data Eng.*, 2017, pp. 1117–1128.

[10] M. Ota, H. Vo, C. Silva, and J. Freire, "STaRS: Simulating taxi ride sharing at scale," *IEEE Trans. Big Data*, vol. 3, no. 3, pp. 349–361, Sep. 2017.

[11] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.

[12] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," in *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 2017–2028, 2014.

[13] Y. Zeng, Y. Tong, Y. Song, and L. Chen, "The simpler the better: An indexing approach for shared-route planning queries," in *Proc. VLDB Endowment*, vol. 13, no. 13, pp. 3517–3530, 2020.

[14] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1197–1210.

[15] A. Karduni, A. Kermanshah, and S. Derrible, "A protocol to convert spatial polyline data to network formats and applications to world urban road networks," *Sci. Data*, vol. 3, 2016, Art. no. 160046.

[16] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin, "Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees," in *Proc. VLDB Endowment*, vol. 13, no. 5, pp. 602–615, 2020.

[17] H. Yuan, G. Li, Z. Bao, and L. Feng, "Effective travel time estimation: When historical trajectories over road networks matter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2135–2149.

[18] S. C. Ho, W. Y. Szeto, Y.-H. Kuo, J. M. Y. Leung, M. Petering, and T. W. H. Tou, "A survey of dial-a-ride problems: Literature review and recent developments," *Transp. Res. Part B*, vol. 111, pp. 395–421, 2018.

[19] N. Ta, G. Li, T. Zhao, J. Feng, and H. Ma, "An efficient ride-sharing framework for maximizing shared route," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 2, pp. 219–233, Feb. 2018.

[20] J. J. Pan, G. Li, and J. Hu, "Ridesharing: Simulator, benchmark, and evaluation," in *Proc. VLDB Endowment*, vol. 12, pp. 1085–1098, 2019.

[21] S. Ma, Y. Zheng, and O. Wolfson, "Real-time city-scale taxi ridesharing," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1782–1795, Jul. 2015.

[22] Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu, and W. Li, "An efficient insertion operator in dynamic ridesharing services," in *Proc. IEEE Int. Conf. Data Eng.*, 2019, pp. 1022–1033.

[23] Y. Xu et al., "An efficient insertion operator in dynamic ridesharing services," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3583–3596, Aug. 2022.

[24] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen, "Price-and-time-aware dynamic ridesharing," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1061–1072.

[25] L. Chen, Y. Gao, Z. Liu, X. Xiao, C. S. Jensen, and Y. Zhu, "PTrider: A price-and-time-aware ridesharing system," in *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 1938–1941, 2018.

[26] J. Wang et al., "Demand-aware route planning for shared mobility services," in *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 979–991, 2020.

[27] M. Ota, H. Vo, C. Silva, and J. Freire, "A scalable approach for data-driven taxi ride-sharing simulation," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 888–897.

[28] Y. Zeng, Y. Tong, and L. Chen, "Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees," in *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 320–330, 2019.

[29] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core OLTP under high contention," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1583–1598.

[30] C. Diaconu et al., "Hekaton: SQL Server's memory-optimized OLTP engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1243–1254.

[31] K. Kim, T. Wang, R. Johnson, and I. Pandis, "ERMIA: Fast memory-optimized database system for heterogeneous workloads," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1675–1687.

[32] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 21–35.

[33] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. Symp. Operating Syst. Princ.*, 2013, pp. 18–32.

[34] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time traveling optimistic concurrency control," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1629–1642.

[35] H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 691–706.

[36] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," in *Proc. VLDB Endowment*, vol. 10, no. 2, pp. 49–60, 2016.

[37] Z. Gong, Y. Zeng, and L. Chen, "A fast insertion operator for ridesharing over time-dependent road networks," 2023, *arXiv:2303.03614*.

[38] Y. Tong, Z. Zhou, Y. Zeng, L. Chen, and C. Shahabi, "Spatial crowdsourcing: A survey," *VLDB J.*, vol. 29, no. 1, pp. 217–250, 2019.

[39] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," in *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 3, pp. 462–467, 2017.

James Jie Pan is a postdoctoral researcher with the Department of Computer Science, Tsinghua University, in Beijing, China. His research interests include spatial database systems and applications.

Guoliang Li is a professor with the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include data quality, database systems, and data cleaning and integration.