A Spatio-temporal Optimized Framework for High-throughput Shortest Distance Query Processing on Dynamic Road Networks

Xinjie Zhou
Division of EMIA, HKUST and DSA Thrust, HKUST(GZ)
Hong Kong and Guangzhou, China
xzhouby@connect.ust.hk

Lei Li
DSA Thrust, HKUST(GZ) and CSE Department, HKUST
Guangzhou and Hong Kong, China
thorli@ust.hk

Mengxuan Zhang
School of Computing, Australian National University
Canberra, Australia
mengxuan.zhang@anu.edu.au

Xiaofang Zhou
CSE Department, HKUST and DSA Thrust, HKUST(GZ)
Hong Kong and Guangzhou, China
zxf@cse.ust.hk

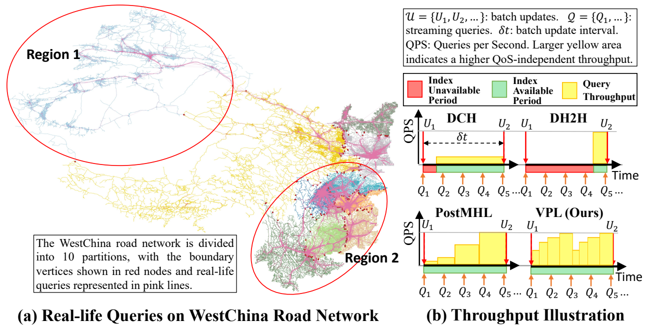
ABSTRACT

High-throughput shortest path query processing on dynamic road networks is fundamental to many location-based services. Existing solutions achieve this through efficient query processing or rapid index maintenance. However, they overlook the fact that partial dynamic updates cannot affect all queries, missing opportunities to leverage the unaffected index for higher throughput. To this end, this paper proposes a novel spatio-temporal optimized framework called *VPSP* to further exploit the availability of the shortest path index during index maintenance to enhance query throughput. Specifically, we first provide a theoretical analysis of unaffected queries and propose a *two-phase index validation* algorithm to efficiently identify the unaffected queries. Based on this, we put forward the *VPSP* framework, which adopts a non-trivial partitioned shortest path (PSP) index called *Validation-based Partitioned Labeling (VPL)* to exploit tree decomposition and thread parallelization for fast query and update efficiency, and a novel *rotating update schedule* to maximize unaffected query number. Last, we put forward a dynamic algorithm for *VPL* to efficiently process mixed updates and further enhance its throughput by *tree height-aware PSP ordering* optimization. Experimental evaluation on real-world datasets demonstrates our solution outperforms state-of-the-art in query throughput, achieving up to 2 orders of magnitude improvement.

PVLDB Reference Format:
Xinjie Zhou, Lei Li, Mengxuan Zhang, and Xiaofang Zhou. A Spatio-temporal Optimized Framework for High-throughput Shortest Distance Query Processing on Dynamic Road Networks. PVLDB, 18(7): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Finding the Shortest Path (SP) from one place to another is a building block in many location-based services (LBSs), such as route planning [5, 30], traffic optimization [25, 56], POI recommendation [27], and k-nearest neighbor searching [29, 68], etc. These


Region 1
Region 2
The WestChina road network is divided into 10 partitions, with the boundary vertices shown in red nodes and real-life queries represented in pink lines.
$\mathcal{U} = \{U_1, U_2, ...\}$: batch updates. $\mathcal{Q} = \{Q_1, ...\}$: streaming queries. δt: batch update interval. QPS: Queries per Second. Larger yellow area indicates a higher QoS-independent throughput.
Index Unavailable Period
Index Available Period
Query Throughput
DCH
DH2H
PostMHL
VPL (Ours)
QPS
Time
U_1
U_2
δt
Q_1 Q_2 Q_3 Q_4 Q_5 ...

(a) Real-life Queries on WestChina Road Network (b) Throughput Illustration

Figure 1: Real-life Query Distribution on WestChina and Throughput Comparison of Different Solutions

location-based services typically encounter numerous shortest path or distance queries, necessitating high-throughput SP query processing on road networks. For instance, as one of China's most widely used ride-sharing platforms, *Didi* had about 462,962 queries per second in 2018 [57]. Uber may also need to compute millions of queries per second [22]. The basic solutions for SP computation are index-free algorithms (*e.g., Dijkstra's* [13], A^* [20], *BiDijkstra* [41] *etc.*) which directly traverse the graph on the fly in a best-first manner but suffer from low query efficiency. Therefore, a plethora of shortest path indexes [1–3, 9, 17, 33, 34, 42, 46] are proposed, which improved the query efficiency by 2-6 orders of magnitude.

However, real-world road networks are inherently dynamic as the edge weights (travel time) usually evolve with frequent updates due to the changing traffic flow or traffic accidents [43, 59]. For example, Beijing and New York City had about 13.8 and 5.78 edge weight updates per second on average [58, 67]. To adapt to the dynamics of road networks, many SP index maintenance algorithms [4, 43, 44, 53, 59, 61, 62, 65, 66, 71, 72] are also investigated in recent years. Among them, *Dynamic Contraction Hierarchies (DCH)* [43] and *Dynamic Hierarchical 2-Hop labeling (DH2H)* [59] are two state-of-the-art methods for road networks in terms of index update and query processing efficiency. Nevertheless, as identified in [63] and [72], existing dynamic algorithms still fail to handle high throughput queries due to either slow index maintenance (*e.g., DH2H*) or low query efficiency (*e.g., DCH*). Worse yet, the *large graph size* and *Quality of Service (QoS)* requirement such as *query response time* (*i.e.,* the time taken from the arrival of a query Q to the time at which Q's answer is computed) [21, 72] could further deteriorate the update time and throughput. To address this, a recent

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:XX.XX/XXX.XX

work [72] proposes a novel *partitioned shortest path (PSP)* index called *PostMHL* to exploit the thread parallelization for fast index update while continuously enhancing its query efficiency during the index maintenance. As shown in Figure 1-(b), *PostMHL* consecutively enhances its query efficiency (QPS, Queries per Second) over four stages, significantly outperforming *DCH* and *DH2H*-based solutions and achieving the state-of-the-art throughput. However, *PostMHL* has two drawbacks. Firstly, its query throughput remains constrained because the index is unavailable during maintenance, and the maintenance duration tends to increase with larger update volumes or network sizes. Secondly, it relies on *Tree Decomposition-based (TD)* for graph partitioning (i.e., TD-partitioning), which makes the partition result largely beyond our control, making it difficult to optimize graph partitioning for higher throughput.

Motivation. Figure 1-(a) presents the WestChina road network partitioned by *PUNCH* [12] and the real-life queries obtained from NavInfo [40]. We observe that real-life queries are generally aggregated in clusters, and a good partition result (*e.g.,* the one generated by *PUNCH*) tends to comply with these query clusters. Besides, partial edge updates cannot affect all queries (*e.g.,* the updates within Region 1 could not affect the queries in Region 2 due to the long distances between them), which provides an opportunity to leverage the stale index to process those unaffected queries during index maintenance. In other words, the index could be partially available even during maintenance. Motivated by this, in this paper, we propose a spatio-temporal optimized framework that further exploits the availability of the PSP index during index maintenance, thereby improving query throughput on dynamic road networks.

Challenges. However, it is challenging to achieve this goal.

Challenge 1: How to efficiently identify the unaffected queries given a batch of updates? Although the unaffected queries seem to be determined once the updates are given, there is no theoretical analysis of how to identify unaffected queries before or during index maintenance. To this end, we theoretically analyze the identification of unaffected queries given an edge update and reveal that the PSP index is an effective way to identify the unaffected queries for batch updates. Besides, we propose a novel *two-stage index validation* algorithm to efficiently identify the unaffected queries. (Section 4)

Challenge 2: How to design the underlying PSP index and spatio-temporal optimized framework for high query throughput? Although *PostMHL* has exceptional index update and query efficiency, its partition result is not adjustable due to the *TD-partitioning*. Since the real-life queries on road networks tend to aggregate in clusters, it is important to adjust the graph partitions to comply with the query clusters for higher throughput. To this end, we first propose a *Validation-based Partitioned Labeling (VPL)* that supports various partition results and achieves cutting-edge query efficiency and index update by building a tree decomposition on graph partitions. Based on this, we propose the *VPSP* framework adopting a novel *rotating update schedule* that subtly schedules the arrival of batch updates on different partitions without violating the update interval constraint to maximize the unaffected query number. (Section 5)

Challenge 3: How to design dynamic algorithms for mixed updates and further optimize the query throughput? Real-life updates on road networks are usually processed in batches, with each batch containing both edge weight decrease and increase updates. However, all existing dynamic algorithms generally deal with these two types of updates separately and can hardly process them simultaneously, making our *VPSP* framework infeasible. To address this, we propose a novel dynamic algorithm to efficiently process mixed updates for *VPL* and further put forward *tree height-aware PSP ordering* optimization to enhance the query throughput. (Section 6)

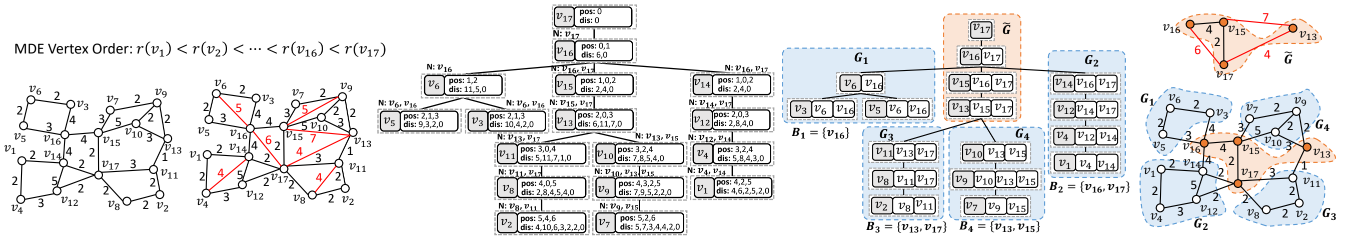
Contributions. We summarize our contributions as follows:

- We provide a theoretical analysis of the unaffected query identification and propose a novel *two-phase index validation* algorithm to efficiently detect unaffected queries for batch updates.
- We propose a novel *VPL* index to achieve cutting-edge query and update efficiency and support various partition results. Based on this, we put forward a spatio-temporal optimized *VPSP* framework with a *rotating update schedule* to exploit the availability of *VPL* index during index maintenance for high query throughput.
- We propose a novel dynamic algorithm to support mixed updates for *VPL* and put forward *tree height-aware PSP ordering* optimization to further enhance the throughput.
- Extensive experiments on real-world datasets demonstrate the effectiveness of our techniques, yielding up to two orders of magnitude higher query throughput than the state-of-the-art.

2 PRELIMINARIES

Let $G = (V, E)$ be a weighted road network where the vertex set V represents intersections and the edge set $E \subseteq V \times V$ represents road segments. We use $n = |V|$ and $m = |E|$ to denote the vertex number and edge number. Each edge $e(u, v) \in E$ is associated with a positive weight (*i.e.,* travel time) $|e(u, v)| = w(u, v)$. The edge weight $w(u, v)$ can change dynamically in the range of $[w^*(u, v), \infty)$, where $w^*(u, v)$ is the smallest travel time determined by the spatial length and corresponding speed limit of edge $e(u, v)$. For each vertex $v \in V$, its neighbors are denoted as $N_G(v) = \{u|(u, v) \in E\}$ with *degree* $deg_G(v) = |N_G(v)|$. Besides, each vertex $v \in V$ is also associated with a *vertex order* $r(v)$ representing its importance in G. A *path* p_G from s to t is a sequence of consecutive vertices $p_G = \langle s = v_0, ..., v_j = t \rangle = \langle e_1, ..., e_j \rangle$ with length $len(p_G) = \sum_{e \in p_G} |e|$. The *shortest path* $sp_G(s, t)$ from s to t is the path with the minimum length. The *shortest distance* query $Q(s, t)$ aims to compute the length of $sp_G(s, t)$, *i.e.,* $d_G(s, t)$. We omit the notation G when the context is clear. We focus our discussion on undirected graphs while it is easy to extend our techniques to directed graphs. In practice, both the topological structure (vertex/edge insertion/deletion) and edge weight (increase/decrease) of G evolve over time. This paper focuses on edge weight updates as they occur more frequently in road networks, and the topological changes can be transformed into edge weight updates [43, 59, 61, 63].

Following [38, 72], we adopt the *batch update arrival model* to handle the edge updates, *i.e.,* the updates are processed in batches with the update interval being δt seconds. Such batch processing of updates is common in most LBSs, *e.g.,* TomTom [39], INRIX [47] update the road networks every minute while AutoNavi [54] updates every few minutes. Besides, we assume the queries arrive at the system as a *Poisson process* and are queued for processing [21, 72]. When each update batch arrives, we assume the system immediately handles the updates before the query processing to avoid *staleness* [21, 45, 72]. If the update time $t_u \geq \delta t$, the system *throughput* (*i.e.,* the number of processed queries per unit time) is zero


MDE Vertex Order: $r(v_1) < r(v_2) < \cdots < r(v_{16}) < r(v_{17})$
$B_1 = \{v_{16}\}$
$B_2 = \{v_{16}, v_{17}\}$
$B_3 = \{v_{13}, v_{17}\}$
$B_4 = \{v_{13}, v_{15}\}$
G_1
G_2
G_3
G_4
$\tilde{G}$

(a) Example Graph G (b) CH Shortcut Graph $\mathcal{G}$ (c) H2H Index (d) TD-partitioning (e) TD-partition Result

Figure 2: Example Road Network G, CH Index $\mathcal{G}$, H2H Index, and TD-partitioning

since it spends all its time on the updates. Same with [72], we adopt *average query response time* R_q^* as a QoS constraint. Let t_q, V_q be the average and variance of query processing time, respectively; and t_u be the average update time; λ_q^* be the *maximum average throughput*. As per [72], for SP index with only one type of query processing, $\lambda_q^* \le \min\{\frac{2\cdot(R_q^*-t_q)}{V_q+2\cdot R_q^*\cdot t_q - t_q^2}, \frac{\delta t - t_u}{t_q\cdot\delta t}\}$. The *High Throughput Shortest Path (HTSP)* problem studied in this paper aims to maximize the *maximum average throughput* λ_q^*.

3 EXISTING SOLUTIONS

3.1 Hierarchy-based Solution

Contraction Hierarchy (CH) [17, 43] is a hierarchical method for SP computation. We denote its dynamic version as *DCH* [43]. The key idea of CH is to build a shortcut graph $\mathcal{G}$ by iteratively contracting vertices according to a pre-defined vertex order r. In particular, it iteratively contracts the least important vertex v_i while preserving the distance information by adding shortcuts among the neighbors of v_i. The static CH index [17] performs a *Dijkstra*'s search to decide whether to add those shortcuts or not during the contraction while *DCH* directly adds all-pair shortcuts among the neighbors of v_i to enable fast index update. When processing a query, they perform a modified bidirectional *Dijkstra*'s search on $\mathcal{G}$ in a bottom-up manner (*i.e.,* search from low-order vertices to high-order vertices) to reduce the search space. *DCH* [43] maintains the shortcut in a *bottom-up* manner, achieving exceptional index update efficiency [43, 63].

Example 1. *Figure 2-(a) shows an example road network G and its MDE vertex order. Figure 2-(b) presents the CH shortcut graph $\mathcal{G}$, where the red lines are newly added shortcuts. For instance, the shortcut $sc(v_{11}, v_8) = 4$ is generated during the contraction of v_2.*

3.2 Hop-based Solution

Hierarchical 2-Hop Labeling (H2H) [42] is a *2-hop labeling* [10] based on the *tree decomposition (TD)*.

Definition 1 (Tree Decomposition). *A tree decomposition of G, denoted as T, is a tree composed of tree nodes*[1] *(subsets of vertices in V). In particular, for each $v \in V$, there is a corresponding tree node $\{X(v)|X(v) \subseteq V, v \in X(v)\}$ such that [34, 42]: (1) $\bigcup_{v\in V} X(v) = V$; (2) $\forall(u,w) \in E, \exists X(v), v \in V$ such that $\{u,w\} \subseteq X(v)$; (3) $\forall v \in V$, the set $\{X(u)|v \in X(u), u \in V\}$ induces a subtree of T.*

The *treewidth* of T is $tw(T) = \max_{X\in T}|X| - 1$ and the *tree height* $h(T)$ is the maximum depth of all nodes. The depth of a tree node $X(v)$ is the distance from it to the root node. We obtain T through *minimum degree elimination (MDE)* [8, 55]. In particular, we start by initializing G^0 as G. In each i^{th} round ($i \in [1,n]$), we extract minimum-degree vertex v along with its neighboring edges from G^{i-1}, generating a node $X(v) = \{v\} \cup X(v).N$, where $X(v).N$ are the neighbors of v in G^{i-1} (also called *accessory neighbors* of v). We then insert/update the all-pair edges among $X(v).N$ in G^{i-1} to preserve the shortest distances, forming a contracted graph G^i. We define all-pair edges formed in the MDE process as *local shortcuts* (or *shortcut* when the context is clear). The vertex order r is consistent with the contraction order of MDE. T is formed by setting $X(u)$ as the patent of $X(v)$, where u is the lowest-order vertex in $X(v).N$. We represent the ancestors of $X(v)$ as $X(v).A$. Given T, H2H index defines of two arrays for each vertex $v \in V$: *distance array* $X(v).dis$ stores the distance $d(v,u)$ for each ancestor $u \in X(v).A$; *position array* $X(v).pos$ records the position of neighbor $u \in X(v)$ in $X(v).A$. For query processing of $Q(s,t), \forall s,t \in V$, suppose X is the *Lowest Common Ancestor (LCA)* [7] of $X(s)$ and $X(t)$, then $Q(s,t) = \min_{i\in X.pos}\{X(s).dis[i] + X(t).dis[i]\}$. Due to the pruned label entries by LCA, H2H [42] and its variant P2H [9] achieve state-of-the-art query efficiency on road networks. The dynamic version of *H2H* is denoted as *DH2H* [59].

Example 2. *Figure 2-(c) presents the H2H index on an MDE-based tree decomposition T with $tw(T) = 3$ and $h(T) = 7$. Given a query $Q(v_2, v_7)$, $d(v_2, v_7) = \min_{i\in X(v_{13}).pos}\{X(v_2).dis[i] + X(v_7).dis[i]\} = 7$, where $X(v_{13})$ is the LCA of $X(v_2)$ and $X(v_7)$.*

3.3 Partition-based Solution

The *Partitioned Shortest Path (PSP)* index refers to the shortest path index that adopts graph partitioning [64], which has been widely studied to improve indexing time [30, 35, 36], index size [31, 37, 50], or index update [64, 72]. In general, they first decompose a road network G into multiple subgraphs $\{G_i|1 \le i \le k\}$ by graph partitioning methods [12, 26] such that $\bigcup_{i\in[1,k]} V(G_i) = V, V(G_i) \cap V(G_j) = \phi(\forall i \ne j, i,j \in [1,k])$. An *overlay graph* $\tilde{G}$ is then built among the boundary vertices of all partitions to preserve the global shortest distances of G. For $\forall e \in E$, if its endpoints are boundary vertices from different partitions, we call it an *inter-edge*, *i.e.,* $e \in E_{inter}$. Otherwise, it is an *intra-edge*, *i.e.,* $e \in E_{intra}$. The PSP index L over G consists of the *partition indexes* $\{L_i\}$ for all subgraphs $\{G_i\}$ and the *overlay index* $\tilde{L}$ for $\tilde{G}$, *i.e.,* $L = \{L_i\} \cup \tilde{L}$. It relies on the *PSP strategy* for index construction, query, and update [64]. There are four PSP strategies [64, 72]: *pre-boundary*, *no-boundary*, *post-boundary*, and *cross-boundary*. The *pre-boundary strategy* [64] relies on *Dijkstra*'s searches for computing the global distances between all-pair boundary shortcuts $\{d(b_{i1}, b_{i2})|b_{i1}, b_{i2} \in B_i, b_{i1} \ne b_{i2}\}$ for each partition G_i, and thus is time-consuming when the boundary

[1]$v \in V$ of G is called *vertex* while tree node in T is called *node* in this paper.

vertex number is large. By contrast, *no-boundary* and *post-boundary strategies* [64] skip the slow pre-computation phase by directly leveraging $\{L_i\}$ for boundary shortcut computation, thus achieving much faster index construction and maintenance. Nevertheless, their query efficiency is limited on cross-partition queries due to the multi-hop distance concatenation on $\tilde{L}$ and $\{L_i\}$. To this end, [72] proposes a novel *cross-boundary strategy* that precomputes a global index L^* by concatenating $\tilde{L}$ and $\{L_i\}$, enabling efficient 2-hop querying on cross-partition queries.

To handle the HTSP problem, *PostMHL* [72] first leverages the vertex order generated by MDE to obtain a tree decomposition T, and then chooses a root vertex u for each partition G_i, regarding u and its descendants as the in-partition vertices while $X(u).N$ as the boundary vertices for G_i. As a result, it decomposes T into multiple partitions, making it possible to integrate *post-boundary* and *cross-boundary* PSP strategies into T to continuously enhance query efficiency during index maintenance. Nevertheless, it cannot adapt to various partition results (*e.g.,* it cannot leverage the natural cuts such as bridges on road networks as the boundaries) and identify the unaffected queries during index maintenance. Moreover, its query and index update efficiency are limited by the MDE-based ordering, resulting in unsatisfactory throughput.

Example 3. *Figure 2-(d) shows the TD-partitioning on T, where v_6, v_{14}, v_{11}, and v_{10} are chosen as the root vertices for G_1 to G_4, respectively, resulting in boundary vertex sets $B_1 = \{v_{16}\}$, $B_2 = \{v_{16}, v_{17}\}$, $B_3 = \{v_{13}, v_{17}\}$, $B_4 = \{v_{13}, v_{15}\}$. Figure 2-(e) presents the TD-partition result on the original network and the overlay graph $\tilde{G}$.*

4 UNAFFECTED QUERY IDENTIFICATION

In this Section, we provide a theoretical analysis of unaffected query identification and then propose a novel *two-phase index validation* algorithm to efficiently identify them.

4.1 Theoretical Analysis of Unaffected Queries

We use U to denote the set of edge updates. When U is small, it is more efficient to incrementally maintain the SP index L than reconstruct it from scratch. However, the maintenance of L could be time-consuming for large road networks, leading to long system-unavailable time and very low throughput. Fortunately, we observe that not all queries are affected by the update U. Therefore, we aim to identify the *unaffected queries* before or during the index maintenance, such that we could leverage the "old" index L for correct query processing during index maintenance. Given update U, the query $Q(s,t)$ is an *affected query* if the shortest distance from s to t changes. Otherwise, $Q(s,t)$ is an *unaffected query.* Then three questions arises: 1) How many queries are affected by U? 2) Is affected number bounded? 3) Can we determine whether a query is affected with the SP index L before maintenance? In the following, we analyze the decrease and increase update scenarios.

4.1.1 Decrease Update. We consider the simple scenario when only one edge $e(a,b)$ decreases from $w(a,b)$ to $w'(a,b)$ with $\delta_{a,b} = w(a,b) - w'(a,b)$, which has two general cases:

Case D1: If $e(a,b) \in sp(s,t)$ before update, then $Q(s,t)$ must be affected with new distance $d'(s,t) = d(s,t) - \delta_{a,b}$;

Case D2: If $e(a,b) \notin sp(s,t)$ before update. A straightforward solution is to directly check whether the updated $e'(a,b)$ constitutes new shortest paths. Therefore, we identify unaffected queries for an edge weight decrease update by Lemma 1. Note that proofs of Lemmas in this paper are elaborated in the Appendix of the full version [70] due to limited space.

Lemma 1. *Given decrease update of $e(a,b)$, suppose a is closer to s than b, $Q(s,t)$ is affected if $d(s,a) + w(a,b) + d(b,t) < d(s,t)$ or $d(s,a) + w'(a,b) + d(b,t) < d(s,t)$. Otherwise, it is unaffected.*

Although Lemma 1 can identify the affected queries of a decrease update, its computational cost is high for batch updates due to the multi-hop distance connections. We have the following lemma regarding the affected query number.

Lemma 2. *Given decrease update of $e(a,b)$, the number of affected queries range in $[\phi(a,b), |V|^2]$, where $\phi(a,b)$ is the number of shortest paths pass through edge $e(a,b)$ before update.*

Although the upper bound seems large, the actual number is limited in practice as the minimal edge weight is bounded by $w^*(a,b)$.

4.1.2 Increase Update. When an edge $e(a,b)$ increases from $w(a,b)$ to $w'(a,b)$ with $\delta_{a,b} = w'(a,b) - e(a,b)$, there are two general cases:

Case I1: $e(a,b) \in sp(s,t)$ before update. Although the increased edge lies in $sp(s,t)$, there may exist another shortest path from s to t that does not constitute $e(a,b)$. Therefore, $Q(s,t)$ is only affected if $e(a,b)$ lies in **all** shortest paths of $d(s,t)$, which cannot be determined only with L. Otherwise, it is unaffected;

Case I2: $e(a,b) \notin sp(s,t)$ before update. $Q(s,t)$ is unaffected since the increased edge does not constitute $sp'(s,t)$. Therefore, we can safely identify this kind of unaffected queries by checking whether $e(a,b)$ lies in $sp(s,t)$ with Lemma 3. The affected query number is shown in Lemma 4.

Lemma 3. *$Q(s,t)$ by the increase of $e(a,b)$ is unaffected if $d(s,a) + w(a,b) + d(b,t) \neq d(s,t)$ and $d(s,b) + w(a,b) + d(a,t) \neq d(s,t)$.*

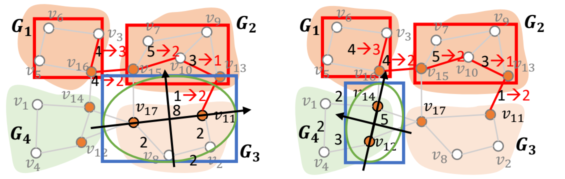
Lemma 4. *The number of queries affected by the increase of $e(a,b)$ is bounded by $\phi(a,b)$ and ranges in $[0, \phi(a,b)]$, where $\phi(a,b)$ is the number of shortest paths pass through edge $e(a,b)$ before update.*

From above analysis, we can identify all the unaffected queries by a single decrease with Lemma 1, but can only identify partial unaffected queries by a single increase with Lemma 3. Moreover, the above solution suffers from an additional multi-hop concatenation cost for each online query. Worse yet, the batch update will significantly deteriorate the performance of the above approach, making the cost of identifying unaffected queries unaffordable.

4.2 From Affected Area to Unaffected Queries

Since checking the unaffected queries online is laborious, can we simplify the problem to determine the *(un)affected area* of given updates before index maintenance, such that the queries in the *unaffected area* can be correctly answered by the "old" index? Specifically, we call $v \in V$ an *affected vertex* if its index $L(v)$ changes after the index update. As such, directly using the "old" $L(v)$ to answer the queries involving v will lead to an incorrect result. Besides, we call the area that consists of all affected vertices as an *affected area* Ω_A, while the remaining unaffected vertices form the *unaffected area* $\Omega_{\bar{A}}$. Because not all vertices are affected if U is small, then $\Omega_{\bar{A}}$ could help to determine the unaffected queries as follows:

Lemma 5. *Given an update set U and its unaffected area $\Omega_{\bar{A}}$, any query $Q(s,t)$ is an unaffected query if $s \in \Omega_{\bar{A}}$ and $t \in \Omega_{\bar{A}}$.*



(a) MBR Overlap Check for G_3 (b) MBR Overlap Check for G_4

Figure 3: Illustration of MBR Overlap-based Validation

With Lemma 5, we can replace the previous online query check with a one-shot affected area identification. Although some queries involving affected vertices may also be unaffected queries, we do not identify them since it is challenging to do so. Now, the question becomes how to efficiently identify the unaffected area before the index maintenance. The exact solution is the time-consuming Lemma 1 and Lemma 3-based vertex check, which still cannot find all unaffected vertices for the increase update in *Case I1*. Then could we resort to an approximate unaffected areas by trading certain accuracy for efficiency? The answer is yes. Our idea is partitioning a graph G into multiple subgraphs $\{G_i\}$ and identify the unaffected area at the granularity of the partition level. We call G_i *inner-affected* if $\exists s, t \in G_i, Q(s,t)$ is an affected query. Otherwise, G_i is *inner-unaffected*. Besides, we call G_i *outer-affected* if $\exists b \in B_i$ whose overlay index $\tilde{L}(b)$ has changed, and *outer-unaffected* otherwise. Therefore, we call G_i an *unaffected partition* if it is both inner-unaffected and outer-unaffected. Otherwise, G_i is an *affected partition*. Based on these definitions, we propose Theorem 1 to identify the unaffected same-partition and cross-partition queries.

Theorem 1. ***(Unaffected Query Detection)*** *Given an inner-unaffected partition G_l, the same-partition query $Q(s,t), \forall s,t \in G_l$ is an unaffected query. Given two unaffected partitions G_i and G_j, the cross-partition query $Q(s,t), \forall s \in G_i, t \in G_j$ is an unaffected query.*

Proof. The same-partition query naturally holds as per the definition of inner-unaffected partition. Given a cross-partition query $Q(s,t)$, we take the concise form of $sp(s,t)$ by extracting only the boundary vertices as $sp_c = \langle s, b_1, ..., b_q, t\rangle$ $(b_l \in B, 1 \le l \le q)$, which can be divided into three parts: $sp(s,b_1)$, $sp(b_1,b_q)$, and $sp(b_q,t)$. $sp(s,b_1)$ and $sp(b_q,t)$ are unaffected since G_i and G_j are inner-unaffected partitions. $sp(b_1,b_q)$ is also unaffected since G_i and G_j are outer-unaffected. Hence, $Q(s,t)$ is unaffected. □

Theorem 1 provides a theoretical basis to convert unaffected partitions to unaffected queries, making it promising to detect unaffected queries for batch updates efficiently.

4.3 Two-Phase Index Validation

In this section, we propose a novel *Two-stage Index Validation (TIV)* algorithm to detect unaffected partitions efficiently in two stages: 1) *V-Stage 1* identifies partial inner-unaffected partitions via *minimum bounding rectangle (MBR)* overlap-based approximation before the index maintenance; 2) *V-Stage 2* finds all inner-unaffected partitions and out-unaffected partitions based on the label changes during the index maintenance.

V-Stage 1: MBR Overlap-based Validation. The intuition is that the partitions that are far away from the graph updates tend to be inner-unaffected. As shown in Figure 1, the updates in Region 1 can hardly affect Region 2. However, it is non-trivial to decide how far from Region 1 is enough to ensure that Region 2 is inner-unaffected. We introduce two important definitions as follows.

Algorithm 1: MBR Overlap-based Validation

```
   Input: Partitions $\{G_i|1 \le i \le k\}$ and a R-tree $T_R$ built on their MBRs, $L$, $U$
   Output: Inner-unaffected partitions $\Omega_{IU}$
1  $U_G \leftarrow \emptyset; I(G_i) \leftarrow false, \forall 1 \le i \le k;$ //$I(G_i)$ indicates if $G_i$ is inner-affected
2  for $e(a,b) \in U, a \in G_i, b \in G_j$ do
3      if $i = j$ or $G_j \notin U_G$ then $U_G \leftarrow U_G \cup G_i; I(G_i) \leftarrow true$;
4      else if $G_i \notin U_G$ and $G_j \in U_G$ then $U_G \leftarrow U_G \cup G_j; I(G_j) \leftarrow true$;
5  $Q.push(G_i), \forall G_i \in U_G$;                                   ▹ $Q$ is a queue
6  while $Q$ is not empty do
7      $G_j \leftarrow Q.front(), Q.pop()$;
8      for each neighbor partition $G_l$ of $G_j$ do
9          if $G_l \notin U_G$ and $G_l$ has not been checked then
10             $MBR_l \leftarrow$LRAMBRCompute($G_l$);       ▹ Compute MBR of $G_l$'s LRA
11             $S \leftarrow$BtreeSearch($MBR_l, T_R$); ▹ Compute overlapped partitions of $G_l$
12             if exists $G_i \in U_G, \forall G_i \in S$ then $I(G_j) \leftarrow true$; $Q.push(G_j)$;
13 return $\Omega_{IU} = \{G_i|I(G_i) = false, \forall 1 \le i \le k\}$;
14 Function LRAMBRCompute($G_l$):
15     for $\forall b_i, b_j \in B_l, b_i \ne b_j$ do
16         $D \leftarrow d_L(b_i,b_j) \cdot v_{max}$;               ▹ $v_{max}$ is the maximum speed limit
17         Compute the ellipse $\Omega_{max}(b_i,b_j)$ with foci $b_i$ and $b_j$, and major axis $D$;
18     Compute the MBR covering all ellipses $\{\Omega_{max}(b_i,b_j)|\forall b_i,b_j \in B_j\}$;
19     return MBR of $G_l$;
```

Definition 2 (Largest Reachable Area). *Given any OD pair (s,t), its Largest Reachable Area (LRA) is the largest spatial coverage area that $sp(s,t)$ may traverse, denoted as $\Omega_{max}(s,t)$.*

Definition 3 (Partition LRA). *We define the Partition LRA of subgraph G_i as the union of all LRAs of any two boundary vertices in B_i*, i.e., $\Omega_{max}(G_i) = \bigcup_{\forall b_1,b_2 \in B_i} \Omega_{max}(b_1,b_2)$.

Based on these definitions, we propose Lemma 6 to convert LRA into an ellipse and Theorem 2 to identify inner-unaffected partitions. The proof of Theorem 2 is in [70] due to limited space.

Lemma 6. *Given any OD pair (s,t), $\Omega_{max}(s,t)$ is an ellipse with foci s and t, and major axis $d(s,t) \cdot v_{max}$, where $d(s,t)$ is the minimum travel time from s to t, v_{max} is the maximum speed limit on G.*

Theorem 2. ***(MBR-based Detection)*** *When G_i has a batch update, G_j is inner-unaffected if $\Omega_{max}(G_j)$ does not overlap with the MBR of G_i before the update.*

The proposed *MBR overlap-based validation* algorithm is described in Algorithm 1. The MBRs of all partitions are obtained by the vertex's coordinates, and an *R-tree* [6] T_R is used to accelerate the overlap search. We first obtain the initial affected partitions U_G (Lines 1-4) and initialize a queue Q with them (Line 5). Next, we identify other affected partitions in BFS (Lines 6-12). For each G_j popped from Q, we check whether its neighboring partitions are affected. For each neighbor G_l, we compute the MBR of its partition LRA (MBR_l) (Lines 14-19). If MBR_l overlaps with any initial affected partitions in U_G, G_l is inner-affected and is pushed into Q (Line 12). The time complexity of Algorithm 1 is $O(k \cdot B_{max}^2 \cdot \tau)$, where $B_{max} = \max_{i \in [1,k]} |B_i|$ and τ is the time to compute the ellipse.

Example 3. *As shown in Figure 3, U_G contains G_1 and G_2. We first check their neighbor G_3's LRA (green ellipse with foci v_{17} and v_{11}) and corresponding MBR (blue rectangle), as shown in (a). G_3 is inner-affected as its MBR overlaps with G_2. By contrast, in (b), G_4 is inner-unaffected since its MBR does not overlap with either G_1 or G_2.*

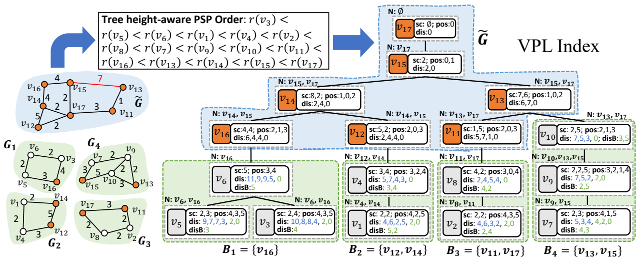

Tree height-aware PSP Order: $r(v_3) < r(v_5) < r(v_6) < r(v_1) < r(v_4) < r(v_2) < r(v_8) < r(v_7) < r(v_9) < r(v_{10}) < r(v_{11}) < r(v_{16}) < r(v_{13}) < r(v_{14}) < r(v_{15}) < r(v_{17})$
$\tilde{G}$
VPL Index
G_1
G_2
G_3
G_4
$B_1 = \{v_{16}\}$
$B_2 = \{v_{12}, v_{14}\}$
$B_3 = \{v_{11}, v_{17}\}$
$B_4 = \{v_{13}, v_{15}\}$

Figure 4: VPL Index Structure

V-Stage 2: Label Change-based Validation. While *V-Stage 1* can identify some inner-unaffected partitions, it does not capture all, nor does it identify outer-unaffected partitions. Therefore, we propose *Label Change-based Validation* in *V-Stage 2*. Its key idea is to leverage partially-updated index and label change information collected during the index maintenance to check whether a partition is inner-affected or outer-affected. In particular, we leverage Lemma 7 to identify all inner-unaffected partitions and determine outer-unaffected and unaffected partitions based on their definitions, ensuring the detection of all unaffected partitions. Detailed integration with index maintenance is discussed in Section 5.2.

Lemma 7. *G_i is inner-unaffected if there is no update in G_i and all distances among its boundary vertices $d(b_1, b_2), \forall b_1, b_2 \in B_i$ remain unchanged after the update. Otherwise, G_i is inner-affected.*

5 VALIDATION-BASED PSP FRAMEWORK

In this Section, we introduce our *VPSP* framework, which exploits the PSP index availability spatially and temporally. In particular, we first propose the *Validation-based Partitioned Labeling (VPL)* as the underlying PSP index and then introduce the *VPSP* framework with its core *rotating update schedule*.

5.1 Validation-based Partitioned Labeling

We first propose a novel *VPL* as the underlying PSP index for our *VPSP* framework. The key difference of *VPL* with the existing PSP indexes is that it supports index validation to identify the unaffected queries to make the best use of the index during maintenance. Moreover, among the existing TD-based PSP indexes [35, 36, 64, 65, 72], *VPL* is the first to build a TD T from graph partitions while integrating different PSP strategies into a single T. Therefore, *VPL* not only supports various graph partition result but also has excellent query and update efficiency. Note that although *PostMHL* [72] also integrates different PSP strategies into a single TD, it achieves this through *TD-partitioning* and thus cannot utilize good graph partition results that comply with query clusters. Moreover, its query and index update efficiency are confined by MDE-based ordering.

We next introduce the *VPL* index structure. Specifically, we first classify all vertices into two types: *boundary* vertices ($v \in \tilde{G}$) and *non-boundary* vertices ($v \in G \setminus \tilde{G}$). For boundary vertices $\forall v \in \tilde{G}$, its tree node $X(v)$ has three types of labels: *position array* $X(v).pos$, *distance array* $X(v).dis$, and *shortcut array* $X(v).sc$ storing the distance from v to vertices in $X(v).N$. We denote the labels of all boundary vertices as *overlay index* $\tilde{L}$. For non-boundary vertices $v \in G_i \setminus B_i$, its tree node $X(v)$ additionally contains a *boundary array* $X(v).disB$, which stores the distances from v to all boundary vertices $u \in B_i$. Since *VPL* adopts post-boundary and cross-boundary PSP strategies, its partition index consists of post-boundary index $\{L_i\}$ and cross-boundary index L^*. In particular, for each tree node $X(v)$, we classify its ancestors into two types: *overlay ancestors* (denoted as $X(v).\tilde{A}$) that belong to $\tilde{G}$ and *non-overlay ancestors* (denoted as $X(v).\bar{A}$) otherwise. The elements in the *distance array* are then divided into two parts: the entries to overlay ancestors and those to non-overlay ancestors. The post-boundary index $\{L_i\}$ consists of the entries to $X(v).\bar{A}$ and $X(v).disB$, while the cross-boundary index L^* is composed of the distance arrays of all vertices.

To build the *VPL* index, we first adopt *PUNCH* [12] as the graph partitioning method to divide G into multiple subgraphs $\{G_i | 1 \leq i \leq k\}$ because it generates the minimal boundary number on road networks [64] and its partition result complies well with the query clusters. Given the partition result, we first derive its order r via *tree height-aware PSP ordering* (will be introduced in Section 6.2). A tree decomposition T and the *shortcut arrays* of its tree nodes are then built based on r. For the label construction, we first build the overlay index from the root of T and then construct the labels for all partitions' non-boundary vertices in parallel. For each tree node $X(v)$, its distance array is computed by using $X(v).N$ as the vertex separator while the boundary array is computed by inheriting corresponding label entries in $X(v).dis$. We provide the pseudo-code of *VPL* construction in the full version [70] due to limited space.

Example 4. *Figure 4 shows the VPL index structure. The order r is obtained via tree height-aware PSP ordering. The overlay index that consists of boundary vertices is in a blue box, and the partition index is in green boxes. Each boundary vertex (*e.g.*, $X(v_{13})$) only has a shortcut array, a position array, and a distance array. The non-boundary vertex has an additional boundary array,* e.g., *$X(v_7).disB = \{4, 3\}$ storing the distances to its boundaries v_{13} and v_{15}. The post-boundary index of non-boundary vertices is presented in green, while the entries only belonging to the cross-boundary index are shown in blue.*

5.2 VPSP Framework Overview

Based on *VPL*, we next provide an overview of the *VPSP* framework. As shown in Figure 5, *VPSP* comprises three main modules: *index update*, *index validation*, and *query processing*. The index update module handles the graph updates in batches through a novel *rotating update schedule* and *dynamic algorithms* (will be detailed in Section 5.3 and Section 6.1, respectively). Based on the information from the index update module, the index validation module continuously identifies unaffected partitions via the *TIV* algorithm introduced in Section 4.3. This information is then relayed to the query processing module, which assesses whether a query is affected and applies appropriate processing methods accordingly. We next briefly introduce these modules.

Index Update Module. Given the rotating updates U produced by the rotating update schedule as input, *VPSP* maintains the *VPL* in five stages (*U1-U5*). We briefly introduce them, while the detailed dynamic algorithms of *U2-U5* will be elaborated in Section 6.1.

U1: On-spot Edge Update. It modifies edge weights on the original road networks, which takes only constant time and is negligible. The index-free *BiDijkstra's search (Q1)* [41] is applicable after *U1*.

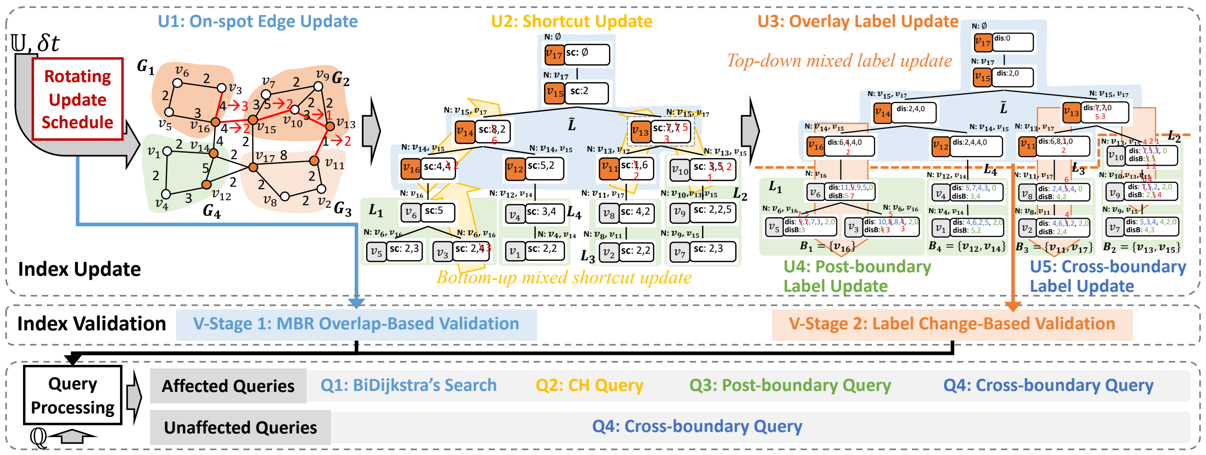


Figure 5: VPSP Framework and Workflow

U2: Shortcut Update. This stage maintains the affected shortcut arrays in a *bottom-up manner*. We first categorize the updates U into two types: U_{inter} of the inter-edge updates, and U_{intra} of the intra-edge updates. If U_{intra} is not empty, we identify the affected partitions and maintain their affected shortcut arrays in parallel. Meanwhile, the affected boundary shortcuts will be added to U_{inter}. The shortcut array of the overlay index is maintained based on U_{inter}, enabling the *CH query (Q2)* [17, 72] after *U2*. We also record the affected tree nodes during this stage.

U3: Overlay Label Update. Sourcing from the highest affected overlay tree nodes identified in *U2*, this stage maintains the distance arrays of overlay vertices (*i.e.,* overlay labels) in a *top-down manner*. Meanwhile, we also record the newly affected non-overlay tree nodes to ensure correct label update propagation. We still use *CH query (Q2)* to handle queries after this stage.

U4: Post-boundary Label Update. Based on the affected tree nodes identified in *U2* and *U3*, this stage parallelly maintains the post-boundary index of non-boundary vertices (*i.e.,* their boundary arrays and distance entries to the non-overlay ancestors) for all affected partitions in a *top-down manner*. After this, a more efficient *post-boundary query (Q3)* is released for query processing.

U5: Cross-boundary Label Update. This stage also maintains the cross-boundary index of non-boundary vertices for all affected partitions in parallel in a *top-down manner*, which enables the fastest *cross-boundary query (Q4)* processing. It is worth noting that *U4* and *U5* can be performed in parallel as they only rely on affected tree nodes identified in *U2* and *U3* and thus are independent of each other [72]. Nevertheless, *U5* typically has a longer index maintenance than *U4* due to the large overlay tree height.

Index Validation Module. This module leverage the *TIV* algorithm introduced in Section 4.3 to detect the unaffected partition. *V-Stage 1* uses rotating updates U as input to quickly determine the unaffected partition through *MBR overlap-based validation*. While *V-Stage 2* has to wait for the finish of U3 since it relies on the updated overlay index and overlay label change information to determine the unaffected partitions. In particular, for each partition G_i, we compute the all-pair distances among its boundary vertices $\{d(b_1, b_2)|\forall b_1, b_2 \in B_i\}$ and leverage Lemma 7 to determine whether G_i is inner-unaffected. Besides, we say G_i is outer-unaffected if all of its boundary vertices' overlay indexes (*i.e.,* $\tilde{L}(b), \forall b \in B_i$) do not change during *U3*. Note that the all-pair boundary computation can be conducted in parallel among different partitions and it is also required in the post-boundary index update. Therefore, *V-Stage 2* does not increase computation workload.

Query Processing Module. Given the unaffected partitions, the query processing module leverages Theorem 1 to determine whether a query $Q(s,t)$ is affected or not. If it is affected, there are four query stages *Q1-Q4* to handle it, and we always adopt the fastest one according to the current update stages. For the unaffected ones, we leverage the "old" index to process it with the fastest cross-boundary query (*Q4*). In particular, *Q1* is the *BiDijkstra' search* and *Q2* is a CH query on the shortcut arrays of *VPL*. These two query methods are essentially search-based and have limited query efficiency. In contrast, the *post-boundary query (Q3)* following *U4* significantly enhances speed by substituting graph search with hop-based querying, which processes the queries in two Cases.

Case 1: Same-Partition: $\forall s, t \in G_i, Q(s,t) = d_{L_i}(s,t)$;
Case 2: Cross-Partition: $\forall s \in G_i, t \in G_j, i \neq j, Q(s,t) =$

$$\begin{cases} d_{\tilde{L}}(s,t) & s,t \in B \\ \min\limits_{b_q \in B_j} \{d_{\tilde{L}}(s,b_q) + d_{L_j}(b_q,t)\} & s \in B, t \notin B \\ \min\limits_{b_p \in B_i} \{d_{L_i}(s,b_p) + d_{\tilde{L}}(b_p,t)\} & s \notin B, t \in B \\ \min\limits_{b_p \in B_i, b_q \in B_j} \{d_{L_i}(s,b_p) + d_{\tilde{L}}(b_p,b_q) + d_{L_j}(b_q,t)\} & s \notin B, t \notin B \end{cases}$$

Post-boundary query reduces the time complexity to $O(\max_{i\in[1,k]} \{|B_i| \cdot w_i + |B_i|^2 \cdot \tilde{w}\})$, where w_i and $\tilde{w}$ are the treewidth of partition and overlay graph, respectively. Nonetheless, the *cross-boundary query (Q4)* over L^* further reduces the time complexity to $O(w)$ by using the same query processing as *H2H*, achieving the fastest query efficiency, where w is the treewidth of L.

5.3 Rotating Update Schedule

Although the *TIV* algorithm can efficiently identify all unaffected partitions, the number of unaffected partitions and queries may be limited if the updates of all partitions come to the system simultaneously, as most partitions may undergo updates during the interval δt. To mitigate this issue, we propose the *rotating update*

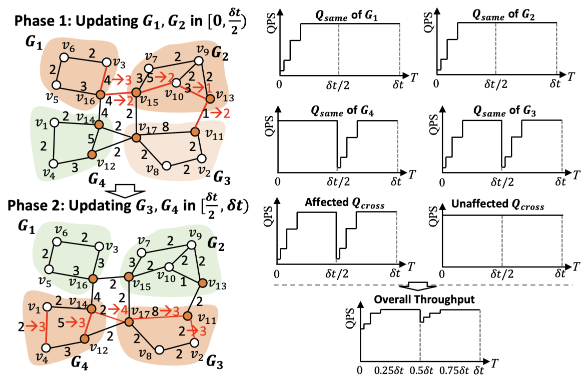

Phase 1: Updating G_1, G_2 in $[0, \frac{\delta t}{2})$
Phase 2: Updating G_3, G_4 in $[\frac{\delta t}{2}, \delta t)$
Q_{same} of G_1
Q_{same} of G_2
Q_{same} of G_4
Q_{same} of G_3
Affected Q_{cross}
Unaffected Q_{cross}
Overall Throughput
QPS
$\delta t/2$
δt
T
$0.25\delta t$ $0.5\delta t$ $0.75\delta t$
(a) Rotating Update in $[0, \delta t)$ (b) Throughput Composition

Figure 6: Illustration of Rotating Update Schedule

schedule that re-organizes the arrival of batch updates of different partitions such that all the partitions are still updated every δt time while allowing more unaffected queries to be processed with the index. Specifically, given a set of k partition $\{G_i\}$, we divide the original δt into k phases and update each G_i at $(\delta t \cdot i)/k$ time slot. From each G_i's perspective, its update frequency is still δt. Since only one partition updates at a time, more distant partitions can remain unaffected, allowing their queries to be processed efficiently using the index. This approach maximizes the unaffected queries and prevents system-level throughput drops due to global index maintenance. However, it also diminishes the parallelization among affected partitions since more partitions are maintained sequentially. Besides, if the number of partitions k is large, each partition maintenance might not finish within $\delta t/k$. To balance the partition update time and the number of unaffected queries, we propose a *query-aware partition clustering* to merge all partitions into $\mathcal{K}$ regions ($1 \le \mathcal{K} \le k$). Based on these regions, we divide the original δt into $\mathcal{K}$ phases and process each region with the same method.

We introduce the *query-aware partition clustering* as follows. It obtains the regions by leveraging the historical cross-partition queries in three Steps. *Step 1: Query-aware Graph Construction.* We build a query-aware graph G_p among all partitions by regarding each partition as a vertex (*i.e.*, $V(G_p) = \{v_{G_1}, ..., v_{G_k}\}$ where v_{G_1} represents the partition G_1) and adding an edge for any two partitions if at least one cross-partition query exists between them (*i.e.*, $E(G_p) = \{e(v_{G_i}, v_{G_j}) | \exists Q(s,t), s \in G_i, t \in G_j\}$). The weight of $e(v_{G_i}, v_{G_j})$ equals the cross-partition query number among them, which reflects their intimacy. *Step 2: Greedy Clustering.* We iteratively process each edge in $E(G_p)$ in descending order of edge weight, because higher edge weights indicate that more cross-partition queries will not be affected if the two partitions are in one region. For each $e(v_{G_1}, v_{G_2})$ being processed, we check whether its endpoints have been assigned to a region. If they are both unassigned, we combine them into a new region. Otherwise, if only one partition has been assigned (*e.g.*, $G_1 \in R_i$), we add the other unassigned partition G_2 into R_i. After processing all edges, any remaining partitions are added to neighboring regions with the fewest partition number. *Step 3: Balance-oriented Assembling.* Given l regions obtained in Step 2, we next assemble them in a balance-oriented manner. We start from the region with a minimal partition number (*e.g.*, R_i), merging R_i with the neighboring region with the smallest partition number. We iteratively assemble these regions until the region number is reduced to $\mathcal{K}$.

Algorithm 2: DVPL: Mixed Shortcut Maintenance

```
   Input: $G$ and its mixed updates $U, L$
   Output: Shortcut Updated $L$
1  $Q.insert(a), S[a].insert(b), \forall e(a,b) \in U$; //$Q$ is a min-heap set, $S[a]$ is a set for $a$
2  while $Q$ is not empty do
3      $a \leftarrow Q.begin(), Q.erase(a)$; //$a$ has the minimum vertex order in $Q$
4      for each $b \in S[a]$ do
5          Compute new shortcut $d_{sc}$ and $\phi_{sc}$ by minimum weight property [59]
6          if $d_{sc} > sc(a,b)$ then $Cw \leftarrow sc(a,b)$; //increase update
7          else if $d_{sc} < sc(a,b)$ then $Cw \leftarrow d_{sc}$; //decrease update
8          $sc(a,b) \leftarrow d_{sc}$; $\phi_{sc}(a,b) \leftarrow \phi_{sc}$; SCUpdatePropagate($a, b, Cw$);

9  Function SCUpdatePropagate($a, b, Cw$):
10     if $sc(a,b)$ is decrease update then
11         DecLabelCheck($a, b, Cw$);    ▷ Identify affected labels, see Appendix of [70]
12         for each $u \in X(a).N$ do
13             if $u \in X(b).N$ then DecSCCheck($b, u, a, Cw + sc(a,u)$);
14             else if $b \in X(u).N$ then DecSCCheck($u, b, a, Cw + sc(a,u)$);

15     else if $sc(a,b)$ is increase update then
16         if $X(a).FN[b] = true$ then IncLabelCheck($a, b, Cw$);
17         for each $u \in X(a).N$ do
18             if $u \in X(b).N$ then IncSCCheck($b, u, Cw + sc(a,u)$);
19             else if $b \in X(u).N$ then IncSCCheck($u, b, Cw + sc(a,u)$);

20 Function DecSCCheck($u, b, a, dis$):
21     if $dis = sc(u,b)$ and $u \notin S[a]$ then $\phi_{sc}(u,b) \leftarrow \phi_{sc}(u,b) + 1$;
22     else if $dis < sc(u,b)$ then $Q.insert(u), S[u].insert(b)$;

23 Function IncSCCheck($u, b, dis$):
24     if $dis = sc(u,b)$ then
25         $\phi_{sc}(u,b) \leftarrow \phi_{sc}(u,b) - 1$;
26         if $\phi_{sc}(u,b) < 1$ then $Q.insert(u), S[u].insert(b)$;
```

Example 5. *Figure 6 shows the rotating update schedule on G with two regions (region 1 $\{G_1, G_2\}$ and region 2 $\{G_3, G_4\}$). The update during δt has Phase 1 for G_1G_2 in $[0, \delta t/2)$ and phase 2 for G_3G_4 in $[\delta t/2, \delta t)$. G_3 is identified as an affected partition during phase 1, while G_1 and G_2 are both unaffected during phase 2. The throughput composition is shown in (b), where G_1, G_2, G_4 are only affected in their corresponding phases, while G_3 is affected in both phases. Although affected Q_{cross} suffers from QPS loss due to the update, the unaffected Q_{cross} remains high query efficiency during δt.*

6 UPDATE ALGORITHM AND OPTIMIZATION

In this Section, we propose a dynamic algorithm to support the mixed update for *VPL* and further optimize throughput via a novel *tree height-aware PSP ordering*.

6.1 Dynamic Algorithm for VPL

In practice, each batch of updates typically contains both decrease and increase updates. However, to the best of our knowledge, all existing solutions [18, 53, 59, 61, 66, 71] deal with decrease and increase updates separately due to their diversified maintenance mechanisms. To handle the mixed update, one optimization is to process all decreases first to reduce the workload of increase updates [62]. However, it still nearly doubles the maintenance time and, more importantly, it does not work in *VPL* index due to the following two reasons: ***R1***: *VPSP* requires the accurate overlay label after *U3*, resulting in updated shortcut arrays with both decreased and increased entries. These various update sources would invalidate the update propagation of each other when applying the existing

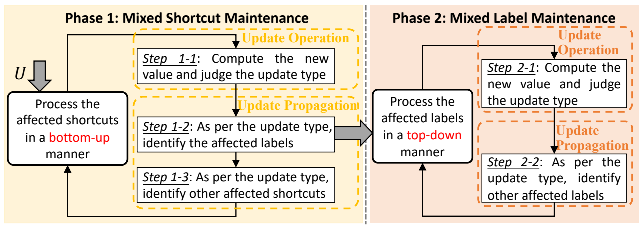

Phase 1: Mixed Shortcut Maintenance
U
Process the affected shortcuts in a bottom-up manner
Update Operation
Step 1-1: Compute the new value and judge the update type
Update Propagation
Step 1-2: As per the update type, identify the affected labels
Step 1-3: As per the update type, identify other affected shortcuts
Phase 2: Mixed Label Maintenance
Process the affected labels in a top-down manner
Update Operation
Step 2-1: Compute the new value and judge the update type
Update Propagation
Step 2-2: As per the update type, identify other affected labels

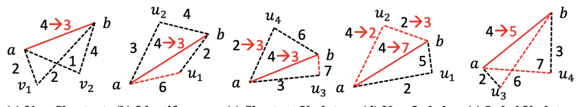
Figure 7: DVPL Framework

solutions (*e.g., DH2H* [59]). ***R2***: the decrease updates do not maintain the important information (*i.e., supportive number* $\phi_{sc}(a,b)$ and $\phi_d(u,v)$, which records the number of contracted vertices that supports the shortcut $sc(a,b)$ and also the distance label $L(u,v)$ hold) used in the increase updates.

Example 6. *Figure 8-(d) shows a mixed label update computation. Suppose $sc(a,b)$ and $L(b,u_2)$ increases to 7 and 3, while $L(a,u_2)$ decreases to 2. If we first conduct the decrease update and then the increase as per [59], $L(a,b)$ would remain 4 while $\phi_d(a,b)$ would increase to 2 during the update propagation of $L(a,u_2)$ and decrease to 1 during the propagation of $L(b,u_2)$ due to R2, which is incorrect.*

To handle the mixed updates, we propose a *Dynamic VPL (DVPL)* algorithm for our *VPL* index. The key idea is to decouple the update operation and update propagation, identifying affected shortcuts (or distance labels) first and maintaining them in a lazy update manner. The *DVPL* algorithm consists of two phases, as shown in Figure 7. Phase 1 is mixed shortcut maintenance (used in *U2* of Figure 5), which starts from the edge update set U and handles the update in a *bottom-up manner* with three steps. For each affected shortcut $sc(a,b), r(a) < r(b)$, *Step 1-1* computes its new value and judges its update type by comparing the new value and old one. Based on the update type, we propagate the update in two steps: *Step 1-2* identifies affected labels of a, which are the input sources of Phase 2; *Step 1-3* then identifies other affected shortcuts via the accessory neighbors of a. Phase 2 is mixed label maintenance (used in *U3-U5* of Figure 5), which starts from the affected labels identified Phase 1 and handles the update in a *top-down manner* with two steps. For each affected labels $L(a,b), r(a) < r(b)$, similar to shortcut maintenance, *Step 2-1* first computes its new value and judges its update type. *Step 2-2* further detects other affected labels during the update propagation. We next use the non-partitioned L to demonstrate the key idea of *DVPL* while it is easy to extend it to a partitioned one with parallelization. We present the pseudo-code of shortcut maintenance in Algorithm 2, where a min-heap set Q and a min-heap set $S(a)$ are used to record affected tree nodes (*e.g., a*) and the neighbors of *a*' affected shortcuts, ensuring the updates are processed in a bottom-up manner. For each affected shortcut, we use the procedure introduced in Phase 1 to process the update. We dynamically maintain the $\phi_{sc}(u,b)$ in the decrease update as per Theorem 3. Besides, for each shortcut $sc(a,b), \forall a \in G, b \in X(a).N$, we additionally maintain a boolean value $X(a).FN[b]$ to indicate whether the label $L(a,b)$ equals $sc(a,b)$, which helps to significantly prune unnecessary search space during shortcut increase update.

Theorem 3. *Given a decrease update of $sc(a,b)$, we only need to increase $\phi_{sc}(u,b)$ by 1 if $sc'(a,b) + sc(a,u) = sc(u,b)$ and $u \notin S[a]$.*



(a) New Shortcut Computation **(b) Identify Affected Labels** **(c) Shortcut Update Propagation** **(d) New Label Computation** **(e) Label Update Propagation**

Figure 8: Illustration of Index Maintenance

Algorithm 3: DVPL: Mixed Label Maintenance

```
Input: Root tree node set $V_{root}$, Shortcut updated $L$
Output: Label updated $L$
for all $X(a) \in T(u), \forall u \in V_{root}$ in a top-down manner do
    if $X(a).ifU = true$ then
        for each $b \in X(a).A$ and $C(a,b) \neq 0$ do
            Compute new distance $d$ and $\phi_d$ by minimum distance property [59];
            if $d > L(a,b)$ then $Cw \leftarrow L(a,b)$; //increase update
            else if $d < L(a,b)$ then $Cw \leftarrow d$; //decrease update
            $L(a,b) \leftarrow d, \phi_d(a,b) \leftarrow \phi_d$; DisUpdatePropagate($a, b, Cw$);

Function DisUpdatePropagate($a, b, Cw$):
    if $L(a,b)$ is decrease update then
        for each $u \in X(a).D$ and $X(u).FN[a] = true$ do
            DecUpdateCheck($u, b, a, Cw + L(u,a)$);
        for each $u \in X(b).D$ and $X(u).FN[b] = true$ and $a \in X(u).A$ do
            DecUpdateCheck($u, a, b, Cw + L(u,b)$);
    else if $L(a,b)$ is increase update then
        for each $u \in X(a).D$ and $X(u).FN[a] = true$ do
            IncUpdateCheck($u, b, Cw + L(u,a)$);
        for each $u \in X(b).D$ and $X(u).FN[b] = true$ and $a \in X(u).A$ do
            IncUpdateCheck($u, a, Cw + L(u,b)$);

Function DecUpdateCheck($u, a, b, dis$):
    if $dis = L(u,a)$ and $C(u,b) = 0$ then $\phi_d(u,a) \leftarrow \phi_d(u,a) + 1$;
    else if $dis < L(u,a)$ then $C(u,a) \leftarrow -1, X(u).ifU \leftarrow true$;

Function IncUpdateCheck($u, a, dis$):
    if $dis = L(u,a)$ then
        $\phi_d(u,a) \leftarrow \phi_d(u,a) - 1$;
        if $\phi_d(u,a) < 1$ then $C(u,a) = 1, X(u).ifU \leftarrow true$;
```

Proof. $u \notin S[a]$ means that $sc(a,u)$ is unaffected. There are three cases for $sc(a,u)$.

Case 1: $sc(a,u)$ decreases to $sc'(a,u)$. We have $sc'(a,b)+sc'(a,u) < sc(u,b)$ and $sc(u,b)$ needs to be maintained. Therefore, there is no need to increase $\phi_{sc}(u,b)$.

Case 2: $sc(a,u)$ increases to $sc'(a,u)$. We have $sc'(a,b)+sc'(a,u) \neq sc(u,b)$, and thus there is no need to increase $\phi_{sc}(u,b)$.

Case 3: $sc(a,u)$ remains unaffected. We have $sc'(a,b) + sc(a,u) = sc(u,b)$, and thus it is necessary to increase $\phi_{sc}(u,b)$ by 1. Otherwise, the increase update of $sc(u,b)$ may be incorrect in the next batch due to the wrong $\phi_{sc}(u,b)$. □

Example 7. *Figure 8 (a)-(c) illustrates three key steps in shortcut update. Suppose $sc(a,b)$ has two supportive vertices v_1, v_2 and the edge $e(a,b)$ decreases from 4 to 3, we have new $d_{sc} = 3$ and $\phi_{sc} = 1$ since only edge $e(a,b)$ makes new $sc(a,b)$ holds after update. In Figure 8-(b), $L(a,u_1)$ is affected since $Cw + L(u_1,b) = 3 + 2 = 5 < L(a,u_1)$ while $L(a,u_2)$ is unaffected. Figure 8-(c) presents the update propagation of $sc(a,b)$, $sc(u_3,b)$ is affected since $sc'(a,b) + sc(a,u_3) = 3 + 3 = 6 < sc(u_3,b)$ while we do not check $sc(b,u_4)$ as $u_4 \in S[a]$.*

Algorithm 3 presents the label maintenance of *DVPL*. Different from shortcut maintenance and existing solutions [59, 66], we do not use a max-heap set (or priority queue) to ensure the *top-down update manner*. This is because for the batch update scenario, the affected labels are tremendous, and using the priority queue can lead to significant computation. By contrast, sourcing from the

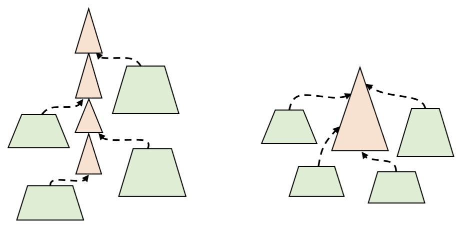

(a) Naive PSP Ordering (b) Tree Height-aware PSP Ordering

Figure 9: Illustration of PSP Ordering

affected tree nodes of the highest order ($\forall u \in V_{root}$), we check all descendants of $X(u)$, *i.e.*, $X(a) \in T(u)$, and only maintain $X(a)$ if $X(a)$ is affected, *i.e.*, $X(a).ifU = true$ (Lines 1-2). For each affected $X(a)$, we check all its ancestors and only maintain the label $L(a,b)$ if it is affected (*i.e.*, $C(a,b) \neq 0$). Note that to facilitate the update propagation, we also precompute the *downward neighbors* $X(v).D$ for all $v \in G$ in index construction, where we have $v \in X(u).N, \forall u \in X(v).D$. We also propose Theorem 4 to maintain $\phi_d(u,b)$ during the decrease update of $L(a,b)$.

Theorem 4. *Given a decrease update of* $L(a,b)$, *we only need to increase* $\phi_d(u,a)$ *by 1 if* $L'(a,b) + L(u,b) = L(u,a)$ *and* $C(u,b) = 0$.

Theorem 5. *The time complexity of Algorithm 2 and Algorithm 3 are* $O(\Delta_{sc}^{T} \cdot \log(\Delta_{sc}^{T}) + \Delta_{sc} \cdot (w + \log(w) + h))$ *and* $O(\Delta_d \cdot (w + deg_{\mathcal{G}^-}^{max}) + \Delta_n)$, *respectively, where* Δ_{sc}^{T} *and* Δ_{sc} *are the number of affected tree nodes and shortcuts,* w *is treewidth,* $deg_{\mathcal{G}^-}^{max}$ *is the maximal downward-neighbor degree (downward neighbors are the neighbors with lower vertex order) in* $\mathcal{G}$, Δ_d *is affected label number,* Δ_n *is tree node number of the subtree rooted at the highest affected tree node.*

Example 8. *In Figure 8-(d), we compute* $L(a,b)$ *via* $X(a).N = \{u_1, b, u_2\}$, *obtaining new* $L'(a,b) = 5$ *and* $\phi_d(a,b) = 1$ *since only* u_2 *make* $L'(a,b)$ *holds. Figure 8-(e) shows the update propagation,* $L(u_3,b)$ *and* $L(u_4,a)$ *are affected due to the increase of* $L(a,b)$, *where* u_3 *and* u_4 *are the downward neighbors of* a *and* b, *respectively.*

6.2 PSP Ordering Optimization

The vertex order is crucial for index structure and performance, but finding an optimal order is NP-hard [10]. For PSP index, its order should follow the *boundary-first property* [72] (boundary vertices have higher orders than inner for each partition) to leverage the *cross-boundary strategy* [72]. The existing PSP ordering [37, 72] generally computes r in three steps: 1) build a sketch graph among all partitions by regarding each partition as a vertex and applying MDE on this sketch graph to obtain the relative order r_{sketch} among partitions; 2) get the inner-partition orders are obtained through MDE; 3) get the final order according to r_{sketch} and fulfill *boundary-first property*, ensuring the boundaries of the same partition have consecutive order. We call this *naive PSP ordering* as it does not optimize the ordering among boundary vertices and thus its overlay tree looks like a pole, resulting in an lower cross-boundary query efficiency than MDE-based *DH2H* [72].

Observed from *PostMHL*'s tree decomposition to partition result, we find that the above partition-relative order is not necessary. Therefore, we propose a new *Tree Height-aware PSP ordering (THP-ordering)* to obtain an order from partition results with smaller tree height and treewidth in three steps. <u>*Step 1*</u>: We build an ordering $r_{G_i \setminus B_i}$ among the non-boundary vertices in G_i for all partitions by contracting them with MDE. During the MDE process, we dynamically maintain $h^-(T_u)$ for all $u \in G$, where $h^-(T_u)$ is the *subtree height lowerbound* of the subtree rooted at u. Specifically, we first initiate $h^-(T_u) = 1, \forall u \in G$. For each vertex $v \in G$ being contracted, we refresh the *subtree height lower bound* of its neighbors with $h^-(T_u) = \max\{h^-(T_u), h^-(T_v) + 1\}, \forall u \in X(v).N$ since any accessory neighbor $u \in X(v).N$ is an ancestor of v and the subtree rooted at u must have a height no less than $h^-(T_v) + 1$. We obtain a contracted graph G_c after this step. <u>*Step 2*</u>: We generate a vertex order $r_{\tilde{G}}$ for all boundary vertices on G_c with MDE. The intuition is to minimize the overlay tree height during the contraction. In particular, we use the degree as the primary criterion and always process the vertex with the lowest degree first. However, we adopt the *subtree height lower bound* (instead of the vertex ID used in MDE) as the second criterion when the degrees of different vertices are the same, *i.e.*, the vertex u with a lower $h^-(T_u)$ is processed first. <u>*Step 3*</u>: We aggregate $r_{\tilde{G}}$ and $\{r_{G_i}\}$ by setting the inner vertices with lower order than $r_{\tilde{G}}$ to fulfill the *boundary-first property* [72]. Although the proposed PSP ordering does not force the boundary vertices of the same partition to have consecutive order, we prove it still correctly processes the queries in Lemma 8.

Table 1: Real-world Datasets

Name	Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert Q\rvert$	$\lvert\bar{U}\rvert$	$\lvert U\rvert_{max}$	k
BJ	Beijing	296,381	774,660	493,780	472,932	4,181	7,340	32
GD	Guangdong	938,957	2,452,156	809,435	762,110	7,375	13,719	64
SC	South China	1,326,091	3,388,770	892,129	849,838	9,179	16,915	128
WC	West China	1,886,336	4,603,980	864,658	683,586	5,929	11,244	256
EC	East China	3,008,173	7,793,146	2,034,716	1,555,335	14,706	28,487	64
CN	China	9,472,486	24,058,928	4,410,824	4,410,824	40,441	75,249	128

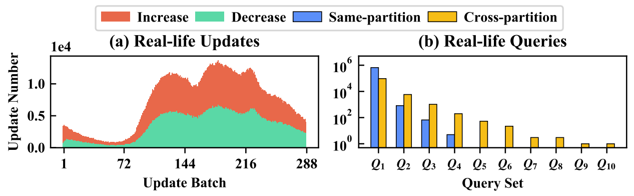


Figure 10: Real-life Updates and Queries of GD

Lemma 8. *The tree height-aware PSP ordering fulfills the boundary-first property, and VPL adopting it has correct query processing.*

Example 9. *Figure 9-(a) shows naive PSP ordering with a large tree height as its boundary vertices pile up to form a pole-like tree. While THP-ordering in Figure 9-(b) reduces the tree height by overlay tree height optimization. Figure 4 shows THP-ordering tree with* $h(T) = 6$, *reducing the MDE tree height of Figure 2-(c) from 7 to 6.*

7 EXPERIMENTS

In this section, we evaluate the proposed techniques by comparing with the state-of-the-art baselines. All algorithms are implemented in C++ with full optimization on a server with 4 Xeon Gold 6248 2.6GHz CPUs (total 80 cores / 160 threads) and 1.5TB RAM.

7.1 Experimental Setting

Datasets. We obtain six real-life road networks[2] and corresponding trajectories from NavInfo [40], which are shown in Table 1. For

[2]SC includes Guangdong, Guangxi, and Hainan provinces. WC includes Chongqing, Sichuan, Shaanxi, Gansu, Ningxia, Qinghai, Xinjiang, Xizang, Yunnan, and Guizhou

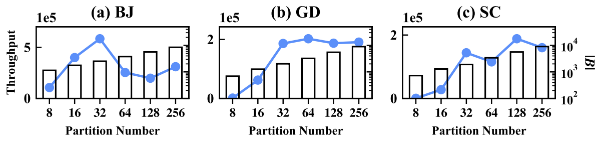


Figure 11: Effect of Partition Number k

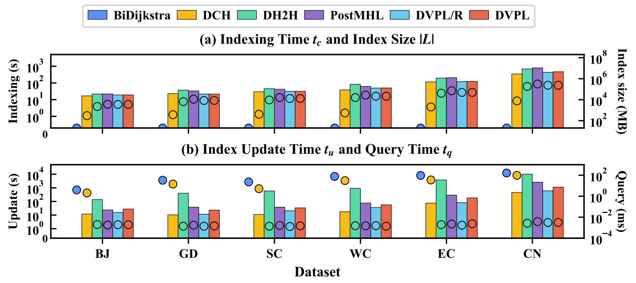


Figure 12: Index Performance (Bar: t_c, t_u, Ball: $|L|, t_q$)

each network G, its $\mathcal{T}$ consists of real-life trajectories that pass through G on January 5, 2016. Real-world queries Q are derived by extracting trajectories between OD pairs located within G. To obtain batch updates $\mathcal{U}$, we first collect the travel time (weight) of the edges traversed by all trajectories in $\mathcal{T}$, and then divide these edge weights into different batches according to their timestamp and batch interval δt. We average the weights of the same edges in each batch, and add an edge update into batch U_i if its value is different from the one in the last batch. The default δt is 300 s, and its average batch update size $|\bar{U}|$ and maximal batch size $|U|_{max}$ are shown in columns 6 and 7 of Table 1. Figure 10-(a) shows the real-life batch updates of GD, where the volumes of increase updates and decrease updates are similar in different batches, with the peak occurring in the afternoon. Figure 10-(b) shows the real-life queries[3], where queries seems to fulfill the power-law distribution (short-distance queries account for the majority).

Algorithms. We compare our solution *DVPL* and *DVPL/R* (*VPL* without *rotating update schedule*) with four baselines: *BiDijkstra* [41], *DCH* [43], *DH2H* [59], and *PostMHL* [72]. *DCH* and *DH2H* are the state-of-the-art (SOAT) dynamic SP indexes in terms of index update and query efficiency, while *PostMHL* is the SOAT in query throughput. Note that we also leverage index-free *BiDijkstra* to answer the queries for *DCH* and *DH2H* during their index maintenance to avoid system unresponsiveness. The parameters of *PostMHL* are set as per [72]. We set the region number $\mathcal{K}$ as 4 for all datasets as per our preliminary experiments. The default thread number is 150.

Measuremets. We use *average query response time* R_q^* as QoS and report *maximum average throughput* λ_q^* over $\mathcal{U}$. Following [72], for each batch update $U \in \mathcal{U}$, we run the system in δt with a certain query arrival rate λ_q over Q and gradually increase it until QoS is violated or the system is overloaded (U cannot be installed in δt). The default δt and R_q^* are 300 s and 1 s. We reuse Q if the query number is insufficient for the test. We also report index construction time t_c, index size $|L|$, average *query time* t_q over Q, and report average *index update time* t_u over $\mathcal{U}$.

provinces. EC includes Shanghai, Jiangsu, Shandong, Anhui, Zhejiang, Fujian, and Jiangxi provinces. CN includes all provinces of China mainland.

[3]Let l_{max} to be the maximum trajectory distances in $\mathcal{T}$ and $x = l_{max}/10$, each set $Q_i, \forall 1 \le i \le 10$ consists of the queries whose travel distances fall in $(x \cdot (i-1), x \cdot i]$.

Table 2: Tree Decomposition Comparison

Metric	Method	BJ	GD	SC	WC	EC	CN
$h(T)$	naive PSP	1433	1059	1531	2102	2285	4695
	PostMHL	705	880	842	950	1593	2126
	DVPL (Ours)	**681**	**635**	**691**	**812**	**1102**	**1670**
$tw(T)$	naive PSP	565	325	536	610	561	1612
	PostMHL	334	291	262	293	459	796
	DVPL (Ours)	**281**	**159**	**202**	**259**	**280**	**521**

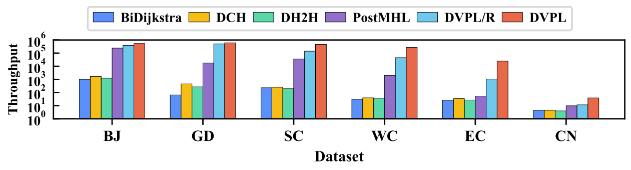


Figure 13: Comparison of Throughput λ^*

Table 3: Average Unaffected Query Number

Query Number During Update	BJ	GD	SC	WC	EC	CN
Total Processed Query Number	28115	96682	80366	90767	274308	9501
Unaffected Query Number $Q_{\bar{A}}$	3841	46397	42520	45023	188570	7097
Percentage of $Q_{\bar{A}}$ (%)	13.66	47.99	52.91	49.60	68.74	74.70

7.2 Experimental Results

Exp 1: Effect of Partition Number. We vary the partition number k from 8 to 256 for *VPL*. Figure 11 illustrates the throughput λ^* and boundary number $|B|$ on BJ, GD, and SC. We observe that both excessively small and large values of k can degrade λ^*, with the optimal k generally increasing with the size of the road network. This is because a larger k enhances partition parallelization but may result in a larger overlay index, which is challenging to parallelize. The default k is in the last column of Table 1.

Exp 2: Index Performance Comparison. We evaluate the index performance $(t_c, |L|, t_q, t_u)$ on all datasets, with the results presented in Figure 12. Compared with *PostMHL*, *DVPL* speeds up index construction by up to 1.84× (on CN) while reducing the index size by up to 31% (on EC), since it has a much better TD. As shown in Table 2, the TD of *VPL* has a much smaller tree height $h(T)$ and treewidth $tw(T)$ than that of *PostMHL*, achieving up to 30.8% and 45.3% improvements, respectively. These improvements are even more pronounced when compared to the *unoptimized boundary-first PSP ordering* in [36, 72] (denoted as *naive PSP*), highlighting the efficacy of our *tree height-aware PSP ordering*. This ordering optimization is also conducive to query processing and index maintenance. Besides, our dynamic algorithm reduces the update workload by the mixed update manner. As such, *DVPL/R* and *DVPL* significantly enhance the index maintenance, achieving up to 4.25× and 2.29× speed up over *PostMHL*, respectively. Note that *DVPL/R* has better index update efficiency than *DVPL* as it has only one region and fully exploits parallelization between all affected partitions during δt. Nevertheless, *DVPL/R* fails to optimize the unaffected query number by the rotating update schedule, resulting in an inferior throughput than *DVPL* (will show in Exp-3).

Exp 3: Throughput Comparison. We report the throughput comparison in Figure 13. *PostMHL* exhibits considerable throughput enhancement over *BiDijkstra*, *DCH*, and *DH2H*, attributed to its multi-stage query enhancement and rapid index maintenance. Nevertheless, *DVPL/R* still outperforms all baselines due to comparable

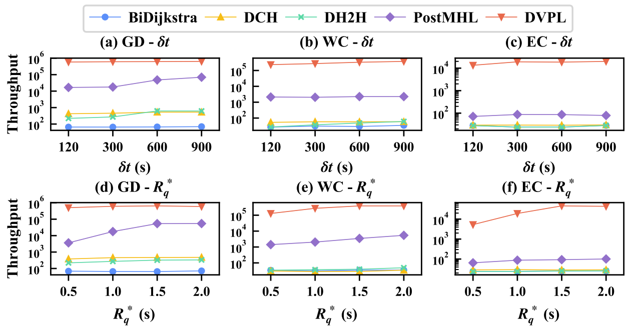


Figure 14: Varying δt and R_q^*

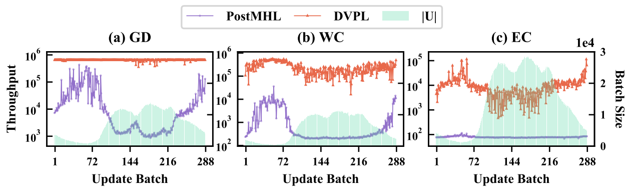


Figure 15: Throughput Evolution (Polyline: λ^*, Bar: $|U|$)

query efficiency but faster index maintenance than *PostMHL*, achieving up to 28.4× speed up in λ^*. Additionally, *DVPL* further leverages *rotating update schedule* optimization to boost the throughput on large road networks, yielding up to two orders of magnitude's improvement than state-of-the-art (132× and 460× on WC and EC, respectively). This enhancement is facilitated by the identification of a significant number of unaffected queries during updates, as detailed in Table 3. There are 188570 unaffected queries detected in EC, comprising 68.74% of total processed queries during index maintenance. The number of unaffected queries of *DVPL* is limited on CN because *U3* fails to be installed within δt, thereby preventing V-Stage 2 from detecting more same-partition and cross-partition queries. Nevertheless, *DVPL* still outperforms all baselines on CN, enhancing the query throughput by 3.9×.

Exp 4: Effect of Update Interval and Response Time. We vary the update interval δt and average response time R_q^* to assess performance across different scenarios. As shown in Figures 14 (a)-(c), increasing δt generally leads to better throughput, but the benefit could be limited because larger δt also leads to larger batch sizes collected during the update interval. Figures 14 (d)-(f) show the throughput when varying R_q^* from 0.5 to 2.0 s. Observe that a smaller R_q^* may lead to deteriorated throughput for *PostMHL* as its later stages cannot be released for efficient query processing. By contrast, *DVPL* exhibits better throughput by detecting unaffected queries, and consistently surpasses all baselines across scenarios, enhancing throughput by up to two orders of magnitude.

Exp 5: Throughput Evolution. We further compare the throughput evolution of *DVPL* and *PostMHL* within a day. Figure 15 presents the results on GD, WC, and EC, where the polyline represents throughput while the green bar indicates the batch size. *PostMHL* is more sensitive to the batch size: a larger batch size (during 7:00-22:00) leads to much lower throughput as index maintenance is time-consuming and efficient cross-boundary query processing may not be released within δt. In contrast, *DVPL* is more robust due to the identification of unaffected queries and consistently outperforms *PostMHL* at any time, with a throughput improvement of about 1-3 orders of magnitude.

8 RELATED WORK

8.1 Static Shortest Path Algorithms

The shortest path algorithms can be divided into two categories. The index-free methods [13, 20, 23, 41, 69] do not need index maintenance and are not influenced by the network update. The *Dijkstra's* [13] searches in a best-first manner, while A^* [20] and *BiDijkstra* [41] reduces search space with heuristic pruning. However, they are still slow to search the graphs. Although batch processing [32, 48, 60] could improve the throughput but not by orders of magnitude. To improve efficiency, index-based methods have been proposed, such as *goal-directed methods* [19, 49], *shortcut methods* [17, 46], *2-hop labeling* [1–3, 10, 14, 24, 33, 34], and *tree decomposition-based methods* [9, 28, 42, 52]. These indexes are about 1-6 orders of magnitude faster. Among them, *PLL* [3] and *PSL* [33] have the fastest query efficiency on small-world networks, while *H2H* [42] and *P2H* [9] achieve the state-of-the-art query efficiency on road networks. To scale up the shortest path algorithm for large graphs, the partitioned shortest path (PSP) index has also been investigated in recent years [16, 34, 51, 64, 68]. However, they are initially designed for static graphs and take time to handle updates.

8.2 Dynamic Shortest Path Algorithms

To adapt the path index to dynamic scenarios, their maintenance methods [4, 11, 15, 18, 43, 44, 53, 59, 61, 62, 64, 71] have been studied recently. Among them, [4, 11, 44, 61, 62] focus on *PLL* and *PSL*. For the road networks, *DCH* [43] maintains CH with a high update efficiency. *DH2H* [59, 66] further maintain the labels with faster query. [63] identifies the throughput as the key issue of the dynamic path indexes. [64] systematically studies the dynamic PSP index and puts forward three general PSP strategies and a universal scheme for designing the dynamic PSP index. *PostMHL* [72] combines DCH and DH2H to form a unified index and can process the update and query in a multi-stage pipeline to improve query throughput. However, none of them have investigated the impact of updates on queries, thus failing to enhance throughput from the perspective of unaffected queries.

9 CONCLUSION

In this paper, we propose a spatio-temporal optimized framework *VPSP* to achieve high-throughput SP query processing on dynamic road networks. By partitioning the network spatially and organizing the update temporally, our *VPSP* framework with *rotating update schedule* eliminates the periodic index unavailability of the SP index caused by index maintenance. Besides, we propose an index validation algorithm *TIV* to efficiently identify the unaffected queries, an efficient dynamic algorithm to support mixed updates, and *tree height-aware PSP ordering* to reduce tree decomposition height dramatically, further improving the performance of *VPSP* framework. Experiments on real-life road networks validate the effectiveness of our solution, which outperforms the state-of-the-art by two orders of magnitude in query throughput.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms: 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings 10*. Springer, 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*. Springer, 24–35.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*. 237–248.

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys* (2016), 19–80.

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[7] Michael A Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*. Springer, 88–94.

[8] Anne Berry, Pinar Heggernes, and Genevieve Simonet. 2003. The minimum degree heuristic and the minimal triangulation process. In *Graph-Theoretic Concepts in Computer Science: 29th International Workshop, WG 2003. Elspeet, The Netherlands, June 19-21, 2003. Revised Papers 29*. Springer, 58–70.

[9] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: efficient distance querying on road networks by projected vertex separators. In *Proceedings of the 2021 International Conference on Management of Data*. 313–325.

[10] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[11] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully dynamic 2-hop cover labeling. *Journal of Experimental Algorithmics (JEA)* 24 (2019), 1–36.

[12] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. 2011. Graph partitioning with natural cuts. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1135–1146.

[13] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[14] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling-Scaling Up Distance Queries on Road Networks. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[15] Muhammad Farhan, Qing Wang, and Henning Koehler. 2022. BatchHL: Answering Distance Queries on Batch-Dynamic Networks at Scale. In *Proceedings of the 2022 International Conference on Management of Data*. 2020–2033.

[16] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan Mckay. 2018. A highly scalable labelling approach for exact distance queries in complex networks. *EDBT* (2018).

[17] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International workshop on experimental and efficient algorithms*. Springer, 319–333.

[18] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.

[19] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory.. In *SODA*, Vol. 5. Citeseer, 156–165.

[20] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[21] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. 2019. An efficient framework for correctness-aware knn queries on road networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1298–1309.

[22] Shuai Huang, Yong Wang, Tianyu Zhao, and Guoliang Li. 2021. A learning-based method for computing shortest path distances on road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 360–371.

[23] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh. 1994. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of VNIS'94-1994 Vehicle Navigation and Information Systems Conference*. IEEE, 291–296.

[24] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop Doubling Label Indexing for Point-to-Point Distance Querying on Scale-Free Networks. *Proceedings of the VLDB Endowment* 7, 12 (2014).

[25] Shan Jiang, Yilun Zhang, Ran Liu, Mohsen Jafari, and Mohamed Kharbeche. 2022. Data-driven optimization for dynamic shortest path problem considering traffic safety. *IEEE Transactions on Intelligent Transportation Systems* 23, 10 (2022), 18237–18252.

[26] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING* 20, 1 (1998), 359–392.

[27] Huayu Li, Yong Ge, Richang Hong, and Hengshu Zhu. 2016. Point-of-interest recommendations: Learning potential check-ins from friends. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 975–984.

[28] Jiajia Li, Yongzhi Chen, Mengxuan Zhang, and Lei Li. 2024. A CPU-GPU Hybrid Labelling Algorithm for Massive Shortest Distance Queries on Road Networks. *Proceedings of the VLDB Endowment* 18, 3 (2024), 770–783.

[29] Jiajia Li, Cancan Ni, Dan He, Lei Li, Xiufeng Xia, and Xiaofang Zhou. 2023. Efficient k NN query for moving objects on time-dependent road networks. *The VLDB Journal* 32, 3 (2023), 575–594.

[30] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-dependent hop labeling on road network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 902–913.

[31] Lei Li, Sibo Wang, and Xiaofang Zhou. 2020. Fastest path query answering using time-dependent hop-labeling in road network. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[32] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast query decomposition for batch shortest path processing in road networks. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1189–1200.

[33] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.

[34] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[35] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. 2021. Efficient constrained shortest path query answering with forest hop labeling. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1763–1774.

[36] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2022. FHL-cube: multi-constraint shortest path querying with flexible combination of constraints. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3112–3125.

[37] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2022. Multi-constraint shortest path using forest hop labeling. *The VLDB Journal* (2022), 1–27.

[38] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment* 11, 5 (2018), 594–606.

[39] TomTom Maps. [n. d.]. *Intermediate Traffic API - Introduction.* https://developer.tomtom.com/intermediate-traffic-service/documentation/product-information/introduction?source_app=b2b&source_product=traffic-apis

[40] NavInfo. [n. d.]. *NavInfo Company.* http://www.navinfo.com/

[41] T Alastair J Nicholson. 1966. Finding the shortest route between two points in a network. *The computer journal* 9, 3 (1966), 275–280.

[42] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.

[43] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 5 (2020), 602–615.

[44] Yongrui Qin, Quan Z Sheng, Nickolas JG Falkner, Lina Yao, and Simon Parkinson. 2017. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *World Wide Web* 20 (2017), 915–937.

[45] Huiming Qu and Alexandros Labrinidis. 2006. Preference-aware query and update scheduling in web-databases. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 356–365.

[46] Peter Sanders and Dominik Schultes. 2005. Highway hierarchies hasten exact shortest path queries. In *ESA*, Vol. 3669. Springer, 568–579.

[47] Anuj Sharma, Vesal Ahsani, and Sandeep Rawat. 2017. Evaluation of opportunities and challenges of using INRIX data for real-time performance monitoring and historical trend assessment. (2017). https://lib.dr.iastate.edu/ccee_reports/24

[48] Jeppe Rishede Thomsen, Man Lung Yiu, and Christian S Jensen. 2012. Effective caching of shortest paths for location-based services. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 313–324.

[49] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. 2005. Geometric containers for efficient shortest-path computation. *Journal of Experimental*

Algorithmics (JEA) 10 (2005), 1–3.

[50] Sibo Wang, Xiaokui Xiao, Yin Yang, and Wenqing Lin. 2016. Effective indexing for approximate constrained shortest path queries on large road networks. *Proceedings of the VLDB Endowment* 10, 2 (2016), 61–72.

[51] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-sketch: Scaling shortest path graph queries on very large networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1946–1958.

[52] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 99–110.

[53] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1841–1856.

[54] Ren Xiaofeng. [n. d.]. *Connecting the Real World: Algorithms and Innovations behind Amap*. https://www.alibabacloud.com/blog/connecting-the-real-world-algorithms-and-innovations-behind-amap_596304

[55] Jinbo Xu, Feng Jiao, and Bonnie Berger. 2005. A tree-decomposition approach to protein structure prediction. In *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*. IEEE, 247–256.

[56] Yehong Xu, Lei Li, Mengxuan Zhang, Zizhuo Xu, and Xiaofang Zhou. 2023. Global routing optimization in road networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2524–2537.

[57] Jieping Ye. 2018. Big Data at Didi Chuxing. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 1341–1341.

[58] Jianting Zhang, Camille Kamga, Hongmian Gong, and Le Gruenwald. 2012. U2SOD-DB: a database system to manage large-scale ubiquitous urban sensing origin-destination data. In *Proceedings of the ACM SIGKDD International Workshop on Urban Computing*. 163–171.

[59] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic hub labeling for road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 336–347.

[60] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2020. Stream processing of shortest path query in dynamic road networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2458–2471.

[61] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

[62] Mengxuan Zhang, Lei Li, Goce Trajcevski, Andreas Zufle, and Xiaofang Zhou. 2023. Parallel Hub Labeling Maintenance With High Efficiency in Dynamic Small-World Networks. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[63] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An experimental evaluation and guideline for path finding in weighted dynamic network. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2127–2140.

[64] Mengxuan Zhang, Xinjie Zhou, Lei Li, Ziyi Liu, Goce Trajcevski, Yan Huang, and Xiaofang Zhou. 2023. A Universal Scheme for Partitioned Dynamic Shortest Path Index. *arXiv preprint arXiv:2310.08213* (2023).

[65] Mengxuan Zhang, Xinjie Zhou, Lei Li, and Xiaofang Zhou. 2025. Partitioned Dynamic Hub Labeling for Large Road Networks. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[66] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative subboundedness of contraction hierarchy and hierarchical 2-hop index in dynamic road networks. In *Proceedings of the 2022 International Conference on Management of Data*. 1992–2005.

[67] Yu Zheng, Yanchi Liu, Jing Yuan, and Xing Xie. 2011. Urban computing with taxicabs. In *Proceedings of the 13th international conference on Ubiquitous computing*. 89–98.

[68] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. 2013. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 39–48.

[69] Xinjie Zhou, Kai Huang, Lei Li, Mengxuan Zhang, and Xiaofang Zhou. 2024. I/o-efficient multi-criteria shortest paths query processing on large graphs. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[70] Xinjie Zhou, Lei Li, Mengxuan Zhang, and Xiaofang Zhou. [n. d.]. *A Spatio-temporal Optimized Framework for High-throughput Shortest Distance Query Processing on Dynamic Road Networks (Full Version)*. https://github.com/ZXJ-DSA/HASP/blob/main/HighAvailableSP_FullVersion.pdf

[71] Xinjie Zhou, Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2024. Scalable Distance Labeling Maintenance and Construction for Dynamic Small-World Networks. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE.

[72] Xinjie Zhou, Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2025. High Throughput Shortest Distance Query Processing on Large Dynamic Road Networks. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE.

10 APPENDIX

10.1 Pseudo-code of VPL Index Construction

The pseudo-code of *VPL* construction is presented in Algorithm 4. Sepcifically, we adopt *PUNCH* [12] as the graph partitioning method to divide G into multiple subgraphs $\{G_i | 1 \leq i \leq k\}$ (Line 1), because it generates the minimal boundary number on road networks [64] and its partition result complies well with the query clusters. Given the partition result, we first derive its order r via *tree height-aware PSP ordering* (will be introduced in Section 6.2). A tree decomposition T and the *shortcut arrays* of its tree nodes are then built based on r (Line 3). As for the label construction, we first build the overlay index for boundary vertices from the root of T (Lines 4-5) and then construct the labels for all partitions' non-boundary vertices in parallel (Lines 6-8). For each tree node $X(v)$, we first obtain the position array by identifying the position of any vertex $x_j \in X(v)$ in $X(v).A$ (Lines 12-13). The distance array is computed by using $X(v).N$ as the vertex separator while the boundary array is computed by inheriting corresponding label entries in $X(v).dis$.

Algorithm 4: VPL Index Construction

```
   Input: Road network $G$
   Output: VPL index $L = \{X(v) | v \in V\}$
1  $\tilde{G}, \{G_i | i \in [1, k]\} \leftarrow$ PUNCH($G$);      ▷ Obtain the partition result by PUNCH [12]
2  $r \leftarrow$PSPOrdering($\tilde{G}, \{G_i\}$);                ▷ Tree Height-Aware PSP Vertex Order
3  $T \leftarrow$TDandShortcutBuild($G, r$);          ▷ TD and shortcut construction based on $r$
4  for all $X(v) \in T, v \in \tilde{G}$ in a top-down manner do
5      TreeNodeBuild($X(v)$);                                  ▷ Overlay index construction
6  parallel_for $G_i, i \in [1, k]$
7      for all $X(v) \in T, v \in G_i \setminus B_i$ in a top-down manner do
8          TreeNodeBuild($X(v)$);                            ▷ Partition index construction
9  return $L$;
10 Function TreeNodeBuild($X(v)$):
11     Suppose $v \in G_i$ and $X(v) = X(v).N \cup v = (x_1, x_2, ..., x_{|X(v)|})$;
12     for $j = 1$ to $|X(v)|$ do
13         $X(v).pos[j] \leftarrow$the position of $x_j$ in $X(v).A$;
14     for $j = 1$ to $|X(v).A| - 1$ do
15         $c \leftarrow X(v).A[j]; X(v).dis[j] \leftarrow \infty$;
16         for $k = 1$ to $|X(v).N|$ do
17             if $X(v).pos[k] > j$ then $d \leftarrow X(x_k).dis[j]$;
18             else $d \leftarrow X(c).dis[X(v).pos[k]]$;
19             $X(v).dis[j] \leftarrow \min\{X(v).dis[j], X(v).sc[k] + d\}$;
20         if $v \notin B$ and $c \in B_i$ then
21             $X(v).disB[l] \leftarrow X(v).dis[j]$, where $l$ is the position of $c$ in $B_i$;
22     $X(v).dis[|X(v).A|] \leftarrow 0$;
```

10.2 Pseudo-code of DVPL Shortcut Maintenance

Algorithm 5 presents the pseudo-code of mixed shortcut maintenance. We use the non-partitioned L to demonstrate the key idea of *DVPL* while it is easy to extend it to a partitioned one with parallelization. In particular, we use a min-heap set Q to record affected tree nodes, and a min-heap set $S(a)$ to record neighbors of a' affected shortcuts, ensuring that the updates are processed in a bottom-up manner. When Q is not empty, we iteratively pop the vertex a with the minimum order from Q, and handle its affected shortcuts $sc(a, b), \forall b \in S[a]$ (Lines 2-8). For each affected shortcut, we compute its new value d_{sc} and supportive number ϕ_{sc} via *minimum weight property* [59] (Line 5). For example, as shown in Figure 8-(a), suppose $sc(a, b)$ has two supportive vertices v_1, v_2 and the edge $e(a, b)$ decreases from 4 to 3, we have $d_{sc} = 3$ and $\phi_{sc} = 1$ since only edge $e(a, b)$ makes new $sc(a, b)$ holds after update. We compare $sc(a, b)$ with d_{sc} to figure out its update types, and assign Cw with old $sc(a, b)$ or new d_{sc} for the increase and decrease (Lines 6-7). After refreshing $sc(a, b)$ and $\phi_{sc}(a, b)$, we propagate the update in function SCUpdatePropagate, where we first identify the affected labels and then detect other affected shortcuts. We next take the decrease as an example (Lines 10-14). We first identify the affected distance labels in function DecUpdateCheck, *i.e.,* for each $u \in X(a).A$, if $L(a, u) > Cw + L(b, u)$ or $L(a, u) > Cw + L(u, b)$, we say the label $L(a, u)$ is affected and use $C(a, u)$ to record this information, where $C(a, u) > 0$ indicates increase update while $C(a, u) < 0$ represents decrease update. We recognize $X(a)$ as affected if there is any $L(a, u)$ that is affected. We also set the boolean value $X(a).ifU$ as true if there is any $L(a, ut)$ that is affected. As illustrate in Figure 8-(b), $L(a, u_1)$ is affected since $Cw + L(u_1, b) = 3 + 2 = 5 < L(a, u_1)$ while $L(a, u_2)$ is unaffected. We then check whether each neighboring shortcut $sc(b, u), u \in X(b).N$ or $sc(u, b), b \in X(u).N$ is affected in function DecSCCheck, where $sc(u, b)$ is recorded as affected if new $dis < sc(u, b)$ (Lines 20-22).

Algorithm 5: DVPL: Mixed Shortcut Maintenance

```
   Input: $G$ and its mixed updates $U, L$
   Output: Shortcut Updated $L$
1  $Q.insert(a), S[a].insert(b), \forall e(a, b) \in U$; //$Q$ is a min-heap set, $S[a]$ is a set for $a$
2  while $Q$ is not empty do
3      $a \leftarrow Q.begin(), Q.erase(a)$; //$a$ has the minimum vertex order in $Q$
4      for each $b \in S[a]$ do
5          Compute new shortcut $d_{sc}$ and $\phi_{sc}$ by minimum weight property [59]
6          if $d_{sc} > sc(a, b)$ then $Cw \leftarrow sc(a, b)$; //increase update
7          else if $d_{sc} < sc(a, b)$ then $Cw \leftarrow d_{sc}$; //decrease update
8          $sc(a, b) \leftarrow d_{sc}$; $\phi_{sc}(a, b) \leftarrow \phi_{sc}$; SCUpdatePropagate($a, b, Cw$);
9  Function SCUpdatePropagate($a, b, Cw$):
10     if $sc(a, b)$ is decrease update then
11         DecLabelCheck($a, b, Cw$);                          ▷ Identify affected labels
12         for each $u \in X(a).N$ do
13             if $u \in X(b).N$ then DecSCCheck($b, u, a, Cw + sc(a, u)$);
14             else if $b \in X(u).N$ then DecSCCheck($u, b, a, Cw + sc(a, u)$);
15     else if $sc(a, b)$ is increase update then
16         if $X(a).FN[b] = true$ then IncLabelCheck($a, b, Cw$);
17         for each $u \in X(a).N$ do
18             if $u \in X(b).N$ then IncSCCheck($b, u, Cw + sc(a, u)$);
19             else if $b \in X(u).N$ then IncSCCheck($u, b, Cw + sc(a, u)$);
20 Function DecSCCheck($u, b, a, dis$):
21     if $dis = sc(u, b)$ and $u \notin S[a]$ then $\phi_{sc}(u, b) \leftarrow \phi_{sc}(u, b) + 1$;
22     else if $dis < sc(u, b)$ then $Q.insert(u), S[u].insert(b)$;
23 Function IncSCCheck($u, b, dis$):
24     if $dis = sc(u, b)$ then
25         $\phi_{sc}(u, b) \leftarrow \phi_{sc}(u, b) - 1$;
26         if $\phi_{sc}(u, b) < 1$ then $Q.insert(u), S[u].insert(b)$;
27 Function DecLabelCheck($a, b, dis$):
28     if $Cw = L(a, b)$ then $\phi_d(a, b) \leftarrow \phi_d(a, b) + 1, X(a).FN[b] \leftarrow true$;
29     else if $Cw < L(a, b)$ then $C(a, b) \leftarrow -1, X(a).ifU \leftarrow true$;
30     for each $u \in X(a).A$ do
31         if $L(a, u) > Cw + L(b, u), r(u) > r(b)$ or
             $L(a, u) > Cw + L(u, b), r(u) < r(b)$ then $C(a, u) \leftarrow -1$;
32 Function IncLabelCheck($a, b, Cw$):
33     for each $u \in X(a).A$ do
34         if $L(a, u) = Cw + L(b, u), r(u) > r(b)$ or
             $L(a, u) = Cw + L(u, b), r(u) < r(b)$ or $u = b$ then
35             $\phi_d(a, u) \leftarrow \phi_d(a, u) - 1$;
36             if $\phi_d(a, u) < 1$ then $C(a, u) \leftarrow 1, X(a).ifU \leftarrow true$;
```

10.3 Proofs

Proof of Lemma 1. There are three cases for edge weight decrease update.

Case 1: $e(a, b) \in sp(s, t)$ before update. In this case, $e(a, b)$ must be in $sp'(s, t)$ after update, thus $d'(s, t) = d(s, a) + |e'(a, b)| + d(b, t) < d(s, t)$.

Case 2: $e(a, b) \notin sp(s, t)$ before update but is in $sp'(s, t)$ after update. In this case, we also have $d'(s, t) = d(s, a) + |e'(a, b)| + d(b, t) < d(s, t)$.

Case 3: $e(a, b) \notin sp(s, t)$ before update and is not in $sp'(s, t)$ after update. Then $d'(s, t) = d(s, a) + |e'(a, b)| + d(b, t) > d(s, t)$. □

Proof of Lemma 2. The lower bound of the number of affected queries is determined by *Case D1*, which is $\phi(a, b)$. *Case D2* determines the upper bound of the number of affected queries, which may affect all queries on G in the worst case. □

Proof of Lemma 5. As per the definitions of $\Omega_{\bar{A}}$, $L(s)$ and $L(t)$ are untouched during the index maintenance. Therefore, $Q(s, t)$ is an unaffected query and could be computed by the "old" index L. □

Proof of Lemma 6. $d(s, t) \cdot v_{max}$ is the longest spatial distance that $sp(s, t)$ could traverse since v_{max} is the maximum speed limit on G. Therefore, using $d(s, t) \cdot v_{max}$ as the major axis of the ellipse with foci at s and t is the largest spatial area $sp(s, t)$ can traverse. □

Proof of Theorem 2. We prove it by contradiction. Suppose G_j is inner-affected, there must $\exists sp'(s, t), s, t \in G_j$ that passes through at least one updated edge $e \in G_i$. We take the concise form of $sp'(s, t)$ as $sp'_c = \langle s, b_1, ..., b_q, t \rangle$. We have $b_1, b_q \in G_j$ and $sp'(b_1, b_q)$ passes through $e \in G_i$ after update. There are two update cases for the length change of $sp(b_1, b_q)$:

Case 1: suppose $d(b_1, b_q) > d'(b_1, b_q)$. $sp'(b_1, b_q)$ must pass through at least one decreased $e(a, b) \in G_i$ after update as per Lemma 1. However, $d'(b_1, b_q) = d'(b_1, a) + e'(a, b) + d'(a, b_q) > d(b_1, b_q)$ because $\Omega_{max}(G_j)$ does not overlap with the MBR of G_i, which is contradict with the assumption $d(b_1, b_q) > d'(b_1, b_q)$.

Case 2: suppose $d(b_1, b_q) < d'(b_1, b_q)$. $sp(b_1, b_q)$ must pass through at least one increased edge $e(a, b) \in G_i$ before update as per Case I1, which is contradict with the assumption $\Omega_{max}(G_j)$ does not overlap with the MBR of G_i. □

Proof of Lemma 7. No updates within G_i means that there is no inner change source, while $d(b_1, b_2), \forall b_1, b_2 \in B_i$ remain unchanged after indicates that there is no outer change source triggering the same-partition query. Therefore, G_i is inner-unaffected. □

Proof of Theorem 4. $C(u, b) = 0$ means that $L(u, b)$ is unaffected. There are three cases for $L(u, b)$.

Case 1: $L(u, b)$ decreases to $L'(u, b)$. We have $L'(a, b) + L'(u, b) < L(u, a)$ and $L(u, a)$ needs to be maintained. Therefore, there is no need to increase $\phi_d(u, a)$.

Case 2: $L(u, b)$ increases to $L'(u, b)$. We have $L'(a, b) + L'(u, b) \neq L(u, a)$, and thus there is no need to increase $\phi_d(u, a)$.

Case 3: $L(u, b)$ remains unaffected. We have $L'(a, b) + L'(u, b) = L(u, a)$, and thus it is necessary to increase $\phi_d(u, a)$ by 1. Otherwise, the increase update of $L(u, a)$ may be incorrect in the next batch due to the wrong $\phi_d(u, a)$. □

Proof of Theorem 5. Algorithm 2 leverages a set Q to record all affected tree nodes, which incurs $O(\Delta_{sc}^T \cdot \log(\Delta_{sc}^T)$. The access of affected shortcuts in $S[a]$ needs $O(\Delta_{sc} \cdot \log(w))$. The computation of all new shortcuts and shortcut update propagation both are $O(\Delta_{sc} \cdot w)$ while the label check during propagation needs $O(\Delta_{sc} \cdot h)$. Therefore, the complexity of Algorithm 2 is $O(\Delta_{sc}^T \cdot \log(\Delta_{sc}^T) + \Delta_{sc} \cdot (w + \log(w) + h))$. Algorithm 3 checks all tree nodes with $O(\Delta^T)$ while only deal with the affected labels $O(\Delta_d)$ with the computation of new label $O(w)$, the update propagation $O(deg_{G^-}^{max})$. Therefore, the complexity is $O(\Delta_d \cdot (w + deg_{G^-}^{max}) + n)$. □

Proof of Lemma 8. *Step 3* ensures the greedy PSP ordering is boundary-first. Given any query $Q(s, t), s \in G_i, t \in G_j$, we take the concise form of $sp(s, t)$ with only OD pair and boundary vertices as $sp_c = \langle s, b_1, ..., b_q, t \rangle$, and prove the correctness in three cases.

Case 1: $s, t \in B$. $d(s, t)$ must be correctly computed since G_c preserves all shortest distances among boundary vertices.

Case 2: $s \notin B, t \in B$ (vice versa). Suppose u is the highest-order non-boundary vertex in $sp(s, d_1)$, there are two subcases. *SubCase 1*: $u = s$, we have $b_1 \in X(s).N$ since other vertex $v \in sp(s, b_1)$ is contracted before s, then b_1 is the ancestor of s and it naturally holds. *SubCase 2*: $u \neq s$, we have $b_1 \in X(u).N$ as per *SubCase 1* and $u \in X(s).A$ since $\forall v \in sp(s, u)$ must be contracted before u, forming the shortcut $sc(s, u)$. So $b_1 \in X(s).A$ and *Case 2* holds.

Case 3: $s, t \notin B$. There must exist $L(s, b_1)$ and $L(t, b_q)$ as per *Case 2*, and $d(b_1, b_q)$ is computed by *Case 1*. Hence, this case holds. □

10.4 Additional Experimental Results

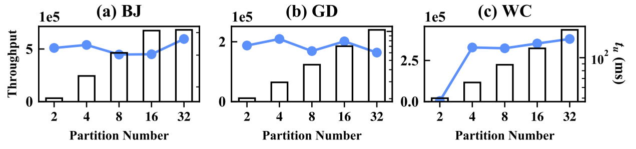


Figure 16: Effect of Region Number $\mathcal{K}$ of VPL

Exp 6: Effect of Region Number of VPL. We vary the region number $\mathcal{K}$ from 2 to 32 for *VPL*. Figure 16 illustrates the throughput λ^* and update time t_u on BJ, GD, and WC. We observe that both too small $\mathcal{K}$ may significantly degrade λ^* as too many partitions undergo updates on each region update phase, resulting in a limited number of unaffected partitions and unaffected queries. By contrast, a larger $\mathcal{K}$ may lead to increased throughput in most cases (*e.g.*, BJ and WC). Nevertheless, it also increases the index update time. To make the system more robust to various scenarios, we select $\mathcal{K} = 4$ for all datasets as it provides a balanced trade-off between throughput and update time.