High Throughput Shortest Path Query Processing for Large Dynamic Road Networks

Anonymous Author(s)

ABSTRACT

Shortest path (SP) computation is a fundamental operation for location-based services, and the SP index has been extensively studied to achieve fast query processing. However, road networks are inherently dynamic and evolve with massive updates caused by the changing traffic flow or traffic accidents. As a result, even the state-of-the-art *Dynamic Contraction Hierarchies (DCH)* and *Dynamic Hierarchical 2-Hop labeling (DH2H)* fail to deal with high throughput queries due to either low query efficiency or poor dynamic adaption. Therefore, in this paper, we aim to propose efficient SP indexes to process high throughput shortest path queries on large dynamic road networks. Specifically, we first put forward a *Multi-stage Hierarchical hub Labeling (MHL)* to subtly exploit the fast index maintenance and exceptional query efficiency of DCH and DH2H. To scale *MHL* to large road networks, we further propose a non-trivial *Partitioned MHL (PMHL)* that utilizes novel partitioned shortest path strategies and thread parallelization to achieve consecutive query efficiency improvement and faster index maintenance. Lastly, to further enhance PMHL's query throughput, we provide an insightful analysis of the essence of partitioned SP index and propose a novel *Post-partitioning MHL (PostMHL)* adopting the *tree decomposition-based graph partitioning* for faster index update and query efficiency. Experiments on real-world road networks show that our methods outperform state-of-the-art baselines in query throughput, yielding up to 1-4 orders of magnitude improvement.

1 INTRODUCTION

Finding the shortest path (SP) from one place to another is a building block in many location-based applications, such as route planning [8, 36, 48, 62], traffic optimization [31, 59], POI recommendation [34], and k-nearest neighbor searching [35, 41, 70] in the road networks. These location-based services typically generate numerous shortest path or distance queries. For instance, as one of China's most widely used ride-sharing platforms, in 2018, *Didi Chuxing* had about 40 billion routing queries every day [60], which equals about 462,962 queries per second on average, putting a heavy burden on the vehicle routing system. The basic solutions for SP computation are index-free methods such as *Dijkstra's* [17], A^* [27], and *BiDijkstra* [45], which traverse the graph on the fly in a best-first manner. However, they suffer from low query efficiency on large road networks and thus have low *query throughput* (*i.e.,* the number of SP queries that can be processed during a given time interval). To improve the query efficiency, a plethora of index-based methods [4–6, 11–13, 23, 24, 30, 33, 39, 40, 47, 50, 56] have been proposed in the past two decades. With the aid of the pre-computed information, they are faster than Dijkstra's algorithm by 1–6 orders of magnitude, but they are initially designed for static networks. Unfortunately, real-world road networks are inherently dynamic as the edge weights (travel time) usually evolve with massive updates caused by the changing traffic flow or traffic accidents [48, 62].

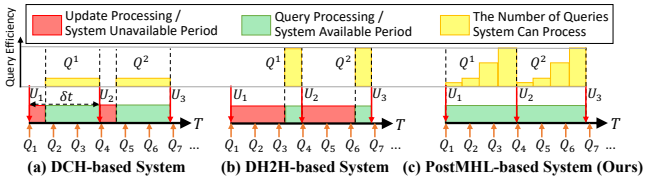


Figure 1: Comparison of Different Path-finding Systems

For example, according to [69], in 2010, Beijing had approximately 67,000 licensed taxis generating over 1.2 million trip records per day, which means about 13.8 edge weight updates per second on average. Similarly, in 2012, there were an average of 5.78 new updates per second in New York City [61]. Due to the frequent updates, commercial live map service providers such as TomTom and INRIX generally update their road traffic information every minute to reflect real traffic conditions [48, 51].

To handle the dynamic road networks, SP index maintenance methods [7, 18, 19, 28, 48, 49, 57, 62, 64, 65, 68, 71] are widely investigated in the past decade. In particular, *Dynamic Contraction Hierarchies (DCH)* [48] and *Dynamic Hierarchical 2-Hop labeling (DH2H)* [62] are two state-of-the-art methods for road networks in terms of index update and query processing efficiency. However, as identified in [66], their query throughput is limited in dynamic road networks due to either low query efficiency or slow index maintenance. Specifically, DCH maintains the index very efficiently, but its (including variants [8, 16]) querying is only 1–3 orders of magnitude faster than Dijkstra's. As illustrated in Figure 1-(a), DCH's query throughput (wide while low yellow block) is limited by its slow query efficiency, despite having a long time for query processing after the index update. By contrast, DH2H is efficient in query with 4-6 orders of magnitude faster than Dijkstra's, but it is slow in index maintenance. As a result, its query throughput (narrow while tall yellow block) is limited by its short available period, as shown in Figure 1-(b). Worse still, when it comes to massive updates or large networks, it becomes completely unavailable due to time-consuming index maintenance (see Exp-5 of Section 6). It is important to note that although many SP queries may be located within a city, nation-level route planning is still important and necessary in scenarios such as inter-provincial logistics and self-driving travel. For example, during the Spring Festival of 2024, there were approximately 7.2 billion driving trips in China, with a peak of 65.2 million routing queries in one day (most were cross-regional queries), averaging up to 755 queries per second [3]. Besides, according to [46], the USA's interstate highway system in 2021 comprised 1 percent of total highway miles available but carried 26 percent of total highway vehicle-miles of travel, which also highlights the importance of national route planning. Therefore, it is crucial to scale dynamic SP indexes to large road networks.

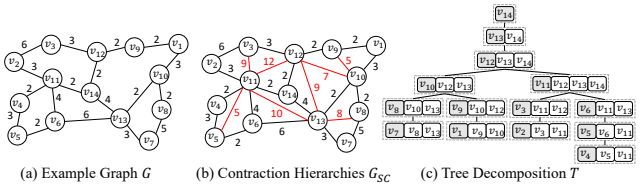
Motivation. To our best knowledge, many navigation service providers tend to adopt CH-based solutions for route planning due to their quick adaptability to dynamic environments [8]. However,

as mentioned before, the CH-based solutions have relatively low query efficiency (about 3 orders of magnitude slower than DH2H) and thus necessitate the deployment of thousands of machines to handle high throughput queries, which is obviously resource-inefficient. Therefore, in this paper, we aim to propose efficient dynamic SP indexes with high query efficiency (comparable to DH2H) and fast index maintenance to handle high throughput SP queries on large dynamic networks.

Our Idea. As discussed before, existing SP indexes can hardly process high throughput queries in dynamic road networks since they are either slow to update (*e.g.,* DH2H) or query (*e.g.,* DCH). Then, a question arises naturally: could we take advantage of these indexes and aggregate them in one system such that the system is responsive with higher throughput? Specifically, when a batch of updates arrives, we first leverage the index-free method to process queries on the fly at the starting stage when indexes are unavailable due to maintenance. As time goes by, we gradually switch to more efficient SP indexes for higher query throughput when they finish updating in the latter stages. Therefore, our basic idea is to develop a multi-stage shortest path index (*e.g.,* PostMHL in Figure 1-(c)) to achieve efficient index maintenance and continuously improved query efficiency, enabling high throughput query processing on road networks of varying scales. To make our idea applicable, the following challenges need to be addressed: 1) How to aggregate multiple shortest path indexes into one system, given that they have independent procedures for index construction, query processing, and index update; 2) How to scale up the shortest path index for large road networks so that both city-level and nation-level queries can be efficiently processed; 3) How to empower the partitioned SP index with query efficiency equivalent to their non-partitioned ones for higher query throughput.

Contributions. In this paper, we tackle the above challenges and make the following our contributions:

- To our best knowledge, we are the first to formally investigate the problem of high throughput shortest path query processing on dynamic road networks. To solve this problem, we first analyze the essential relationships between DCH and DH2H, based on which we propose basic *Multi-stage Hierarchical hub Labeling (MHL)* to subtly aggregate BiDijkstra, DCH, and DH2H into one system to exploit their advantages for high query throughput.
- We propose a non-trivial *Partitioned MHL (PMHL)* index that leverages various partitioned shortest path (PSP) strategies and thread parallelization to improve the query throughput and scalability of MHL. In particular, we optimize existing *no-boundary* and *post-boundary* PSP strategies for faster index maintenance and propose a novel *cross-boundary* strategy to eliminate the distance concatenation for efficient query processing.
- We provide an insightful analysis of the equivalent of partitioned and non-partitioned SP index and then propose a novel *Post-partitioning MHL (PostMHL)* adopting tree decomposition-based graph partitioning to achieve equivalent query efficiency with DH2H but with much faster index update, thus further enhancing the query throughput of PMHL.
- Extensive experiments on real-world datasets demonstrate our methods' superiority over state-of-the-art baselines. PostMHL can yield up to 1-4 orders of magnitude higher query throughput.



(a) Example Graph G (b) Contraction Hierarchies G_{SC} (c) Tree Decomposition T

Figure 2: Road Network G, Contraction Hierarchies $\mathcal{G}$, and T

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 $|e(u, v)| \geq 0$. 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 (*e.g., degree, closeness* [21], or *betweenness* [22]) 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 *distance* of $sp_G(s, t)$ is denoted as $d_G(s, t)$. We omit the notation G when the context is clear. In practice, both the topological structure (vertex/edge insertion/deletion) and edge weight (increase/decrease) evolve over time. This paper concentrates on edge weight changes since topological change can be transformed into weight change [62, 64]. In the following, we will introduce the building blocks of our methods. Some important notations of this paper are listed in Table 1.

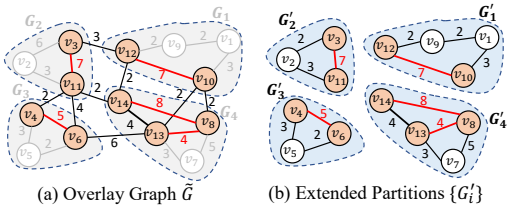
Table 1: Important Notations

Notation	Meaning	Notation	Meaning
G_i, G_i'	Partition i, extended partition i	B_i, B_i'	Boundary vertices of G_i and G_i'
L_i, L_i', L^*	No-, post-, cross-boundary index	$\tilde{G}, \tilde{L}$	Overlay graph and its index
$X(v)$	v's tree node	$X(v).A$	Ancestors of $X(v)$
$X(v).N$	Neighbor vertices of v in $X(v)$	$X(v).ch$	Children set of $X(v)$
$X(v).sc$	Shortcut array of v	$X(v).pos$	Position array of v
$X(v).dis$	Distance array of v	$X(v).disB$	Boundary array of v

2.1 Contraction Hierarchy

Contraction Hierarchy (CH) [24] is a hierarchical method for shortest path computation with two versions: CH with pruning (CH-P) [24, 66] and CH without pruning (CH-W) [48, 66]. They both 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. Differently, CH-W directly adds all-pair shortcuts among the neighbors of v_i while CH-P performs Dijkstra's search to decide whether to add those shortcuts or not. When processing a query, CH performs a bidirectional Dijkstra's search [45] on $\mathcal{G}$ in a bottom-up manner (*i.e.,* search from low-order vertices to high-order vertices) to reduce the search space. As evident in [66], CH-W has comparable query efficiency, faster index construction, and faster maintenance efficiency than CH-P. Therefore, the CH discussed in this paper is referred to as CH-W, and we denote its dynamic version as *DCH* [48].

Example 1. *Figure 2-(a) shows an example road network G with the vertex id as the vertex order,* i.e.,$r(v_1) < r(v_2) < ... < r(v_{14})$. *Figure 2-(b) demonstrates the result of contraction hierarchies $\mathcal{G}$, where*



(a) Overlay Graph $\tilde{G}$ (b) Extended Partitions $\{G'_i\}$

Figure 3: Overlay Graph and Extended Partitions

the red lines are newly added shortcuts. For instance, the shortcut $sc(v_9, v_{10}) = 5$ is generated during the contraction of v_1.

2.2 Hierarchical 2-Hop Labeling

Before introducing *Hierarchical 2-Hop Labeling (H2H)* [47], we first introduce its building block: tree decomposition.

Definition 1 (Tree Decomposition). *A Tree Decomposition of G, denoted as T, is a tree composed of tree nodes*[1] *(i.e., 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 [40, 47]:*

(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 $tw(T) = \max_{v \in V} |X(v)| - 1$. The depth of a tree node $X(v)$ is denoted as $X(v).h$ and the tree height of T is $h(T)$. We obtain T through *minimum degree elimination (MDE)* [10, 58]. 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}. 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 the all-pair edges formed in the MDE process as the *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 $X(v).A$. Figure 2-(c) presents the MDE-based tree decomposition T with $tw(T) = 3$ and $h(T) = 7$.

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, since H2H is a 2-hop labeling [13], $\forall s, t \in V$, suppose X is the *Lowest Common Ancestor (LCA)* [9] 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 [47] and its variant P2H [12] achieve state-of-the-art query efficiency on road networks. The dynamic version of H2H is denoted as *DH2H* [62].

2.3 Partitioned Shortest Path Index

Graph partitioning [15, 32] is a widely used method for scaling up graph algorithms to large-scale networks. In recent years, partitioned shortest path (PSP) index has also been explored to enhance the scalability in terms of indexing time [36, 42, 43], index size [37, 44, 54], or even index update time [67]. In particular, a road network G is first divided into multiple subgraphs $\{G_i|1 \leq i \leq k\}$ by graph partitioning algorithms such that $\bigcup_{i \in [1,k]} V(G_i) = V, V(G_i) \cap V(G_j) = \phi (\forall i \neq j, i, j \in [1, k])$. 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 *Partitioned Shortest Path strategy* for index construction, query, and update [67]. A straightforward solution (called *Pre-boundary strategy* [67]) for the PSP index is to precompute the global distances between all-pair boundary shortcuts $\{d(b_{i1}, b_{i2})|b_{i1}, b_{i2} \in B_i, b_{i1} \neq b_{i_2}\}$ for each partition G_i, and then insert them into both $\tilde{G}$ to preserve global distances and G_i to form an *extended partition* G'_i. Since the pre-computation of shortcuts is conducted by Dijkstra's search, it is time-consuming when the boundary vertex number is large. To avoid the slow pre-computation phase, [67] proposes two novel PSP strategies - *No-boundary* and *Post-boundary* - for efficient index maintenance and query processing.

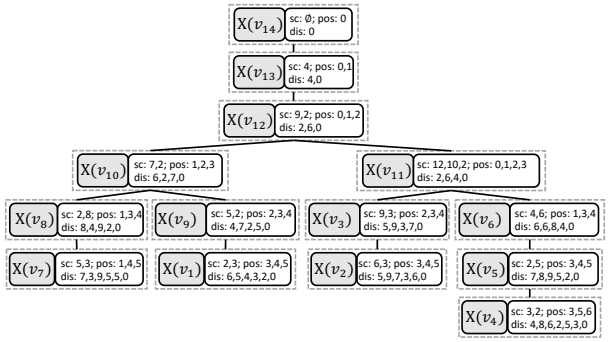


Figure 4: MHL Index for Example Graph G

Example 2. *Figure 3 shows the partition results $\{G_1, G_2, G_3, G_4\}$ of G in Example 1, with boundary vertices depicted in orange. Figure 3-(a) shows the overlay graph $\tilde{G}$ constructed by the pre-boundary strategy, with pre-computed all-pair shortcuts depicted in bold and newly-added shortcuts in red. Extended partition graphs $\{G'_1, G'_2, G'_3, G'_4\}$ are shown in Figure 3-(b), with red edges representing newly-added shortcuts that preserve the global distances.*

2.4 Problem Statement

We formally define the problem of *High Throughput Shortest Path query processing on dynamic road networks (HTSP)* as follows.

Problem Statement. Given a weighted dynamic road network $G = (V, E)$ with batches of edge weight update $\mathcal{U} = \{U_1, U_2, ...\}$ (where the update interval between neighboring batches is δt), and a continuous query stream $\mathcal{Q} = \{Q_1, Q_2, ...\}$, HTSP problem aims to maximize the query throughput Δ (*i.e.,* number of processed shortest path queries) within batch update interval δt.

3 BASIC SOLUTION: MHL-BASED SYSTEM

To solve the HTSP problem, we first propose a non-trivial *Multi-stage Hierarchical hub Labeling (MHL)* to exploit the fast index maintenance of DCH [48] and exceptional query efficiency of DH2H [62].

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

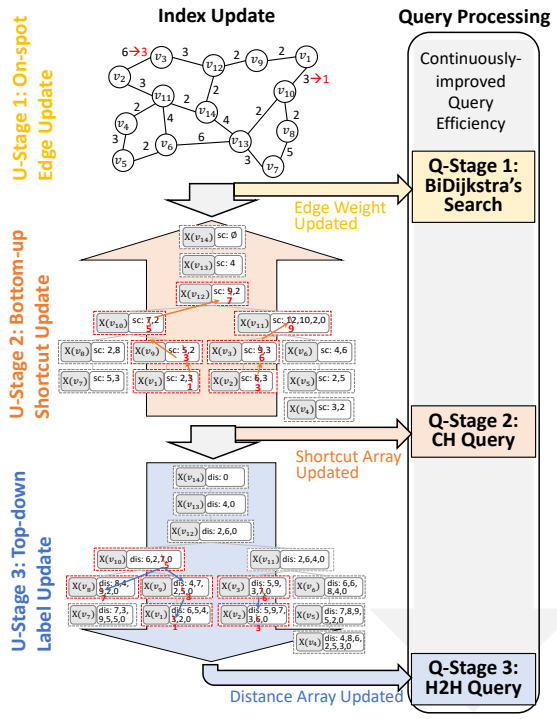


Figure 5: Illustration of MHL-based Path Finding System

3.1 MHL Index

Achieving efficient query processing and index maintenance simultaneously is challenging for any singleton path index, as a trade-off exists between these two factors. Nonetheless, we find a substantial correlation between the construction of CH and H2H, offering an integration opportunity to leverage their benefits.

Lemma 1. *The tree decomposition of H2H can generate equivalent shortcuts required by the CH construction.*

Proof. The proof is straightforward since CH-W and H2H have identical vertex contraction processes (both iteratively contract vertices according to the MDE method [10, 58]). □

Based on Lemma 1, we propose the *MHL* index, denoted as $L_{MHL} = \{X(v)|v \in V\}$, to integrate the CH and H2H for fast index update and query processing. Specifically, from the index structure's perspective, MHL improves the H2H index [47] by additionally maintaining a *shortcut array* $X(v).sc$ for every tree node $X(v)$, which stores shortcut lengths from v to vertices in $X(v).N$. The shortcut array in MHL enables CH query processing on the MHL index. Specifically, we conduct a bidirectional Dijkstra's search [45] on L_{MHL} by searching from low-order vertices to high-order vertices. Besides, each tree node in MHL also has the distance array $X(v).dis$ and position array $X(v).pos$ for H2H query processing.

Example 3. *Figure 4 illustrates the MHL index for example graph G. Each tree node $X(v)$ has three index types: the shortcut array $X(v).sc$, distance array $X(v).dis$, and position array $X(v).pos$. For instance, $X(v_{12}).sc = \{9, 2\}$ stores the shortcut lengths from v_{12} to its neighbors v_{13} and v_{14} during the MDE process, enabling CH search from v_{12}. The position array $X(v_{12}).pos = \{0, 1, 2\}$ holds the position IDs of v_{12} and its neighbors v_{13} and v_{14}, while $X(v_{12}).dis = \{2, 6, 0\}$ stores the distances from v_{12} to its ancestors v_{14}, v_{13}, and itself.*

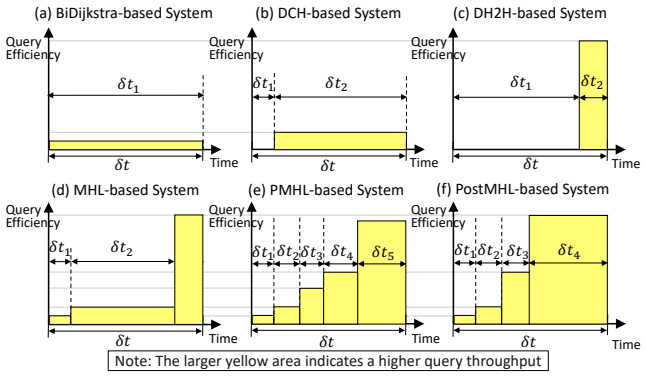


Figure 6: Query Throughput Illustration of Different Systems

3.2 MHL-based System Framework

To improve the query throughput of the DCH- or DH2H-based system, the MHL-based pathfinding system decouples the index maintenance into three stages and leverages the fastest available index after each stage for query processing. Figure 5-(a) illustrates the framework of the MHL-based system, which includes three stages for both Update (U-Stage) and Query (Q-Stage). After completing each U-Stage, the *query processing center* selects the fastest index to answer queries. As a result, MHL achieves a continuously improved query efficiency during index updates, resulting in a higher throughput compared with any single individual index.

U-Stage 1: On-spot Edge Update. Given a batch of edge weight updates, G's edges are modified directly in U-Stage 1. Then the index-free *BiDijkstra* algorithm [45] is applied for query processing in **Q-Stage 1**, as both CH and H2H are unavailable because they are outdated and need maintenance. We adopt BiDijkstra as it is the fastest point-to-point index-free algorithm that can be applied for any criteria of edge weights (*e.g.*, travel time, distance, cost, *etc.*).

U-Stage 2: Bottom-up Shortcut Update. In U-Stage 2, the shortcut arrays of L_{MHL} are maintained by the bottom-up shortcut update mechanism [62]. After that, *CH query processing* is immediately used in **Q-Stage 2** since it is faster than index-free methods.

U-Stage 3: Top-down Label Update. With the affected tree nodes of the smallest tree height obtained after U-Stage 2, the top-down label update mechanism [62] is adopted for the distance array update of MHL in U-Stage 3. Then *H2H query processing* becomes available in **Q-Stage 3**, as it achieves the fastest query efficiency for the MHL path-finding system.

Example 4. *As shown in Figure 5, given edge update batch $U_1 = \{e(v_2, v_3) : 6 \rightarrow 3, e(v_1, v_{10}) : 3 \rightarrow 1\}$, U-Stage 1 performs the edge updates instantly, facilitating BiDijkstra's algorithm for query processing. U-Stage 2 begins from the initially affected tree nodes $X(v_1)$ and $X(v_2)$ and employs the bottom-up shortcut update to propagate the updates, with the CH-based query processing available and released afterward. The affected tree nodes are highlighted in red. In U-Stage 3, the top-down label update initiates from the highest affected tree nodes $X(v_{12})$ and proceeds until all affected distance labels are updated, enabling the most efficient H2H-based query processing.*

Remark 1. To achieve high throughput, MHL adequately exploits the availability and advantage of BiDijkstra, DCH, and DH2H along the temporal dimension and integrates them into one system by

exploiting their intrinsic connections. As such, it can make full use of the index resources, processing as many queries as possible within a given period. As illustrated in Figure 6-(d), the throughput of MHL derives from three parts with increasing query efficiency: BiDijkstra's query processing in Q-Stage 1 → CH in Q-Stage 2 → H2H in Q-Stage 3. The duration of Q-Stage 1, δt_1, overlaps with that of U-Stage 2, while Q-Stage 2 with duration δt_2, happens concurrently with U-Stage 3. The duration of Q-Stage 3, δt_3, is dependent on the remain time within interval δt after U-Stage 3, *i.e.,* $\delta t_3 = \delta t - (\delta t_1 + \delta t_2)$. In brief, this strategic integration of BiDijkstra, DCH, and DH2H within the MHL yields high throughput compared to systems solely reliant on any one of them (Figure 6 (a)-(c)).

4 PMHL-BASED PATH FINDING SYSTEM

Although the MHL-based system can achieve higher throughput than DCH- or DH2H-based systems, its scalability is limited since MHL also inherits the time-consuming index maintenance of DH2H. As a result, the Q-Stage 3 of MHL may fail to be available when the update batch size or the road network is large, thus dramatically deteriorating the query throughput of MHL. To alleviate this problem, we further propose a novel *Partitioned MHL (PMHL)* index which leverages graph partition to improve the scalability.

4.1 PMHL Index

The basic idea of PMHL index is to utilize graph partitioning to divide G into multiple subgraphs and then build the MHL index on these partition graphs $\{G_i\}$ (or their variants) or overlay graph $\tilde{G}$ according to different PSP strategies. In this way, the index construction and maintenance among the partition indexes of PMHL can run in parallel, thus significantly improving update efficiency and scalability. Nevertheless, as observed in [67], the query efficiency of PSP indexes is slower than their non-partitioned counterparts due to distance concatenation between the overlay index and partition indexes. To overcome this limitation, we put forward a series of PSP strategies and subtly integrate them into the PMHL index to enhance query efficiency incrementally across five consecutive stages. By the last stage, PMHL achieves query efficiency that is nearly comparable to the H2H index.

In particular, there are three PSP strategies in PMHL, namely, *optimized no-boundary*, *optimized post-boundary*, and *cross-boundary* strategies. These strategies correspond to three types of indexes: *no-boundary indexes* $\{\tilde{L}, \{L_i\}\}$, *post-boundary partition indexes* $\{L'_i\}$, and *cross-boundary index* L^*. To construct the PMHL index, we first leverage the renowned graph partitioning method PUNCH [15] to divide G into k vertex-disjoint subgraphs $\{G_i | 1 \le i \le k\}$. Given the partition results as input, a *boundary-first* vertex ordering method [43] is leveraged to obtain the vertex order r of all vertices, which assigns the boundary vertices higher ranks than the non-boundary vertices, as the *cross-boundary strategy* needs it (see Section 4.2.2). After obtaining the vertex ordering, there are six main steps to obtain the index of various PSP strategies, the details of which will be introduced in Section 4.2.

4.2 PSP Strategies in PMHL

We next introduce the PMHL index construction and query processing from the perspective of PSP strategy.

4.2.1 ***Optimized No-boundary and Post-boundary Strategy.*** We first introduce the construction of the optimized no-boundary PMHL index, which has three main steps:

Step 1: Construct the partition index $\{L_i\}$ for all $\{G_i\}$ in parallel;

Step 2: Construct the overlay graph $\tilde{G}$ based on the shortcuts among the boundary vertices formed in the MDE process of Step 1;

Step 3: Construct the overlay index $\tilde{L}$ for $\tilde{G}$.

Compared to the original no-boundary strategy in [67], our optimized no-boundary strategy replaces the time-consuming all-pair boundary shortcuts computation by directly leveraging the shortcuts formed in the contraction of partition vertices for the overlay graph construction (Step 2). Such an optimization not only accelerates the no-boundary index construction by eliminating $\sum_{i\in[1,k]} |B_i| \times |B_i|$ queries on $\{L_i\}$, but also reduces the overlay graph size for faster overlay index maintenance. We next prove the correctness of this optimization in Theorem 1 and provide the no-boundary PMHL query processing in Lemma 2.

Theorem 1. *The overlay graph constructed by Step 2 preserves the global distance between any two boundary vertices in G.*

Proof. Given any boundary pair $s, t \in B_i$, we prove the distance $d_G(s,t)$ is preserved in the overlay graph by the following cases:

Case 1: $sp(s,t)$ does not pass through any non-boundary vertex. This case naturally holds since the contracted non-boundary vertices do not affect the distance.

Case 2: $sp(s,t)$ passes through at least one non-boundary vertex. Suppose a non-boundary vertex $u \in B_j$ lays in $sp(s,t)$, we take the concise form of $sp(s,t)$ by extracting only the boundary vertices and u as $sp_c = \langle s = b_0, \ldots, b_p, u, b_q, \ldots, b_n = t \rangle$ $(b_i \in B, 0 \le i \le n)$, where b_p and b_q are the boundary vertices of G_j. The contraction of u and other non-boundary vertices in G_j must lead to shortcut $sc(b_p, b_q)$ in step 1 of optimized no-boundary MHL. Therefore, the distance between b_p and b_q is preserved. □

Lemma 2. *For no-boundary query (Q-Stage 3 of PMHL), $\forall s \in G_i, t \in G_j, Q(s,t) =$*

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

According to Lemma 2, the optimized no-boundary strategy has to concatenate $\tilde{L}$ and $\{L_i\}$ to answer the same-partition queries (*i.e.,* $s, t \in G_i$), sacrificing the same-partition query efficiency. Note that the *same-partition queries* somehow can be regarded as city-level queries on large road networks. Therefore, it is significant to enhance the same-partition query efficiency. To this end, we further propose the optimized post-boundary strategy, which employs the same optimization as the optimized no-boundary strategy but consists of five main steps:

Step 1-3: Same with the optimized no-boundary strategy.

Step 4: Parallely compute $d_G(b_{i1}, b_{i2}), b_{i1}, b_{i2} \in B_i$ by $\tilde{L}$ for each partition G_i, and insert these all-pair shortcuts into G_i to get G'_i;

Step 5: Construct $\{L'_i\}$ on all extended partitions $\{G'_i\}$ in parallel.

Step 4 leverages $\tilde{L}$ to build the extended partitions $\{G'_i\}$ while Step 5 further constructs the correct partition indexes $\{L'_i\}$. Therefore, we process post-boundary PMHL queries by Lemma 3.

Algorithm 1: Cross-boundary PMHL Index Construction

1 **Function** *CrossIndexBuild*($\tilde{L}$, $\{L_i\}$):
2 // $X^*(v).h$ is $X^*(v)$'s tree height, A is the ancestor vector
3 $v \leftarrow$ vertex with the highest order; $X^*(v).h \leftarrow 1$; $A \leftarrow \{v\}$;
4 DFSCrossIndex(v, A, L^*); ▷ Build L^* in a top-down manner
5 **return** $L^* = \{X^*(v) | v \in V\}$;
6 **Function** *DFSCrossIndex*(v, A, L^*):
7 $X^*(v).A \leftarrow A$; $ch \leftarrow \phi$; //ch is the children set
8 **if** $v \in \tilde{G}$ **then**
9 $ch \leftarrow \tilde{X}(v).ch$; $X^*(v).pos \leftarrow \tilde{X}(v).pos$; ▷ Get position array
10 ComputeDisArray($v, A, \tilde{X}(v), L^*$); ▷ Compute distance array
11 **else**
12 ComputeDisArray($v, A, X_i(v), L^*$); //$X_i(v)$ is v's partition index
13 **for** $u \in X_i(v).ch$ **do**
14 **if** $u \notin \tilde{G}$ **then** $ch \leftarrow ch \cup u$;
15 **for** $u \in ch$ **do**
16 $X^*(v).ch \leftarrow u$; $X^*(u).h \leftarrow X^*(v).h + 1$;
17 $A.push\_back(v)$;
18 **for** $u \in \tilde{X}(v).ch$ **do**
19 DFSCrossIndex(u, A, L^*);
20 $A.pop\_back()$;
21 **Function** *ComputeDisArray*($v, A, X(v), L^*$):
22 **for** $i = 1$ *to* $|A| - 1$ **do**
23 $d \leftarrow \infty$; $u \leftarrow A[i]$;
24 **for** $j = 1$ *to* $|X(v).N|$ **do**
25 $w \leftarrow X(v).N[j]$, $d_1 \leftarrow X(v).sc[j]$;
26 **if** w is the ancestor of u **then** $d_2 \leftarrow X^*(u).dis[X^*(w).h]$;
27 **else** $d_2 \leftarrow X^*(w).dis[i]$;
28 **if** $d > d_1 + d_2$ **then** $d \leftarrow d_1 + d_2$;
29 $X^*(v).dis[i] \leftarrow d$;
30 $X^*(v).dis[|A|] \leftarrow 0$;

Lemma 3. *For post-boundary query (Q-Stage 4 of PMHL),* $\forall s \in G_i, t \in G_j, Q(s,t) =$

$$\begin{cases} d_{L'_i}(s,t) & i = j \\ d_{\tilde{L}}(s,t) & i \neq j \text{ and } s,t \in B \\ \min\limits_{b_q \in B_j} \{d_{\tilde{L}}(s,b_q) + d_{L'_j}(b_q,t)\} & i \neq j, s \in B, t \notin B \text{ (v.v.)} \\ \min\limits_{b_p \in B_i, b_q \in B_j} \{d_{L'_i}(s,b_p) + d_{\tilde{L}}(b_p,b_q) + d_{L'_j}(b_q,t)\} & i \neq j \text{ and } s \notin B, t \notin B \end{cases}$$

According to Lemma 3, the same-partition queries can be efficiently answered by L'_i. Nonetheless, the optimized post-boundary PMHL index remains slow in *cross-partition queries* (*i.e.,* $s \in G_i, t \notin G_i$, which can be regarded as nation-level queries) as it requires distance concatenation between overlay index and partition indexes.

4.2.2 ***Cross-boundary Strategy.*** Both no-boundary and post-boundary strategies suffer from lower query efficiency for cross-partition queries due to the distance concatenation. To solve this problem, we further propose a novel *cross-boundary strategy*. Its key idea is to build the global non-partitioned index by concatenating the overlay index and partition indexes in advance so that the cross-partition distance concatenation is eliminated during the querying process. We build the cross-boundary index as follows:

Step 1-5: Same with the optimized post-boundary strategy.

Step 6: Parallelly compute the cross-boundary index L^* for all partitions. In particular, for partition G_i, we precompute the distance labels from the vertex $v \in G_i$ to the union of the hubs of all the boundary vertices in B_i by concatenating the overlay index and partition indexes, *i.e.,* $L^*(v,u) = \min_{b \in B_i}\{d_{L_i}(v,b) + d_{\tilde{L}}(b,u)\}, \forall u \in \bigcup_{b \in B_i} \tilde{L}(b)$.

The pseudo-code of cross-boundary PMHL index construction is demonstrated in Algorithm 1, which constructs the cross-boundary labels in a top-down manner using the DFSCrossIndex function (Lines 6-20). Notably, the process begins from the highest-order vertex, with an ancestor vector A as the input parameter (Lines 3-4). For each vertex v being processed, we compute its children set ch by integrating the overlay tree and partition trees (Lines 7-16). Specifically, we initialize ch with the children set of v's overlay index if v is an overlay index (Line 9), or with an empty set otherwise. We also add a vertex u from the children set of v's partition index to ch if u is not an overlay vertex (Lines 13-14). This integration strategy prioritizes the node relationships of the overlay tree, ensuring that the LCA of the L^* aligns with $\tilde{L}$'s LCA. Therefore, we can directly inherit the position array from $\tilde{L}$ for the overlay vertices (Line 9). The cross-boundary tree also inherits the subordinate relationships of the overlay and partition trees. The distance array of v is computed in ComputeDisArray function by using the neighbor set of v's overlay index or partition index as the vertex separator (Lines 21-30). We next introduce the important *boundary first property* for cross-boundary strategy and prove its correctness for cross-partition query processing.

Property 1. ***(Boundary First Property)*** *The cross-boundary strategy requires that the boundary vertex has a higher vertex order than its non-boundary neighbors.*

Theorem 2. *The L^* constructed by Algorithm 1 can correctly answer cross-partition queries since it fulfill the boundary first property.*

Proof. The proof is in Appendix [2] due to limited space. □

According to Theorem 2, we can answer cross-partition queries by H2H-style query processing, resulting in the cross-boundary PMHL query processing in Lemma 4.

Lemma 4. *For cross-boundary query (Q-Stage 5 of PMHL),* $\forall s \in G_i, t \in G_j, Q(s,t) =$

$$\begin{cases} d_{L'_i}(s,t) & i = j \\ d_{\tilde{L}}(s,t) & i \neq j \text{ and } s,t \in B \\ d_{L^*}(s,t) & i \neq j \text{ and } s \text{ or } t \notin B \end{cases}$$

According to Lemma 4, the cross-boundary query processing eliminates the distance concatenation of cross-partition queries, thus greatly improving the query efficiency.

4.3 PMHL-based System Framework

In this section, we introduce the framework of PMHL-based system and the index maintenance of PMHL. As presented in Figure 7, there are five stages for update and query. After each update stage, the PMHL-based system exploits the fastest available index for querying to achieve continuously improved query efficiency.

U-Stage 1: On-spot Edge Update. Given a batch of updates, PMHL directly modifies the edge weights on G. After that, the index-free *BiDijkstra's search* is leveraged for query processing in **Q-Stage 1**. Besides, the updated edges are classified into two types: inter-edge E_{inter} for the overlay index update and intra-edge E_{intra} for the partition index update for the next update stage.

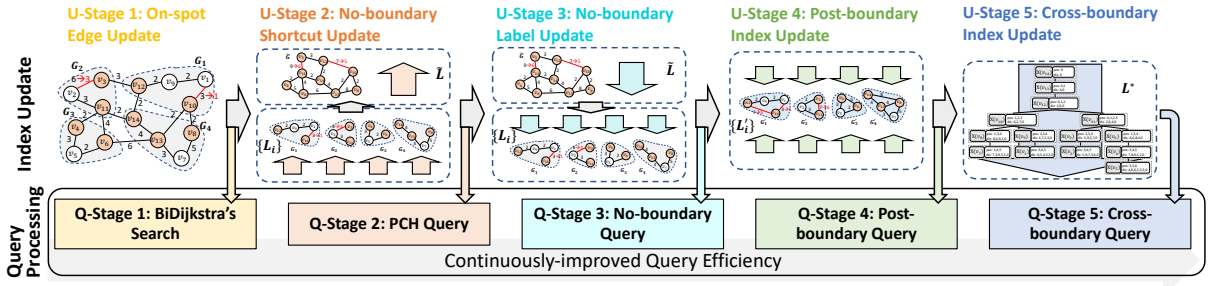


Figure 7: Illustration of PMHL-based Path Finding System

U-Stage 2: No-boundary Shortcut Update. During this stage, the shortcut arrays of the no-boundary index are updated by the bottom-up shortcut update mechanism [62]. In particular, we first identify the affected partitions by the intra-edge E_{intra} and then maintain these affected partition indexes $\{L_i\}$ in parallel. Note that the affected overlay shortcuts are identified and inserted into E_{inter} for the overlay index update during the shortcut update of $\{L_i\}$. Given E_{inter} as input, we next update the overlay index $\tilde{L}$ if the overlay graph edges change (E_{inter} is not empty). Both the partition shortcut and overlay shortcut output the highest affected tree node for the next stage's update. After the no-boundary shortcut updates, we implement the *PCH query* [67] on the shortcut arrays of $\{L_i\}$ and $\tilde{L}$ in **Q-Stage 2**. Note that although PCH searches on the union of the shortcut arrays of partition and overlay indexes [67], it is essentially equivalent to the CH query.

U-Stage 3: No-boundary Label Update. Given the highest affected tree nodes identified in U-Stage 2 as input, the distance arrays of the overlay and partition indexes are concurrently maintained via the top-down label update mechanism [62]. Subsequently, overlay index $\tilde{L}$ and partition index $\{L_i\}$ are fully updated, enabling faster *no-boundary query processing* in **Q-Stage 3**. Additionally, to identify the update area in U-Stage 5, we store the vertices whose labels are updated in an affected vertex set.

U-Stage 4: Post-boundary Index Update. In this stage, we first check whether the all-pair boundary shortcuts of extended partitions have been changed by querying over updated $\tilde{L}$. If boundary shortcut updates exist, the corresponding post-boundary partition indexes $\{L'_i\}$ are concurrently updated using both bottom-up shortcut and top-down label update mechanisms. As a result, the same-partition queries are accelerated through the *post-boundary query processing* in **Q-Stage 4**.

U-Stage 5: Cross-boundary Index Update. In the last stage, the PMHL index updates the distance arrays of the cross-boundary index L^*. In particular, we first identify the affected partitions G_A based on the affected vertex set generated in U-Stage 3. We select a representative vertex v_j with the highest rank for each affected partition $G_j \in G_A$. Next, we remove the vertices that are descendants of other representative vertices in the cross-boundary tree, leaving only the branch roots of all affected vertices (denoted as V_R). Lastly, sourcing from these root vertices in V_R, we maintain the cross-boundary index in a top-down manner. The update of L^* facilitates the most efficient *cross-boundary query processing* that has nearly comparable efficiency to H2H during **Q-Stage 5**.

Remark 2. As illustrated in Figure 6-(e), compared to the MHL-based system, the advantages of the PMHL lie in two folds. Firstly, it uses thread parallelization among partitions to achieve faster index construction and maintenance, resulting in better scalability. This also greatly enhances query throughput as the efficient Q-Stage 5 of PMHL can be released much faster than Q-Stage 3 in MHL. Secondly, it has more intermediate stages (Q-Stages 3 and 4) than the MHL-based system, which leads to much faster query processing than MHL when Q-Stage 5 of PostMHL cannot be released before the next update batch. Therefore, the PMHL significantly enhances the query throughput and scalability of MHL.

5 POSTMHL-BASED PATH FINDING SYSTEM

Despite the significant advancements made by PMHL in enhancing the query throughput and scalability of MHL, we observe that the cross-boundary query of PMHL is generally slower than H2H. To further enhance query throughput, in this section, we first provide an insightful analysis of the essence of the PSP index, based on which we propose a non-trivial *Post-partitioning MHL (PostMHL)* index to unify MHL and PMHL, achieving equivalent query efficiency with H2H but with a much faster update.

5.1 The Essence of Partitioned SP Index

The cross-boundary query of PMHL is essentially an H2H query processing. However, its query efficiency is generally worse than H2H. To explain the phenomenon, we reveal the essence of the partitioned SP index through the following theorem:

Theorem 3 (The Equivalent of SP and PSP Index). *The PSP index essentially generates a vertex ordering that fulfills the boundary-first property. The non-partitioned SP index based on this ordering is equivalent to its cross-boundary PSP index counterpart.*

Proof. Given $Q(s,t), \forall s \in G_i, t \in G_j$, the PSP index query processing relies on the concatenation of distance from s to its boundary vertices $b_p \in B_i$ and distance from t to its boundary vertices $b_q \in B_j$ and distance from b_p to b_q. This requires all non-boundary vertices to store the distances to their boundary vertices, essentially setting a higher vertex order for the boundary vertices. Hence, ordering generated by the PSP index naturally fulfills the boundary-first property. Since the relative order of vertices determines the index structure, the ordering generated by the PSP index should also maintain the relative order among the partition vertices for

Algorithm 2: Tree Decomposition-based Partitioning

```
1  Function TDPartition(T, τ, k_e, β_l, β_u):
2      // Compute all vertices' descendent number
3      cN[v] ← 1, ∀v ∈ V; //cN is partition size vector
4      for X(v) ∈ T in a bottom-up manner do
5          for u ∈ X(v).ch do
6              cN[v] ← cN[v] + cN[u];
7      // Compute root vertex candidates
8      V_C ← ϕ; // V_C is a vector storing the root candidates
9      for v ∈ V in decreasing vertex rank do
10         if β_l · |V|/k_e ≤ cN[v] ≤ β_u · |V|/k_e and |X(v).N| ≤ τ then
11             V_C.push_back(v);
12     // Finalize the root vertices by minimum overlay strategy
13     for v ∈ V_C do
14         if ∀u ∈ V_R, u is not the ancestor of v then
15             V_R ← V_R ∪ v;
16     G̃, {G_i} ← GetPartitions(V_R);          ▷ Get graph partition results
17     return G̃, {G_i};
```

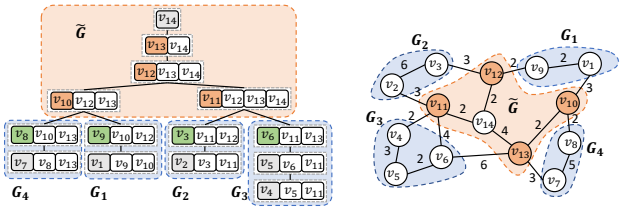
each partition, as well as the relative order among the overlay vertices, to ensure the index equivalence between the non-partitioned SP index and its cross-boundary PSP index counterpart. □

Analysis. Following Theorem 3, the cross-boundary PSP index could have the equivalent query efficiency with its non-partitioned SP index counterpart, and its query efficiency relies on boundary-first vertex ordering r generated by the PSP index. Because the traditional graph partitioning methods are not designed towards a good partitioned SP index, *i.e.,* a good vertex ordering, there is no guarantee for their index performance. Therefore, their query performance is generally worse than that of their non-partitioned counterparts, which often utilize a good vertex ordering. In our scenario, the PMHL tends to have larger tree height and treewidth than the H2H constructed with MDE, so its query is slower.

5.2 From Tree Decomposition to Partitioning

Since graph partitioning cannot guarantee a satisfactory ordering, then can we obtain graph partitions from a good vertex ordering such that PMHL could leverage this partition result to achieve equivalent cross-boundary query efficiency to its non-partitioned counterpart? The answer is affirmative because tree decomposition shares the same goal as graph partitioning: to find vertex separators (boundary vertices) that can divide a graph into two (or multiple) subgraphs. In particular, the neighbor set of any tree node in T, such as $X(u).N$, can serve as a vertex separator for the descendants of $X(u)$ and all other tree nodes. Inspired by this, we propose a novel *Tree Decomposition-based Partitioning (TD-Partitioning)* method. Its key idea is to choose one root vertex u for each partition G_i, setting u and its descendants as the *in-partition vertices* and its neighbor set $X(u).N$ as the boundary vertices. The ancestors of all root vertices (including the boundary vertices of all partitions) are then treated as vertices of the overlay graph for connecting all partitions.

Algorithm 2 presents the pseudo-code of TD-partitioning. The input includes the tree decomposition $T = \{X(v)|v \in V\}$, bandwidth τ, expected partition number k_e, lower and upper partition size bound ratio $\beta_l, \beta_u \in (0, k_e]$. The bandwidth τ aims to limit the boundary vertex number for all partitions, as this significantly



(a) Graph Partitioning on T (b) Overlay Graph and Partition Graphs

Figure 8: Example of Tree Decomposition-based Partitioning

affects the cross-partition query efficiency of no-boundary and post-boundary strategies [67]. On the other hand, β_l and β_u constrain the imbalance of partition sizes such that workloads among threads are more balanced. Firstly, we calculate the descendent number for each vertex $v \in V$ to obtain the in-partition vertex number if v is assigned as the root vertex (Lines 3-6). In particular, we first initiate the partition size of all vertices as 1 (Line 3) and then compute the descendent number in a top-down manner from the root of T (Lines 3-6). Secondly, we calculate the root vertex candidates based on the constraints of bandwidth and partition size bounds (Lines 8-11). For instance, if the neighbor set size of vertex v, $|X(v).N|$, is no larger than the bandwidth τ, and its partition size falls between $\beta_l \cdot \frac{|V|}{k_e}$ and $\beta_u \cdot \frac{|V|}{k_e}$, then we add v to the candidate set V_C. After obtaining the root candidates V_C, we use a ***minimum overlay strategy*** to determine the root vertex for all partitions, obtaining the root vertex set V_R (Lines 12-15). The minimum overlay strategy aims to minimize the overlay graph size since overlay index update time cannot be paralleled, thus enhancing the overall index maintenance efficiency. Lastly, in the function GetPartitions, we directly obtain the partition graph by treating the root vertex's tree node and its descendants as the partition vertices and the ancestors of all root vertices as vertices of the overlay graph (Line 16).

The TD-partitioning inherently fulfills Property 1 as the neighbor set of root vertex has higher vertex orders. Moreover, it allows the partitioned shortest path index to utilize the vertex ordering generated by MDE-based tree decomposition. Thus, cross-boundary queries can achieve the same query efficiency as the H2H index.

Example 5. *Given tree decomposition T and $\tau = 2, \beta_l = 2, \beta_u = 4$, Figure 8-(a) illustrates its TD-partitioning, which consists of four partitions G_1 to G_4 and an overlay graph $\tilde{G}$. The root vertices (green) of these partitions are v_9, v_3, v_6, v_8, and their boundaries are $\{v_{11}, v_{12}\}$, $\{v_{10}, v_{12}\}$, $\{v_{11}, v_{13}\}$, and $\{v_{10}, v_{13}\}$. Figure 8-(b) shows the partition result with orange vertices representing the boundary vertices.*

5.3 PostMHL Index

With the TD-partitioning method, the cross-boundary index now has better query efficiency. However, updating the PSP index is still laborious due to the sequentiality between PSP strategies. Fortunately, instead of designing the partition index from the partition's perspective like PMHL, which utilizes no-boundary or post-boundary strategies to guarantee correctness, Theorem 3 and Algorithm 2 provide us an opportunity to re-design it reversely from the complete index's perspective and further incorporate the partition strategies for faster maintenance. In particular, we discard the no-boundary strategy for the PostMHL index and unify post-boundary

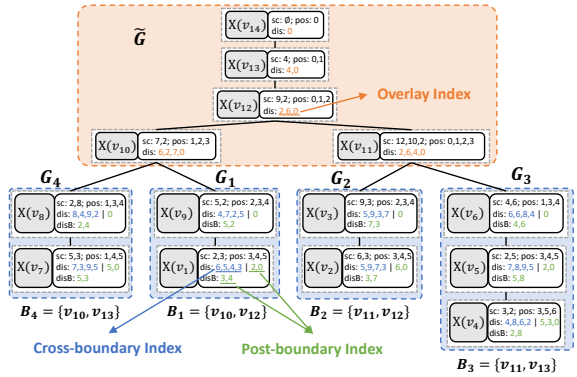


Figure 9: PostMHL Index

and cross-boundary strategies into one tree decomposition to improve the index update efficiency.

The PostMHL has three index types: *overlay index*, *post-boundary index*, and *cross-boundary index*. To better illustrate the index component, we classify the ancestors of each tree node into two types: *overlay ancestors* that belong to $\tilde{G}$ and *in-partition ancestors* otherwise. Besides, compared with MHL, each in-partition node $X(v)$ in PostMHL has an additional data structure called *boundary array* $X(v).disB$, which stores the distances from $v \in G_i$ to all boundary vertices $b_p \in B_i$. As shown in Figure 9, the overlay index consists of the distance arrays of all overlay vertices. While post-boundary and cross-boundary indexes exist in the in-partition vertices. In particular, the post-boundary index is composed of the distance array entries to the in-partition ancestors and boundary arrays, while the cross-boundary index is made up of the distance array entries to the overlay ancestors. Under this structure, we are safe to drop the no-boundary strategy because the indexing and update of the post-boundary and cross-boundary indexes depend only on the overlay index, as proved in Theorem 4.

Theorem 4. *The overlay index is sufficient for constructing and maintaining the post-boundary and cross-boundary index of PostMHL.*

Proof. The proof is in Appendix [2] due to limited space. □

Example 6. *Figure 9 presents the PostMHL index on tree decomposition T, where the overlay index, post-boundary index, and cross-boundary index are depicted in orange, green, and blue colors, respectively. For instance, the distance array of partition vertex v_1 is divided into two parts: the entries of overlay ancestors $\{6, 5, 4, 3\}$ for the cross-boundary index and the entries of in-partition ancestors $\{2, 0\}$ for the post-boundary index. The post-boundary index also includes the boundary array $X(v_1).disB = \{3, 4\}$, which stores the distance from v_1 to border vertices v_{10}, v_{12}. The overlay index of v_{12} comprises the distance array $X(v_{12}).dis = \{2, 6, 0\}$.*

Algorithm 3 showcases the PostMHL index construction. The MDE-based tree decomposition is firstly conducted to obtain T and shortcut arrays for all tree nodes (Line 1). The TD-partitioning (Algorithm 2) is then employed to obtain the graph partition results (Line 2). The distance arrays of the overlay vertices are built in a top-down manner like the MHL index (Line 3). After getting the overlay index, we build the post-boundary index for all partitions in parallel (Lines 5-31). In particular, given the partition G_i, we first compute the all-pair distances among the boundary vertices

Algorithm 3: PostMHL Index Construction

Input: Road network G, bandwidth τ, expected partition number k_e, lower and upper bound of partition size β_l, β_u
Output: PostMHL index $L = \{X(v) | v \in V\}$

1 $T \leftarrow$ TreeDecomposition(G); ▷ MDE-based tree decomposition
2 $\tilde{G}, \{G_i | i \in [1,k]\} \leftarrow$ TDPartition($T, \tau, k_e, \beta_l, \beta_u$); ▷ TD-partitioning
3 $L \leftarrow$ OverlayIndexing($\tilde{G}$); ▷ Top-down Overlay index construction
4 // Post-boundary index construction
5 **parallel_for** $i \in [1,k]$
6 $u \leftarrow$ root vertex of G_i;
7 **for** $b_1 \in X(u).N$ *and* $b_2 \in X(u).N$ **do**
8 $D[b_1][b_2] \leftarrow Q(b_1, b_2)$; ▷ Get all-pair boundary distances
9 **for** $v \in G_i$ *and* $X(v) \in T$ *in a top-down manner* **do**
10 Suppose $X(v).N = (x_1, x_2, \ldots)$;
11 **for** $j = 1$ *to* $|X(v).N|$ **do**
12 $X(v).pos[j] \leftarrow$the position of x_j in $X(v).A$;
13 // Compute boundary array
14 **for** $j = 1$ *to* $|X(u).N|$ **do**
15 $X(v).disB[j] \leftarrow \infty$;
16 **for** $k = 1$ *to* $|X(v).N|$ **do**
17 **if** $x_k \in \tilde{G}$ **then** $d \leftarrow X(v).sc[k] + D[x_j][x_k]$;
18 **else** $d \leftarrow X(v).sc[k] + X(x_k).disB[j]$;
19 $X(v).disB[j] \leftarrow \min\{X(v).disB[j], d\}$;
20 // Compute distance array
21 **for** $j = 1$ *to* $|X(v).A| - 1$ **do**
22 **if** $X(v).A[j] \in \tilde{G}$ **then** continue; ▷ Prune overlay ancestors
23 $c \leftarrow X(v).A[j]; X(v).dis[j] \leftarrow \infty$;
24 **for** $k = 1$ *to* $|X(v).N|$ **do**
25 **if** $x_k \in \tilde{G}$ **then**
26 $l \leftarrow$ position of x_k in $X(u).N$; $d \leftarrow X(c).disB[l]$;
27 **else**
28 **if** $X(v).pos[k] > j$ **then** $d \leftarrow X(x_k).dis[j]$;
29 **else** $d \leftarrow X(c).dis[X(v).pos[k]]$;
30 $X(v).dis[j] \leftarrow \min\{X(v).dis[j], X(v).sc[k] + d\}$;
31 $X(v).dis[|X(v).A|] \leftarrow 0$;
32 **parallel_for** $i \in [1,k]$
33 CrossPartiIndexing(G_i, L); ▷ Cross-boundary index construction
34 **return** L;

of G_i based on the overlay index and then store the results in a map table D for later reference (Lines 6-8). Sourcing from the root vertex, the post-boundary index, including the boundary array and distance array entries of in-partition ancestors, is conducted in a top-down manner (Lines 9-31). Lastly, the cross-boundary index is constructed in parallel in a top-down manner from each partition's root vertex, similar to the overlay index construction (Lines 32-33).

Theorem 5. *The space complexity of PostMHL is $O(n \cdot h + n_p \cdot \tau)$ while the time complexity is $O(n \cdot (w^2 + \log(n)) + n_o \cdot (\log(n_o) + \tilde{h} \cdot w) + \frac{n_p}{k} \cdot (\log(n_p) + h \cdot \tau))$, where h is the tree height of T, $\tilde{h}$ is the maximum tree height of overlay vertices in T, τ is bandwidth, w is treewidth, k is partition number, n_o and n_p are the overlay and in-partition vertex number, respectively.*

Proof. The proof is in Appendix [2] due to limited space. □

5.4 Framework of PostMHL-based System

We next introduce the framework of PostMHL-based path-finding system. As illustrated in Figure 10, there are five main stages for PostMHL index update. The U-Stage 1 and 2 of PostMHL are the same as those of PMHL. The U-Stage 3 of PostMHL is equivalent

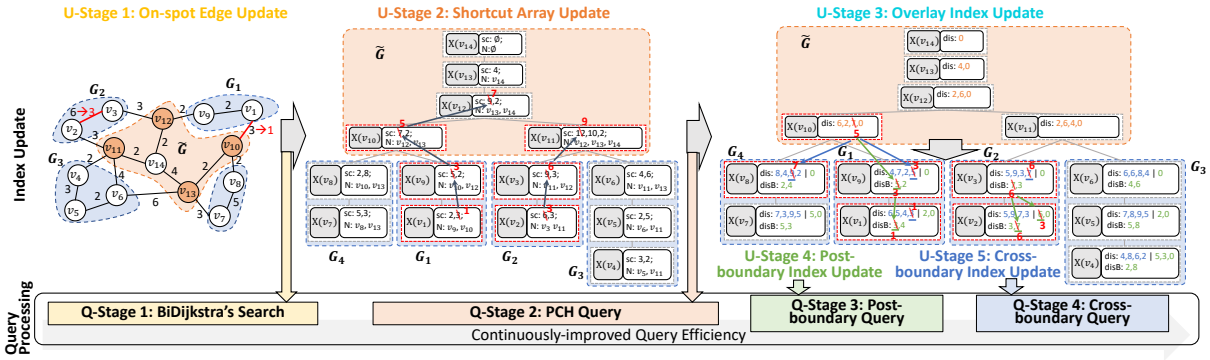


Figure 10: Illustration of PostMHL-based Path Finding System

to the overlay label update of U-Stage 3 in PMHL. Note that during the update of U-Stage 2 and U-Stage 3, we record the in-partition vertices whose labels are updated in an affected vertex set for each partition to facilitate the post-boundary index update in U-Stage 4 and cross-boundary index update in U-Stage 5. The post-boundary query is used in Q-Stage 3 after the post-boundary index update of U-Stage 4. After the update of U-Stage 5, the cross-boundary query with the same query efficiency as H2H is available in Q-Stage 4. It is worth noting that, based on Theorem 4, the post-boundary (U-Stage 4) and cross-boundary (U-Stage 5) index updates can be conducted in parallel after the overlay index maintenance (U-Stage 3), which further increases the throughput. Nevertheless, updating a cross-boundary index generally takes more time than updating a post-boundary index due to the high tree height of the overlay index. Therefore, having a post-boundary index is crucial for improving query throughput, as it becomes available earlier.

Theorem 6. *The index update complexity of edge decrease in PostMHL is* $O(w \cdot (\delta + \Delta\tilde{h} + \frac{\Delta h_p}{k}))$ *while edge increase is* $O(w \cdot (\delta + (\Delta\tilde{h} + \frac{\Delta h_p}{k}) \cdot (w + \epsilon)))$*, where* δ *is the affected shortcut number,* $\Delta\tilde{h}$ *and* Δh_p *are the affected tree height in* $\tilde{G}$ *and* $\{G_i\}$*,* ϵ *is the maximum number of nodes in the subtree involving the affected node.*

Proof. The proof is in Appendix [2] due to limited space. □

Example 7. *Figure 10 also demonstrates PostMHL's index update. U-Stage 1 of PostMHL refreshes the edge weights on-spot when a batch of updates* ($U_1 = \{e(v_2, v_3) : 6 \rightarrow 3, e(v_1, v_{10}) : 3 \rightarrow 1)\}$) *comes, and BiDijkstra's is used for query at first. In U-Stage 2, the in-partition shortcuts are first updated in parallel. For instance, the update of* $sc(v_1, v_{10})$ *propagates to* $sc(v_9, v_{10})$ *in* G_1 *while the update of* $sc(v_2, v_3)$ *propagates to* $sc(v_3, v_{11})$ *in* G_2*. The affected overlay shortcuts are recorded and passed to the overlay update. Consequently, all affected shortcuts* ($sc(v_{10}, v_{12}), sc(v_{11}, v_{12}), sc(v_{12}, v_{13})$) *are refreshed to the correct values in U-Stage 2, enabling the PCH for query in Q-Stage 2. U-Stage 3 starts with the overlay index update, e.g.,* $X(v_{10}).dis[3]$ *is updated from 7 to 5. After updating the overlay index, the post-boundary and cross-boundary index updates are performed simultaneously for in-partition vertices. For instance,* v_9*'s post-boundary index,* $X(v_9).disB[1]$*, is updated from 5 to 3, while* v_8*'s cross-boundary index,* $X(v_8).dis[3]$*, is updated from 9 to 7. After updating the post-boundary index, post-boundary query processing is available, while the fastest cross-boundary query is available after updating the cross-boundary index.*

Table 2: Real-world Datasets

Name	Dataset	$\|V\|$	$\|E\|$	k	τ
FLA	Florida	1,070,376	2,687,902	32	100
CAL	California	1,890,815	4,630,444	32	200
W	Western USA	6,262,104	15,119,284	32	300
CTR	Central USA	14,081,816	33,866,826	32	400
USA	Full USA	23,947,347	57,708,624	32	400

Remark 3. Figure 6-(f) demonstrates the query throughput of PostMHL. Compared to PMHL, PostMHL benefits from faster index maintenance by eliminating the no-boundary strategy and sharing the same tree decomposition among all indexes. As a result, the post-boundary and cross-boundary query processing can be released in advance. Furthermore, the efficiency of cross-boundary query processing is higher than PMHL due to the better vertex ordering. Therefore, the PostMHL-based system outperforms the PMHL-based system in terms of query throughput.

6 EXPERIMENTS

In this section, we evaluate the proposed methods 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.

6.1 Experimental Setting

Datasets. We conducted experiments on six real-life road networks from DIMACS [1], which are shown in Table 2. The maximal dataset USA has about 23.95 million vertices and 57.71 million edges.

Algorithms. We compare our MHL, PMHL, and PostMHL with three baselines: BiDijkstra [45], DCH [48], and DH2H [62]. BiDijkstra can be applied to dynamic road networks directly. DCH and DH2H are the state-of-the-art dynamic shortest path index in terms of index update and query efficiency. We set $k_e = 32$, $\beta_l = 0.1$, and $\beta_u = 2$ for PostMHL based on our preliminary experiments.

Query and Update Set. We generate a query set Q with 100,000 random queries and an update set $\mathcal{U}$ with 100,000 update instances for each network. The update interval δt ranges in $\{100, 500, 1000\}$ seconds and the batch size $|U|$ varies in $\{100, 1000, 10000\}$ edge updates. The default values of δt and $|U|$ are 100 and 1000.

Performance Measures. We report average *query time* T_q over Q, as well as average *index update time* T_u and *query throughput* Δ (number of processed SP queries) over 10 update batches.

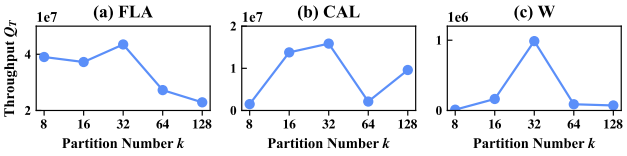


Figure 11: Effect of Partition Number k of PMHL

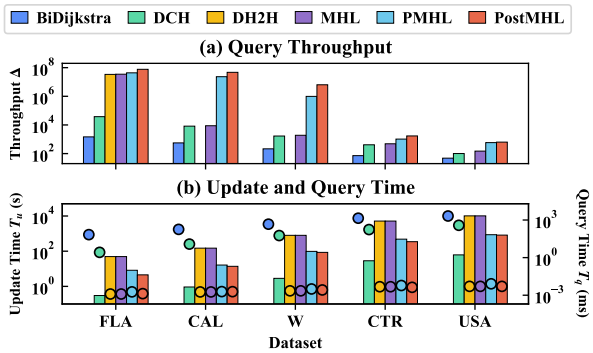


Figure 12: Comparison with Baselines (Bar: Δ or T_u, Ball: T_q)

6.2 Experimental Results

Exp 1: Effect of Partition Number k for PMHL. We first vary k from 8 to 128 on FLA, CAL, and W. As shown in Figure 11, both small and large k result in reduced query throughput, while $k = 32$ performs the best. This is due to the balance between boundary vertex number and parallelization. A larger k can make better use of parallelization but can also increase the number of boundary vertices and the tree height for the cross-boundary index. We set $k = 32$ for all datasets since it has the largest query throughput. Note that we also tested the effect of bandwidth τ for PostMHL, which is elaborated in the Appendix [2] due to the limited space. The default τ of PostMHL are listed in the last column of Table 2.

Exp 2: Comparison of Throughput, Update Time, and Query Time. We compare our methods with baselines on all datasets to evaluate their effectiveness. As shown in Figure 12, DH2H only has query throughput on the smallest dataset FAL because it cannot finish the index update in larger graphs. By contrast, MHL has better availability since it subtly amalgamates BiDijkstra, DCH, and DH2H into one system, thus achieving better Δ than the systems solely reliant on any one of them. However, MHL's query throughput is not very large when it comes to large graphs (*e.g.*, CAL) because the fastest H2H query cannot be released before the next update batch. On the other hand, PMHL has better scalability and throughput due to its faster index update achieved by thread parallelization and more intermediate stages. Furthermore, PostMHL outperforms all competitors in terms of query throughput because of the cutting-edge query efficiency (equivalent to DH2H) and exceptional index update efficiency. For example, the throughput of PostMHL is about 2.26× better than DH2H on FLA. For larger road networks where DH2H cannot be applied, it achieves up to 4 orders of magnitude higher query throughput than the best applicable baselines.

Exp 3: Evolution of Query Throughput and Efficiency. We set the sampling frequency as 1 Hz and report the evolution of query throughput and query efficiency of all methods on FLA, CAL, and W. As shown in Figure 13, the DH2H-based system is only

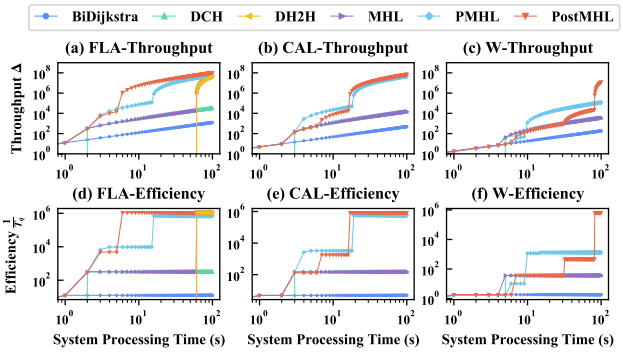


Figure 13: Evolution of Query Throughput and Efficiency

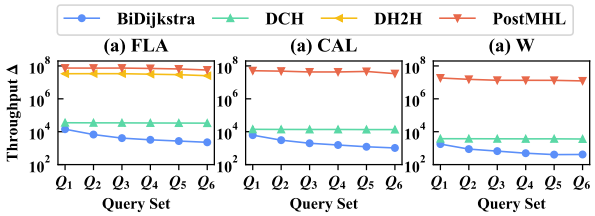


Figure 14: Effect of Query Distribution

applicable to FLA due to time-consuming index maintenance. By contrast, the MHL-based system is always available and consistently outperforms the system that solely uses BiDijkstra, DCH, or DH2H in terms of query throughput. On the other hand, owing to the subtle integration of novel PSP strategies, PMHL achieves faster index maintenance and a much larger query throughput than MHL. Note that PMHL may even exhibit superior throughput to PostMHL when system processing time falls between 10 and 80 in W due to a faster post-boundary query as shown in Figure 13-(f). Nevertheless, PostMHL outperforms PMHL in most cases due to the faster overall index update time and cross-boundary query efficiency, thus achieving the best query throughput in most cases. Since PostMHL generally has the best performance, we use it as the representative of our methods in the following experiment.

Exp 4: Effect of Query Distribution. We next evaluate the effect of query distribution. To simulate real-world query distribution, we generate 6 query sets $Q_1, Q_2, ..., Q_6$, where Q_i contains random $(i-1) \times 10,000$ cross-partition queries and $(11-i) \times 10,000$ same-partition queries. As shown in Figure 14, all methods tend to decrease the query throughput Δ when the proportion of cross-partition queries increases. This can be attributed to the increase in average query distance. Nevertheless, PostMHL exhibits little performance deterioration and consistently outperforms the baselines with a range of about $2.2 \times -4097.3\times$ improvement.

Exp 5: Varying Batch Size $|U|$ and Update Interval δt. Lastly, we vary $|U|$ and δt to demonstrate the robustness of PostMHL under different dynamic scenarios in Figure 15. Owing to the post-partitioning multi-stage index scheme, PostMHL can efficiently handle all update scenarios while providing the best query throughput. The throughput improvement is more evident in large graphs or frequent update scenarios. It is worth noting that when $|U|$ is small and Δ is large (*e.g.*, $|U| = 100, \Delta = 1000$ for FLA and CAL), despite PostMHL having faster index maintenance, it only slightly outperforms DH2H as both of them exhibit highly efficient index

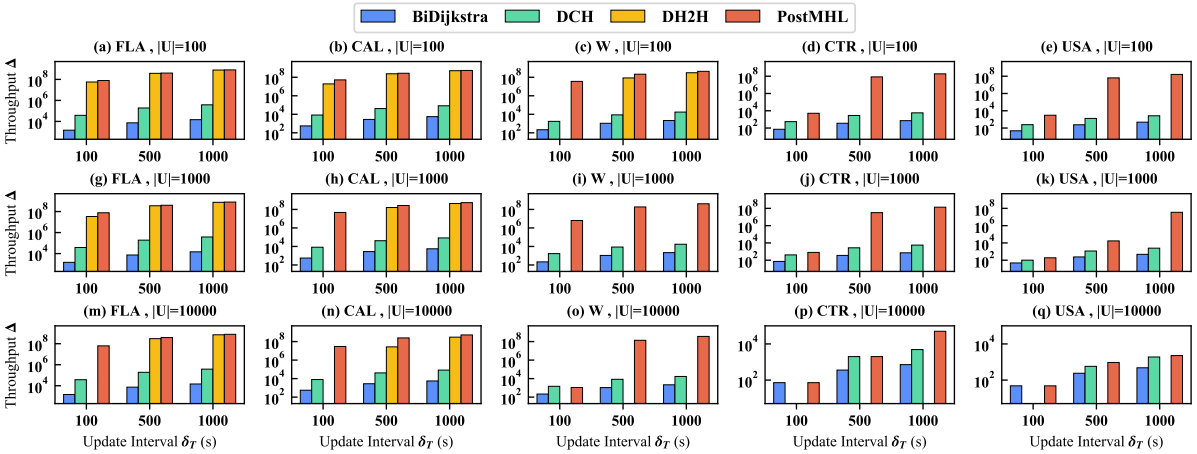


Figure 15: Varying Update Batch size $|U|$ and Update Interval δt

maintenance. Nevertheless, PostMHL achieves about 38% − 927% throughput improvement over DH2H in most cases. In cases where DH2H is not applicable (update is intense or network is large), PostMHL outperforms DCH by 1-4 orders of magnitude when the post-boundary query can be released within the update interval (e.g. Figure 15 (d)-(e)). It also achieves 20%-60% higher query throughput than DCH (e.g. δT = 500 or 1000 in Figure 15-(r)) when only PCH can be released in δT due to the faster index update of PCH. When updates are frequent and numerous (δT = 100 for Figure 15 (q)-(r)) such that DCH cannot be applied, PostMHL achieves the same query throughput as BiDijkstra. Overall, PostMHL not only exhibits better query throughput than baselines but also demonstrates excellent scalability and robustness, enabling efficient processing of both city-level and nation-level queries.

7 RELATED WORK

7.1 Static Shortest Path Algorithms

The shortest path algorithms on static graphs can be divided into two categories: index-free [17, 27, 29, 45] and index-based [6, 12, 13, 24, 39, 47, 53] methods. *Dijkstra's* [17] is the first index-free shortest path algorithm which searches the shortest path on the fly in a best-first manner, while A^* [27] and bidirectional methods such as *BiDijsktra* [45] and *Bi-A** [29] improves *Dijkstra's* efficiency by heuristic pruning techniques. However, in the worst case, all index-free methods require $O(|E| + |V| \log |V|)$ time to search the graphs for SP query processing. Although batch processing [38, 52, 63] could improve the throughput of the index-free path algorithm but not by orders of magnitude. To improve the query efficiency, numerous index-based methods have been proposed in the past few decades, such as *goal-directed methods* [26, 53], *hierarchical methods* [24, 50], *2-hop labeling* [4–6, 13, 30, 39, 40], and *tree decomposition-based methods* [12, 47, 56]. These indexes are about 1-5 orders of magnitude faster than the index-free methods in query processing. Among them, *PLL* [6] and *PSL* [39] have the fastest query efficiency on small-world networks, while *H2H* [47] and *P2H* [12] 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 [20, 40, 55, 67]. However, all these indexes take time to update when the graph evolves as them are initially designed for static graphs [48, 62, 64–66].

7.2 Dynamic Shortest Path Algorithms

To adapt the shortest path index to dynamic scenarios, some work about shortest path index maintenance [7, 14, 19, 25, 48, 49, 57, 62, 64, 65, 67] have been studied in the past decade. Among them, [7, 14, 49, 64, 65] focus on the index maintenance of 2-hop labeling PLL and PSL. For the road networks, *DCH* [48] adopts a shortcut-centric paradigm and shortcut weight propagation mechanism for the CH index maintenance, achieving the fastest update efficiency. *DH2H* [62] utilizes the star-centric paradigm for tracing index change of the H2H index. Moreover, it puts forward the bottom-up super-edge update and top-down label update mechanisms, enabling efficient batch updates for the H2H index. [67] systematically studies the dynamic PSP index and puts forward three general partitioned shortest path strategies and a universal scheme for designing the dynamic PSP index. However, most of these index maintenance methods only focus on improving the update efficiency of the corresponding static shortest path index, thus lacking the flexible and robust ability to cope with high throughput queries and updates.

8 CONCLUSION

In this paper, we investigate high throughput query processing on large dynamic road networks. We first propose the *MHL* index to subtly exploit the advantages of DCH and DH2H. To scale MHL into large road networks, we propose a non-trivial *PMHL* to leverage thread parallelization and multiple PSP strategies for fast index maintenance and continuously improved query efficiency. Lastly, based on an insightful analysis of the essence of PSP index, we propose a novel *PostMHL* adopting tree decomposition-based partitioning to achieve the same query efficiency with DH2H but a much faster index update efficiency. The experiments demonstrate that PostMHL yields up to 1-4 orders of magnitude throughput improvement compared to state-of-the-art baselines and is more robust and scalable for various update scenarios.

REFERENCES

[1] [n. d.]. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/download.shtml.

[2] [n. d.]. *High Throughput Shortest Path Query Processing for Large Dynamic Road Networks (Appendix and Code).* https://anonymous.4open.science/r/HTSP-22F5/

[3] [n. d.]. The number of trips during the Spring Festival in 2024 will reach 9 billion, 80% of which are self-driving trips. https://www.beijing.gov.cn/ywdt/zybwdt/202401/t20240117_3537753.html.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21 (2012), 869–888.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–49.

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

[18] Muhammad Farhan, Henning Koehler, and Qing Wang. 2023. BatchHL+: batch dynamic labelling for distance queries on large-scale networks. *The VLDB Journal* (2023), 1–29.

[19] 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.

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

[21] Linton C Freeman et al. 2002. Centrality in social networks: Conceptual clarification. *Social network: critical concepts in sociology. Londres: Routledge* 1 (2002), 238–263.

[22] Linton C Freeman, Stephen P Borgatti, and Douglas R White. 1991. Centrality in valued graphs: A measure of betweenness based on network flow. *Social networks* 13, 2 (1991), 141–154.

[23] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. IS-LABEL: an Independent-Set based Labeling Scheme for Point-to-Point Distance Querying. *Proceedings of the VLDB Endowment* 6, 6 (2013).

[24] 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.

[25] 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.

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

[27] 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.

[28] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully dynamic shortest-path distance query acceleration on massive networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1533–1542.

[29] 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.

[30] 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).

[31] 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.

[32] 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.

[33] Kartik Lakhotia, Rajgopal Kannan, Qing Dong, and Viktor Prasanna. [n. d.]. Planting Trees for scalable and efficient Canonical Hub Labeling. *Proceedings of the VLDB Endowment* 13, 4 ([n. d.]).

[34] 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.

[35] 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.

[36] 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.

[37] 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).

[38] 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.

[39] 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.

[40] 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.

[41] Zijian Li, Lei Chen, and Yue Wang. 2019. G*-tree: An efficient spatial index on road networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 268–279.

[42] 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.

[43] 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.

[44] 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.

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

[46] U.S. Bureau of Transportation Statistics. [n. d.]. *Transportation Statistics Annual Report*. https://www.bts.gov/tsar

[47] 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.

[48] 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.

[49] 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.

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

[51] 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

[52] 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.

[53] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. 2005. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)* 10 (2005), 1–3.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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).

[66] 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.

[67] 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).

[68] 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.

[69] 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.

[70] 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.

[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.

9 APPENDIX

9.1 Pseudo-code of MHL Index Construction

The pseudo-code of MHL index construction is presented in Algorithm 4. The first step is to leverage MDE-based tree decomposition to obtain T and shortcut arrays for all vertices (Line 1). In particular, we first initiate G^0 as the original road network G. Then, in the i-th iteration, we contract the vertex v with the smallest degree in G^{i-1}, forming the tree node $X(v)$ and getting the shortcut array $X(v).sc$ (Lines 15-22). After obtaining the tree nodes, we generate the tree decomposition T in a top-down manner by setting the parent of $X(v)$ as the vertex with the lowest order in $X(v).N$ (Lines 23-25). The second step of MHL is to compute the position array and distance array for all vertices in a top-down manner by leveraging the neighbor set $X(v).N$ as the vertex separator, obtaining the final MHL index $L_{MHL} = \{X(v)|v \in V\}$ (Lines 3-14).

Algorithm 4: MHL Index Construction

```
   Input: Road network G = {V, E}
   Output: MHL index L_MHL = {X(v)|v ∈ V}
1  T ← TreeDecomposition(G);                    ▷ MDE-based TD and get shortcut array
2  // Compute position array and distance array
3  for X(v) ∈ T in a top-down manner do
4      Suppose X(v) = (x_1, ··· , x_|X(v)|) = v)
5      for i = 1 to |X(v)| do
6          X(v).pos[i] ← the position of x_i in X(v).A;
7      for i = 1 to |X(v).A| − 1 do
8          c ← X(v).A[i]; X(v).dis[i] ← ∞;
9          for j = 1 to |X(v).N| do
10             if X(v).pos[j] > i then d ← X(x_j).dis[i];
11             else d ← X(c).dis[X(v).pos[j]];
12         X(v).dis[i] ← min{X(v).dis[i], X(v).sc[j] + d};
13     X(v).dis[|X(v).A|] ← 0;
14 return L_MHL = {X(v)|v ∈ V};
15 Function TreeDecomposition(G):
16     G^0 ← G; T ← φ;
17     for i = 1 to |V| do
18         v ← the vertex with the smallest degree in G^{i−1};
19         X(v).N ← v's neighbor vertices in G^{i−1};
20         for j = 1 to X(v).N do
21             X(v).sc[j] ← sc(v, X(v).N[j]);
22         X(v) ← X(v).N ∪ v; G^i ← Contract v on G^{i−1}; r(v) ← i;
23     for X(v) ∈ T in a top-down manner do
24         u ← the vertex with the lowest vertex order in X(v).N;
25         Set the parent of X(v) be X(u) in T;
26         X(v).A ← the ancestors of X(v) in T;
27     return T = {X(v)|v ∈ V};
```

9.2 Pseudo-code of PMHL Index Construction

The pseudo-code of PMHL index construction is presented in Algorithm 5. PMHL fist leverage graph partitioning method PUNCH [15] to divide the original road network G into k subgraphs (Line 1). Given the partition results as input, a *boundary-first* vertex ordering method [43] is leveraged to obtain the vertex order r of all vertices, which assigns the boundary vertices higher ranks than the non-boundary vertices since the *cross-boundary strategy* should fulfill the boundary-first property (Line 2). After obtaining the vertex order, there are six main steps for constructing the PMHL index (Lines 3-13), obtaining the index of various PSP strategies (see Section 4.2 for details).

Algorithm 5: PMHL Index Construction

```
   Input: Road network G = {V, E}
   Output: PMHL index L_PMHL = {L̃, {L_i}, {L'_i}, L*}
1  {G_i|1 ≤ i ≤ k} ← Partitioning G by PUNCH [15];   ▷ Get partition graphs
2  r ← BoundaryFirstOrder({G_i});                    ▷ Boundary-first vertex ordering
3  // No-boundary Index Construction
4  parallel_for i ∈ [1, k]
5      L_i ← MHLIndexing(G_i);                       ▷ Step 1: Build partition indexes {L_i}
6  G̃ ← OverlayGraphBuild({L_i});                     ▷ Step 2: Build overlay graph
7  L̃ ← MHLIndexing(G̃);                               ▷ Step 3: Build overlay index L̃
8  // Post-boundary Index Construction
9  parallel_for i ∈ [1, k]
10     G'_i ← GetExtendedGraph(L̃, G_i);              ▷ Step 4: Build extended partitions
11     L'_i ← MHLIndexing(G'_i);                     ▷ Step 5: Build post-boundary index {L'_i}
12 // Cross-boundary Index Construction, see Algorithm 1
13 L* ←CrossIndexBuild(L̃, {L_i});                    ▷ Step 6: Build cross-boundary index L*
14 return L_PMHL = {L̃, {L_i}, {L'_i}, L*};
```

9.3 Proofs

Proof of Lemma 1. As the vertex contraction process of CH and H2H is identical, the shortcuts produced by them are naturally equivalent, provided that they are executed in the same order. □

Proof of Theorem 2. The proof of this theorem has two aspects. We first prove that the distance array of L^* is correct. There are two cases:

Case 1: $v \in \tilde{G}$. This is naturally correct as the cross-boundary tree inherits the node relationship of the overlay tree and leverages the neighbor set of v's overlay index as the vertex separator.

Case 2: $v \notin \tilde{G}, v \in G_i$. As per Case 1, the cross-boundary labels of all G_i's boundary vertices are accurate. Therefore, the distance from v to its ancestor $u \notin G_i$ computed by the top-down label construction is correct as it essentially uses B_i as the vertex separator. Meanwhile, the calculation of the distance from v to its ancestor $u \in G_i$ is also accurate, as it uses the neighbor set of v's partition index as the vertex separator.

We next prove that $\forall s \in G_i, t \in G_j, i \neq j$, the LCA of $X^*(s)$ and $X^*(t)$ is the vertex separator of s and t. There are four cases:

Case 1: $s \in \tilde{G}, t \in \tilde{G}$. The LCA of $X^*(s)$ and $X^*(t)$ (denoted as $LCA(X^*(s), X^*(t))$) is the same as $LCA(\tilde{X}(s), \tilde{X}(t))$ since L^* has equivalent node relationships among the overlay vertices of $\tilde{L}$. Hence, $LCA(X^*(s), X^*(t))$ is the vertex separator.

Case 2: $s \in \tilde{G}, t \notin \tilde{G}$. We take the concise form of $sp(s,t)$ by extracting only the boundary vertices as $sp_c = \langle s = b_0, \ldots, b_n, t\rangle$ ($b_i \in B, 0 \leq i \leq n$). As per Case 1, we have $LCA(X^*(s), X^*(b_n))$ is the vertex separator of s and b_n. Besides, $X^*(b_n)$ is the ancestor of $X^*(t)$ since L^* inherits the subordinate relationships of the partition trees and $r(b_n) > r(t)$. Therefore, $LCA(X^*(s), X^*(t))$ is the same as $LCA(X^*(s), X^*(b_n))$ and it is the vertex separator.

Case 3: $s \notin \tilde{G}, t \in \tilde{G}$. The proof process is similar to Case 2.

Case 4: $s \notin \tilde{G}, t \notin \tilde{G}$. We take the concise form of $sp(s,t)$ by extracting only the boundary vertices as $sp_c = \langle s, b_0, \ldots, b_n, t\rangle$ ($b_i \in B, 0 \leq i \leq n$). As per Case 1, we have $LCA(X^*(b_0), X^*(b_n))$ is the vertex separator of b_0 and b_n. Besides, $X^*(b_0)$ is the ancestor of $X^*(s)$ while $X^*(b_n)$ is the ancestor of $X^*(t)$. Therefore,

$LCA(X^*(s), X^*(t))$ is the same as $LCA(X^*(b_0), X^*(b_n))$ and it is the vertex separator.

Since the distance array is correct and the LCA is a vertex separator, L^* can correctly answer cross-partition queries. □

Proof of Theorem 4. The post-boundary index needs the overlay index to construct all-pair boundary shortcuts and check whether any boundary shortcut has changed when updating it. On the other hand, the cross-boundary index requires the overlay index for index construction and to identify the affected in-partition vertices during index maintenance. Since they are independent of each other and only rely on overlay index, the theorem is proved. □

Proof of Theorem 5. The index size of PostMHL equals the H2H index size $O(n \cdot h)$ [47] plus boundary array size $O(\tau \cdot n_p)$. The indexing time of PostMHL consists of tree decomposition $O(n \cdot (w^2 + \log(n)))$, TD-partitioning $O(n \log(n))$, overlayindexing $O(n_o \cdot (\log(n_o) + \tilde{h} \cdot w))$, thread-parallel post-boundary and cross-boundary indexing $O(\frac{n_p}{k} \cdot (\log(n_p) + h \cdot \tau))$. □

Proof of Theorem 6. According to [62], the H2H decrease update and increase update complexity is $O(w \cdot (\delta + \Delta h))$ and $O(w \cdot (\delta + \Delta h \cdot (w + \epsilon)))$, respectively. Therefore, the theorem holds since the post-boundary and cross-boundary index updates can be conducted in parallel for all partitions. □

9.4 Additional Experimental Results

Exp 6: Effect of Bandwidth for PostMHL's Query Stages. We further demonstrate the effect of bandwidth for different query stages of PostMHL. As shown in Figure 16, the increase of τ has little effect on Q-Stage 1, 2, and 4 since the graph that BiDijkstra and PCH search on and the cross-boundary index do not change. However, the increase in bandwidth deteriorates the query efficiency of Q-Stage 3 (post-boundary query) for two reasons. Firstly, the proportion of cross-boundary queries increases as the overlay graph size reduces. Secondly, the cross-boundary query efficiency is undermined due to the larger boundary vertex number. Therefore, a smaller bandwidth may be more suitable for smaller road networks requiring frequent updates and where the cross-boundary query cannot be released but the post-boundary query is available.

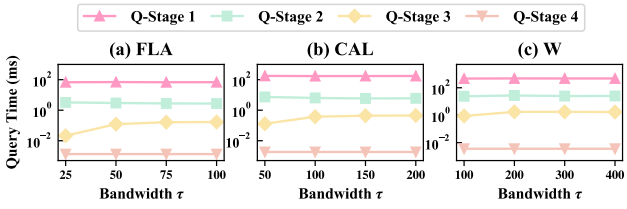


Figure 16: Effect of τ for PostMHL's Query Stages

Exp 7: Performance of Different Query Stages. We also present the performance of different query stages in MHL, PMHL, and PostMHL, to demonstrate the effectiveness of these stages. As demonstrated in Figure 17, all these methods show improved query efficiency toward the latter stages. Additionally, PMHL and PostMHL employ novel PSP strategies that allow the last query stage to be released as soon as possible. Furthermore, PMHL and PostMHL take advantage of the proposed novel PSP strategies to enable the last query stage to be released as soon as possible. For instance, in the FLA graph, MHL takes about 60 seconds to release the H2H query, while PMHL only takes 15 seconds to make the cross-boundary query online. Besides, PostMHL further enhances the index update efficiency by unifying post-boundary and cross-boundary strategies into one tree decomposition. It is the only index that can release the cross-boundary query within 100 seconds, as shown in Figure 17-(c).

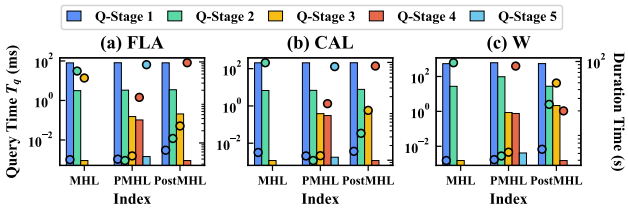


Figure 17: Performance of Different Query Stages