High Throughput Shortest Distance Query Processing on Large Dynamic Road Networks

Xinjie Zhou[1,2], Mengxuan Zhang[3], Lei Li[2,1], Xiaofang Zhou[1,2]
[1]The Hong Kong University of Science and Technology, Hong Kong SAR, China.
[2]DSA Thrust, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China.
[3]School of Computing, The Australian National University, Canberra, Australia.
xzhouby@connect.ust.hk, Mengxuan.Zhang@anu.edu.au, thorli@ust.hk, zxf@cse.ust.hk

***Abstract*—Shortest path (SP) computation is the building block for many location-based services, and achieving high throughput SP query processing with real-time response is crucial for those services. However, existing solutions can hardly handle high throughput query processing on large-scale dynamic road networks due to either slow query efficiency or poor dynamic adaption. In this paper, we leverage graph partitioning and propose novel *Partitioned Shortest Path (PSP)* indexes to process SP queries with high throughput in large dynamic road networks. Specifically, we first put forward a *cross-boundary strategy* to accelerate the query processing of PSP index and analyze its efficiency upper bound theoretically. After that, we propose a non-trivial *Partitioned Multi-stage Hub Labeling (PMHL)* that subtly aggregates multiple PSP strategies to achieve fast index maintenance and consecutive query efficiency improvement during index update. Lastly, to further optimize throughput, we design *tree decomposition-based graph partitioning* and propose *Post-partitioned Multi-stage Hub Labeling (PostMHL)* with faster query processing and index update than *PMHL*. Experiments on real-world road networks show that our methods outperform state-of-the-art baselines in query throughput, yielding up to 2 orders of magnitude improvement.**

I. Introduction

Shortest Path (SP) computation on the road networks is a building block for many location-based services (LBSs), such as navigation [1], POI recommendation [2], logistics, and ride-sharing [3]. With the increasing utilization of LBSs, three tendencies for SP query emerge: 1) *Massive queries*: *Didi* had 462,962 queries per second on average [4] in 2018, while Uber faces millions of queries per second. Processing SP queries with high *throughput* (the number of queries processed per unit time) is vital for the real-time response; 2) *Large-scale networks*: during Spring Festival, millions of long-range queries across multiple provinces in China are issued everyday [5]; the interstate highways handled the highest traffic volume in US [6]; 3) *Traffic dynamicity*: Beijing and New York had about 13.8 and 5.8 edge weight (travel time) updates per second on average [7], [8]. To better support the LBSs, we aim to answer SP queries on large dynamic road networks with high *throughput* in this paper.

The index-free SP algorithms such as *Dijkstra's* [10] and *BiDijkstra* [11] can search on the updated network directly but suffer from low efficiency [12], [13]. The index-based algorithms such as *Contraction Hierarchy (CH)* [14] and *2-Hop Labeling (HL)* [15]–[24] are faster in query with help of shortcuts (CH) and distance labels (HL). To handle the updates, index maintenance [25]–[36] are proposed: *Dynamic CH (DCH)* [32] is fast in index update by tracing the shortcut changes, but it is slow in query; *Dynamic Hierarchical 2-Hop labeling (DH2H)* [33] is the fastest in query, but is slow in index update due to massive label updates. Nevertheless, the system *throughput* is affected by both query and update efficiency [37], [38]. Note that this work follows previous works [9], [32], [33], [38], [39] and assumes the update arrives in batch while the system (SP index) should immediately handle the updates before query processing to avoid inaccurate results caused by the outdated index. As illustrated in Figure 1, suppose the edge updates arrive in batches (U_i) [40]–[42], we start by updating the road network and its SP index, which leads to an index unavailable period (red) for query processing until it is updated to the latest correct version (green). Although there is a longer time for *DCH* to process queries, its throughput is limited by its low query efficiency; while the throughput of *DH2H* is constrained by its short query processing period. Worse yet, the *large graph size* and *Quality of Service (QoS)* requirement such as *query response time* (*i.e.,* the time taken from the arrival of a query q to the time at which q's answer is computed) [37], [38] could further deteriorate the update time and throughput. To this end, *Partitioned Shortest Path (PSP) index* has been investigated for better scalability [43]–[51] and faster index maintenance [35].

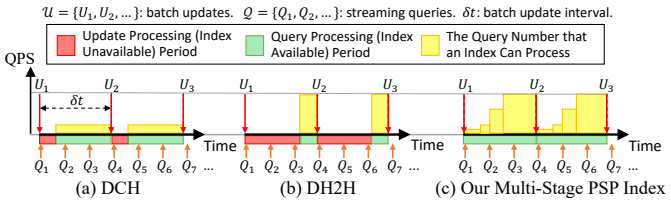


Fig. 1: Throughput Illustration of Dynamic SP Index (A larger Y-axis value (QPS, Queries per Second) indicates faster query processing; a larger yellow area indicates a higher *QoS-independent* [9] throughput; length of red and green is index update and query processing time, respectively)

Motivation. Given that the PSP index can exploit the parallelism among partition indexes for fast index maintenance and exhibits good scalability, there comes a question: is it possible to take advantage of this property to achieve our goal? In other words, how about we partition a network into

multiple subgraphs with one index for each subgraph, such that we can deal with large networks? Then with the parallelized index update for each subgraph, we can cope with updates quickly and leave more time for query processing. Therefore, we aim to propose novel dynamic PSP indexes to process high throughput SP queries on large dynamic road networks.

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

Challenge 1: How to enhance the query efficiency of the PSP index? Although PSP has a faster index update, it sacrifices the query efficiency through distance concatenantion [35]. To this end, we propose a general *cross-boundary strategy* to precompute global 2-hop labels and thus accelerate online query processing by eliminating the distance concatenation of the PSP index. Moreover, we dig into the vertex ordering of PSP index and insightfully reveal the *upper bound of PSP index query efficiency*.

Challenge 2: How to design a PSP index with high query throughput? High throughput is achieved by fast query and fast updates [35], [37]. However, they are hard to balance as a faster query requires a larger index that takes longer time to maintain [37], [39]. To this end, we propose to improve throughput by taking advantage of the index (un)available time. In particular, we subtly divide the index maintenance interval into multiple stages and leverage the currently fastest available index or algorithm for query processing for each stage. As illustrated in Figure 1-(c), the query efficiency is gradually enhanced, thereby improving the query throughput. To implement this idea, we first analyze the connection between *DCH* and *DH2H* and put forward an extended *H2H* index called *Multi-stage Hub Labeling (MHL)* that subtly integrates the *CH* and *H2H* index. Using *MHL* as the underlying SP index, we propose a novel *Partitioned MHL (PMHL)* that incorporates and optimizes multiple PSP strategies to enhance query efficiency across different update stages during index maintenance, thereby achieving high query throughput.

Challenge 3: How to empower PMHL with its optimal query efficiency and further accelerate the index update for higher throughput? Although the query efficiency is constrained by the aforementioned *PSP query upper bound*, we discover that it can be enhanced with appropriate vertex ordering. To this end, we put forward *tree decomposition-based graph partitioning (TD-partitioning)* to utilize the higher quality vertex ordering for partitioning. Based on this, we propose a novel *Post-partitioned MHL (PostMHL)* with a redesigned index structure and deliberately optimized PSP strategies to achieve optimal query efficiency (equivalent to *DH2H*) and further expedite index maintenance for higher query throughput.

Contributions. We summarize our contributions as follows:

- We propose a novel *cross-boundary strategy* to enhance the query processing and provide an insightful analysis of the *upper bound of PSP index query efficiency*;
- We propose a novel *PMHL* index and design various optimizations to achieve fast index maintenance and continuously improved query efficiency during index maintenance;
- We propose a novel *TD-partitioning* to leverage good vertex ordering for partitioning and propose the *PostMHL* index with both faster query answering and index update than *PMHL* to further improve query throughput;
- Extensive experiments on real-world datasets show that our methods have better query throughput than existing solutions, yielding up to 2 orders of magnitude improvement.

II. Preliminary

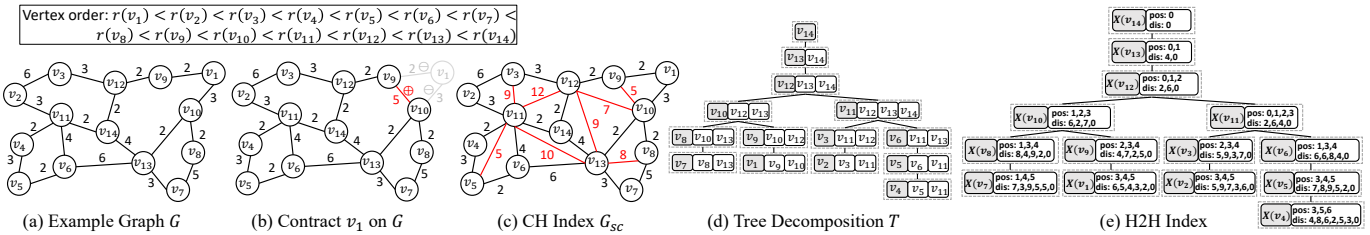
Let $G = (V, E)$ be a weighted graph where the vertex set V and edge set E denote road intersections and segments. We use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges. Each edge $e(u, v) \in E$ is associated with a positive weight $|e(u, v)|$ for travel time. For each vertex $v \in V$, we denote its neighbor set as $N_G(v) = \{u|(u, v) \in E\}$ and assign it an order $r_G(v)$ indicating its importance in G. A *path* p_G from s to t is a sequence of consecutive vertices $p_G = \langle s = v_0, ..., v_j = t\rangle = \langle e_1, ..., e_j\rangle$ with length $len(p_G) = \sum_{e\in p_G} |e|$. The *shortest path* $sp_G(s, t)$ from s to t is the path with the minimum length. The *shortest distance query* $q(s, t)$ aims to return the length of $sp_G(s, t)$ denoted as $d_G(s, t)$. We omit the notation G when the context is clear. We consider G as an undirected graph, and our techniques can be easily extended to directed graphs. The dynamicity in this paper refers to edge weight increase or decrease updates.

We next introduce the system model to handle the SP queries. We adopt the *batch update arrival* model [37] since the updates are often processed by batch in most LBSs (*e.g.,* TomTom [40], INRIX [41] update every 1 minute while AutoNavi [42] updates every few minutes) and previous works [32], [33]. In particular, we divide time into periods, each of a duration of δt seconds. The graph updates U are collected in a batch at the beginning of each period, reflecting the graph changes in the last period. Following previous works [37], [38], we assume the queries arrive at the system as a Poisson process and are queued for processing. Besides, following [38], when each update batch arrives, we assume the system immediately handles the updates before the query processing to avoid *staleness* [9] (*i.e., inaccurate SP computation*). If the update time $t_u \geq \delta t$, the system will have no throughput since it spends all its time on the updates. We adopt the *average query response time* R_q^* as a QoS constraint. Let t_q, V_q be the average and variance of query (processing) time, respectively; and t_u be the average update time. Therefore, the *maximum average throughput* λ_q^* can be computed by Lemma 1.

Lemma 1. $\lambda_q^* \leq \min\{\frac{2\cdot(R_q^*-t_q)}{V_q+2\cdot R_q^*\cdot t_q-t_q^2}, \frac{\delta t-t_u}{t_q\cdot\delta t}\}$.

Proof. The system is an M/G/1 queue. By Pollaczek-Khinchine formula [52], we have $R_q = \frac{\rho+\lambda_q\mu V_q}{2\cdot(\mu-\lambda_q)} + t_q$, where $\mu = 1/t_q$ is the service rate and $\rho = \lambda_q/\mu$. Hence, the first item holds. Besides, fulfilling the update constraint yields $\delta t - \delta t \cdot \lambda_q^* \cdot t_q \geq t_u$, so the second term holds. □

Therefore, the High Throughput Shortest Path query processing (HTSP) problem aims to maximize the maximum average query throughput λ_q^*.



(a) Example Graph G (b) Contract v_1 on G (c) CH Index G_{sc} (d) Tree Decomposition T (e) H2H Index

Fig. 2: Example Road Network G, CH Index G_{sc}, Tree Decomposition T, and H2H Index

III. Existing Solutions

The current solutions for *HTSP* are the following *hierarchy-based* [32], *hop-based* [33]) and *partition-based* [35].

A. Hierarchy-based Solution

CH [14] builds a hierarchical *shortcut index* G_{sc} by iteratively contracting vertices in the order of $r(v)$. The contraction involves two steps: 1) for each pair $\forall v_j, v_k \in N(v_i)$, we add the shortcut $sc(v_j, v_k)$ with weight $|sc(v_j, v_k)| = |e(v_i, v_j)| + |e(v_i, v_k)|$ to G; 2) we remove v_i and its adjacent edges from G when all pairs of neighbors in v_i are processed. We obtain G_{sc} after contracting all vertices. The query processing of *CH* is a modified *BiDijkstra*'s search [11] on G_{sc} which only searches the edges/shortcuts from low-order vertices to high-order vertices. *DCH* [32] propose *shortcut-centric* paradigm and *Shortcut Supporting Graph* to identify and propagate the affected shortcuts, resulting in fast index maintenance of the *CH* index. Nevertheless, *DCH* fails to handle high throughput queries due to low query efficiency.

Example 1. *Figure 2-(a) shows an example network G with vertex order $r(v_1) < ... < r(v_{14})$. Figure 2-(b) illustrates the contraction of vertex v_1, adding shortcut $sc(v_9, v_{10})$ to G and eliminating v_1 and its adjacent edges. The contraction process is iteratively conducted until all vertices are examined, forming the CH index G_{sc} shown in Figure 2-(c).*

B. Hop-based Solution

It assigns each vertex $v \in V$ with a vertex set $L(v) \subseteq V$ and a label set $\{(u, L(v,u)) | u \in L(v)\}$, where $L(v,u) = d(v,u)$. The labels should fulfill *2-hop cover* property that $\forall s, t \in G$, $d(s,t) = \min_{c \in L(s) \cap L(t)}\{L(s,c) + L(t,c)\}$. *Hierarchical 2-Hop Labeling (H2H)* [22] is the state-of-the-art in road networks, which is built based on tree decomposition:

Definition 1 (Tree Decomposition). *The tree decomposition T of graph $G = (V, E)$ is a rooted tree in which each tree node $X(v)$ corresponds to a vertex $v \in V$ and is a subset of V ($X(v) \subseteq V$), such that [22]: (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 $\{X(u) | v \in X(u)\}$ induces a subtree of T.*

We obtain T through *Minimum Degree Elimination (MDE)* [53], [54]. In particular, we start by initializing G as G^0 and then iteratively contract all vertices. In the i^{th} round ($i \in [1, n]$), we extract minimum-degree vertex v along with its neighboring edges from G^{i-1}, generating a tree 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 shortcuts among $X(v).N$ in G^{i-1} to preserve the shortest distances, forming a contracted graph G^i. The contraction order produces a vertex rank r in ascending order. After the vertex contraction, T is formed by setting $X(u)$ as the parent of $X(v)$ if u is the lowest-order vertex in $X(v).N$. H2H index defines two arrays for each node $X(v) \in T$: *distance array* $X(v).dis$ stores the distances from v to all vertices $u \in X(v).A$, where $X(v).A$ is the ancestor set of $X(v)$; *position array* $X(v).pos$ records the position of neighbor $u \in X(v)$ in $X(v).A$.

Given a query $q(s,t)$, its shortest distance is calculated as $d(s,t) = \min_{i \in X.pos}\{X(s).dis[i] + X(t).dis[i]\}$ with X denoting the *Lowest Common Ancestor (LCA)* [55] of $X(s)$ and $X(t)$. *DH2H* is the dynamic version of *H2H* which maintains *H2H* index in two phases: *bottom-up shortcut update* and *top-down label update*, which both rely on *star-centric* paradigm to trace and update the affected shortcuts or labels [33]. Nevertheless, its label update phase is time-consuming, making it hard to obtain high throughput.

Example 2. *Figure 2 (d)-(e) presents the tree decomposition T and its H2H index. Given a query $q(v_7, v_4)$, we first identify the LCA of $X(v_7)$ and $X(v_4)$, which is $X(v_{12})$. Then $d(v_7, v_4) = \min_{i \in X(v_{12}).pos}\{X(v_7).dis[i] + X(v_4).dis[i]\} = 11$.*

C. Partition-based Solution

Graph partitioning [56], [57] is widely used to scale up SP index by reducing the construction time [47], [51], [58], index size [46], [49], [59], and maintenance time [35]. We refer to the SP index that leverages graph partitioning as *Partitioned Shortest Path (PSP)* index. As discussed in [35], the PSP index could be classified into three types from the perspective of partition structure: *planar*, *core-periphery*, and *hierarchical* PSP indexes. For example, the classic CRP [44] is planar PSP since it treats all partitions equally on one level; G-Tree [45] and ROAD [60] are hierarchical PSP index since they organize the partitions hierarchically in multiple levels. The PSP indexes discussed in this paper are all planar PSP indexes when the context is clear. In particular, a road network G is divided into multiple subgraphs $\{G_i | 1 \leq i \leq k\}$ such that $\bigcup_{i \in [1,k]} V(G_i) = V, V(G_i) \cap V(G_j) = \emptyset (\forall i \neq j, i, j \in [1, k])$. We denote the edges between different partitions on G as inter-edges E_{inter} while other edges as intra-edges E_{intra}. An *overlay graph* $\tilde{G}$ is then built among the boundary vertices of all partitions to preserve the global shortest distances of G. We denote the boundary vertex set of G_i as B_i and $B = \bigcup_{i \in [1,k]} B_i$. The PSP index L over G consists of the *partition index* $\{L_i\}$ and the *overlay index*

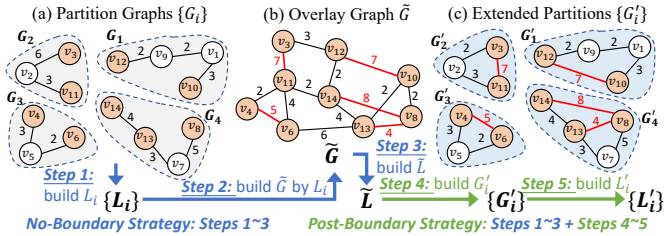


Fig. 3: Illustration of No-boundary and Post-boundary Strategy

$\tilde{L}$, *i.e.*, $L = \{L_i\} \cup \tilde{L}$. Usually, the PSP index is built by precomputing the distances of all-pair boundary shortcuts $E_B(G_i) = \{d(b_{i1}, b_{i2}) | \forall b_{i1}, b_{i2} \in B_i\}$ for each partition G_i by *Dijkstra*'s algorithm. These shortcuts are then inserted into G_i to form an *extended partition* G'_i and are also used to construct $\tilde{G}$ ($V(\tilde{G}) = B, E(\tilde{G}) = \bigcup_{i\in[1,k]} E_B(G_i) \cup E_{inter}$). L_i and $\tilde{L}$ are built on G'_i and $\tilde{G}$, respectively. The above approach is called *Pre-boundary Strategy* [35], whose query processing involves two cases:

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

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

To enhance the index construction and update of pre-boundary strategy, [35] proposes two novel strategies.

No-boundary Strategy. It avoids *Dijkstra*'s searches in boundary shortcut construction and builds the index by 3 *Steps*: 1) build the partition index $\{L_i\}$ on $\{G_i\}$; 2) leverage $\{L_i\}$ to compute the boundary shortcuts and construct $\tilde{G}$; 3) construct $\tilde{L}$ on $\tilde{G}$. The no-boundary strategy subtly converts the boundary shortcut computation from search-based to index-based, thus resulting in faster index construction and maintenance. Nonetheless, it sacrifices *same-partition* query efficiency due to the concatenation between $\{L_i\}$ and $\tilde{L}$: $\forall s, t \in G_i, q(s,t) = \min\{d_{L_i}(s,t), \min\limits_{b_p, b_q \in B_i} \{d_{L_i}(s,b_p) + d_{\tilde{L}}(b_p,b_q) + d_{L_i}(b_q,t)\}\}$.

Post-boundary Strategy. It aims to further improve the same-partition query efficiency by adding 2 more *Steps*: 4) leverage $\tilde{L}$ to compute the all-pair boundary shortcuts and form extended partitions $\{G'_i\}$; 5) build correct partition index $\{L'_i\}$ on $\{G'_i\}$. The final post-boundary index is $\{\tilde{L}, \{L'_i\}\}$, which has the same query processing as the pre-boundary strategy due to the correct partition index $\{L'_i\}$.

Example 3. *Figure 3-(a) presents the partition graphs $\{G_i\}$, and (b) shows overlay graph $\tilde{G}$ with pre-computed all-pair shortcuts in red. Extended partitions $\{G'_i\}$ are shown in (c), with orange boundary vertices. The index construction procedures are also illustrated at the bottom of Figure 3, where no-boundary strategy involves Steps 1-3 while post-boundary further includes Steps 4-5. Their index updates are similar to the index construction by replacing the build with the update.*

Remark 1. Although *DCH* can swiftly adapt to dynamic environments, it is relatively slow in query answering, which may necessitate the deployment of hundreds to thousands of servers to address HTSP problem. *DH2H* offers much higher query efficiency, but its index maintenance is too time-consuming to achieve high query throughput. *Partition-based solution* [35] provides chances to accelerate the index update of *hop-based solutions* (*i.e.*, *DH2H*) by thread parallelism among subgraphs, making it promising to leverage the *hop-based solution* for high query throughput in practice.

IV. Theoretical Analysis of PSP Index

In this section, we first propose a general *Cross-boundary Strategy* for faster PSP query efficiency. Then we provide an insightful analysis of its query efficiency upper bound. Both parts pave the way for our proposed solution as will be introduced in Sections V and VI.

A. Cross-boundary Strategy

Although the PSP index exhibits faster index updates than its non-partitioned counterparts, it sacrifices query efficiency since it has to concatenate the overlay index and partition indexes to obtain the accurate global distance for cross-partition queries [35]. To improve its query efficiency, we propose the *Cross-boundary Strategy* to build a global non-partitioned cross-boundary index L^* by concatenating the overlay and partition indexes in advance. In particular, suppose $\tilde{L}$ and $\{L'_i\}$ have been built by *post-boundary strategy* with 2-hop labeling as the underlying index. For $\forall v \in B_i$, we direct inherit its overlay index, *i.e.*, $L^*(v) = \tilde{L}(v)$ and $L^*(v,c) = \tilde{L}(v,c), \forall c \in \tilde{L}(v)$; otherwise for $\forall v \in G_i \setminus B_i$, $L^*(v) = L'_i(v) \cup \bigcup_{b\in B_i} \tilde{L}(b)$ with the distance label calculated as $L^*(v,c) = \min_{b\in B_i}\{d_{L'_i}(v,b) + d_{\tilde{L}}(b,c)\}, \forall c \in \bigcup_{b\in B_i} \tilde{L}(b)$ or $L^*(v,c) = L'_i(v,c), \forall c \in L^*(v) \setminus \bigcup_{b\in B_i} \tilde{L}(b)$. Then we prove the index correctness of L^* in the lemma below:

Lemma 2. *L^* satisfies the 2-hop cover property,* i.e., $\forall s, t \in G, d(s,t) = \min_{c\in L^*(s)\cap L^*(t)}\{L^*(s,c) + L^*(t,c)\}$.

Proof. For $\forall s \in G_i, t \in G_j, i \neq j$, there are three cases based on whether the query endpoints are boundary vertices or not: *Case 1*: $s, t \in B$. It holds since $\tilde{L}$ direct inherited overlay index for both s, t. *Case 2*: $s \in B, t \notin B$. Suppose $b \in B_j$ lies in $sp(s,t)$, then there must contain a vertex $c \in \tilde{L}(b) \cap \tilde{L}(s)$ in both $sp(s,b)$ and $sp(s,t)$, thus $d(s,t) = L^*(s,c) + L^*(t,c)$. *Case 3*: $s, t \notin B$. Suppose $b_1 \in B_i$, $b_2 \in B_j$ lie on $sp(s,t)$, then there must contain a vertex $c \in \tilde{L}(b_1) \cap \tilde{L}(b_2)$ in $sp(b_1,b_2)$ and $sp(s,t)$, thus $d(s,t) = L^*(s,c) + L^*(t,c)$. For $\forall s, t \in G_i$, the inherited L'_i naturally satisfies 2-hop cover. □

The multi-hop distance concatenations are eliminated and turned to 2-hop concatenation for cross-partition queries with the query complexity reduced by a factor of $O(|B_{max}|^2)$, where $|B_{max}| = \max_{i\in[1,k]}\{|B_i|\}$. To update the index, we first maintain $\tilde{L}$ and $\{L'_i\}$. Then for each partition G_i, if $\forall b \in B_i, \tilde{L}(b)$ has changed, we update $L^*(v)$ of all non-boundary vertices (*i.e.*, $v \in G_i \setminus B_i$). Otherwise, we only check the $L^*(u)$ of non-boundary vertices u with changed $L_i(u)$.

Example 4. We illustrate the cross-boundary strategy in *Figure 4. For $v_{10}, v_{12} \in B_1$ (blue), we directly inherit their overlay index from $\tilde{L}$. For $v_1, v_9 \notin G_1 \setminus B_1$, we concatenate*

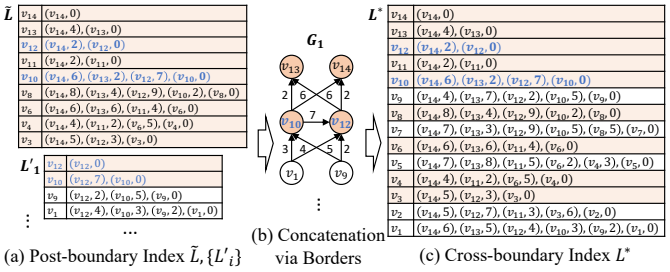


Fig. 4: Illustration of Cross-boundary Strategy

L'_1 and $\tilde{L}$ via the borders v_{10} and v_{12} to obtain the global distances to $v_{14}, v_{13}, v_{12}, v_{10}$, e.g., *$L^*(v_1, v_{14}) = L'_1(v_1, v_{12}) + \tilde{L}(v_{12}, v_{14}) = 6$. L^* accelerates cross-partition queries by 2-hop labeling,* e.g., *$q(v_1, v_7) = L^*(v_1, v_{13}) + L^*(v_7, v_{13}) = 8$.*

The *cross-boundary* strategy successfully accelerates query answering by stitching the partitioned indexes and overlay index into a whole one satisfying the 2-hop cover property, which will also be leveraged in Section V. Nevertheless, we observe that its query answering speed is still inferior to that of *DH2H*. To explain this, we next reveal the upper bound of PSP index query efficiency theoretically.

B. Upper Bound of PSP Index Query Efficiency

It is well known that vertex order determines the index structure which further affects the SP query efficiency [14], [22], [61]. Hence, we first analyze the order of PSP index under the cross-boundary strategy. Note that to make the discussion meaningful, we assume the 2-hop labeling based on a vertex order r is *canonical labeling* [17], *i.e.,* only higher-rank vertex u could be the hub of vertex v and no label is redundant, so it has a minimal label size *w.r.t.* order r [17]. We introduce the *Boundary-first Property*: for each partition in PSP index, its boundary vertices have a higher order than non-boundary ones. This is because for a cross-partition query $q(s,t)$ ($\forall s \in G_i, t \in G_j$), $d(s,t)$ is calculated by concatenating distance values $d(s,b_p)+d(b_p,b_q)+d(b_q,t)$ ($\forall b_p \in B_i, b_q \in B_j$). This generally requires non-boundary vertices to store the distances to their boundary vertices, essentially setting a higher vertex order for the boundary vertices as only a higher-order vertex could be the hub of another vertex in *canonical 2-hop labeling* [17].

Then we discuss how to construct a PSP vertex order satisfying the *boundary-first property*. As illustrated in Figure 5, given a graph partitioning result $\{G_i\}$, we denote the non-boundary vertex set as I_i and boundary vertex set as B_i for each G_i. The first step is ordering vertices for all $\{G_i\}$ and $\tilde{G}$ with two conditions: 1) the partition orders $\{r_{G_i}\}$ should satisfy the boundary-first property; 2) the relative order among the boundary vertices in $\{r_{G_i}\}$ should be consistent with overlay order $r_{\tilde{G}}$. After obtaining the individual vertex orders, the second step is aggregating $r_{\tilde{G}}$ and $\{r_{G_i}\}$ for overall vertex ordering r_G while preserving the relative order in them. As shown in Figure 5-(c), there are various overall vertex orders satisfying this aggregation requirement. For example, we can set all boundary vertices with higher ranks than non-boundary

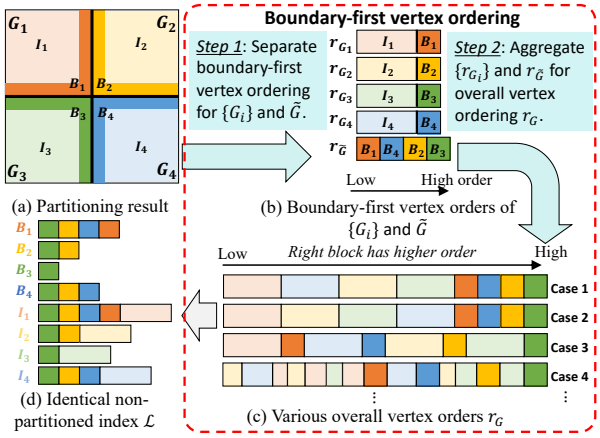


Fig. 5: Illustration of Boundary-first Vertex Ordering

vertices and randomly interleave the non-boundary vertices (*e.g.,* Case 1 and 2). Nevertheless, the 2-hop labeling based on these different PSP vertex orders leads to identical *canonical labeling* $\mathcal{L}$ [17] as proved in the following lemma.

Lemma 3. *All vertex orders of a PSP index that satisfy boundary-first property lead to identical canonical labeling.*

Proof. Given any boundary-first vertex ordering r_G, we discuss whether a vertex u is the hub of $\forall v \in V, r(u) > r(v)$.

Case 1: $v \in B$ ($v \in B_i$). When $u \notin B$, u cannot be v's hub if $u \in I_i$ since $r(u) < r(v)$. For the case that $u \in I_j, i \neq j$, u also cannot be v's hub since $sp(u,v)$ must contain a vertex $b \in B_j$ with $r(b) > r(u) > r(v)$, leading to pruned label $\mathcal{L}(v,u)$. Therefore, u could be v's hub only if $u \in B$ and v's labels are only determined by $r_{\tilde{G}}$. Since all r_G should preserve the relative order of $r_{\tilde{G}}$, they have identical labels.

Case 2: $\forall v \in V \setminus B$. When $u \in I_j, i \neq j$, u cannot be v's hub since $sp(u,v)$ must contain a vertex $b \in B_j$ with $r(b) > r(u) > r(v)$, pruning label $\mathcal{L}(v,u)$. When $u \in B_i$ or $u \in I_i$, u could be v's hub. For $u \in B_j, i \neq j$, u could be v's hub if $r(b) < r(u), \forall b \in B_i$. Hence, v's labels are determined by r_{G_i} and $r_{\tilde{G}}$, which is fulfilled by all r_G. □

Example 5. *Figure 5-(d) illustrates the identical non-partitioned index $\mathcal{L}$ based on various r_G. For the boundary vertex such as $\forall v \in B_2$, all other higher-order boundary vertices ($u \in B_2 \cup B_3$) could be its hub. For the non-boundary vertex such as $\forall v \in I_4$, any higher-order vertex $u \in I_4 \cup B_4 \cup B_2 \cup B_3$ could be its hub since any vertex in B_3 and B_2 has higher rank than boundary vertex in B_4.*

Lemma 3 reveals that although a PSP index could have various boundary-first orders, the canonical 2-hop labels based on these orders are identical, leading to Theorem 1.

Theorem 1 (The Upper Bound of PSP Query Efficiency). *Given a PSP index L and its boundary-first order r_G, the optimal query efficiency that L (or its cross-boundary index L^*) can achieve is that of canonical labeling $\mathcal{L}$ based on r_G.*

Proof. We prove by dividing all scenarios into two cases and indicating that $\mathcal{L}$ is the subset of or equal to L^*: *Case 1*: $\forall v \in B$. $L^*(v)$ and $\mathcal{L}(v)$ have identical index as per Lemma 3; *Case 2*: $\forall v \in V \setminus B$ such as $v \in I_i$. We have $\mathcal{L}(v) \subseteq L^*(v)$ since $\mathcal{L}(v)$ is canonical while $L^*(v)$ may exist redundant labels.

For example, suppose $b_1, b_2 \in B_i, r(b_1) > r(b_2)$ and b_1 is the highest-order vertex in the shortest path between v and b_2, then $\mathcal{L}(v, b_2)$ is pruned. By contrast, b_2 exists in $L^*(v)$ as per the cross-boundary strategy. Therefore, $\mathcal{L} \subseteq L^*$. □

This theorem theoretically explains the inferior query efficiency of the existing PSP index, as no graph partitioning method is designed toward a high-quality vertex ordering. Such a pessimistic conclusion seems to restrain any attempt to improve query efficiency through PSP index. But should we really give up? In Section VI, we propose to utilize this theorem reversely by introducing a novel vertex ordering-oriented partitioning to optimize PSP query efficiency.

V. Partitioned Multi-Stage 2-Hop Labeling

In this section, we propose our first solution – *Partitioned Multi-stage Hub Labeling (PMHL)* – for the HTSP problem.

A. Integration of DCH and DH2H

As analyzed in Section III, there is a trade-off between query and update efficiency for both *DCH* and *DH2H*. Nevertheless, we discover a connection between them that leverages both of their benefits, as shown in the lemma below:

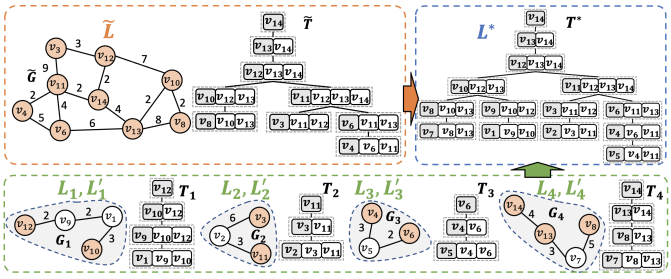
Lemma 4. *DH2H can generate equivalent shortcuts required by DCH if they use the same vertex order r.*

Proof. As introduced in Section II, *DCH* and *DH2H* have identical vertex contraction processes (both iteratively contract vertices as per r). Therefore, the shortcut generated in the tree decomposition of H2H is equivalent to the shortcut index of CH if they both use the MDE-based vertex ordering [53]. □

Lemma 4 reveals that the CH index construction (resp. update) could be regarded as the first step of H2H index construction (resp. update). Therefore, we could integrate the CH index into the H2H index by extending each tree node $X(v), \forall v \in V$ in the H2H index with an additional *shortcut array* $X(v).sc$ storing distances from v to vertices in $X(v).N$. We call such an extended H2H index as *Multi-stage Hierarchical 2-hop Labeling (MHL)* since it provides an intermediate stage to enable the release of CH-based query processing during the H2H index maintenance. Moreover, we could leverage index-free algorithms such as *BiDijkstra* [11] to process queries when both the CH and H2H indexes are unavailable. Consequently, *MHL* achieves higher query throughput than the solution solely based on *BiDijkstra*, *DCH* or *DH2H*. We call the scheme of making full advantage of the fastest available index or algorithm for query processing during index maintenance as *multi-stage scheme*.

B. Overview of PMHL-based Solution

Even though with higher throughput, *MHL* also inherits the laborious maintenance of *DH2H*. As a result, the fast query processing could be unavailable when the maintenance is time-consuming with numerous updates or in large networks, resulting in lower throughput. Therefore, we introduce the partition-based solution (called *Partitioned Multi-stage Hub Labeling (PMHL)*) since it offers an opportunity to accelerate maintenance through multi-thread parallelization [35]. *PMHL*



$\tilde{T}$ is overlay tree of $\tilde{L}$ while $\{T_i\}$ are the partition trees of no-boundary index $\{L_i\}$. T^* is cross-boundary tree of L^*, which is generated by aggregating $\tilde{T}$ and $\{T_i\}$.

Fig. 6: Illustration of PMHL Index

subtly exploits multiple sets of indexes and different PSP strategies (*no-boundary*, *post-boundary*, and *cross-boundary*) to gradually improve query efficiency while updating the index. Corresponding to the PSP strategies, *PMHL* includes three types of indexes: *no-boundary index* $\{\tilde{L}, \{L_i\}\}$, *post-boundary index* $\{L_i'\}$, and *cross-boundary index* L^*.

Example 6. *Figure 6 illustrates the example of PMHL index, which consists of no-boundary $\{L_i\}$ and post-boundary $\{L_i'\}$ for each $\{G_i | i \in [1, 4]\}$, overlay index $\tilde{L}$, and cross-boundary index L^*. The underlying index is MHL and the corresponding tree decompositions are also presented.*

Before diving into the details of *PMHL*, we overview its framework in Figure 7. There are five stages for update and query. After each update stage, it exploits the fastest available index for querying thus continuously enhancing query efficiency. It is worth noting that since the *DCH* update is the initial step of *DH2H* maintenance (Section V-A), we decouple the no-boundary index update into shortcut update stage (U-Stage 2) and label update stage (U-Stage 3), such that *Partitioned CH (PCH) query* (which adopts the same mechanism as CH with the only difference that it searches on the union of the shortcut arrays of $\tilde{L}$ and $\{L_i\}$) [35] could be released after U-Stage 2. After U-Stage 5, the cross-boundary index (an H2H-style query processing) is released for the fastest query processing.

C. PMHL Index Construction and Query Processing

To build *PMHL* index, we first leverage *PUNCH* [57] to divide G into k subgraphs $\{G_i\}$ and then set the *boundary-first* ordering r with *MDE* [53]. Next, we build the *PMHL* index in six steps. *Step 1*: Construct index $\{L_i\}$ for each partition $\{G_i\}$ in parallel; *Step 2*: Construct the overlay graph $\tilde{G}$ based on the shortcuts among the boundary vertices; *Step 3*: Construct index $\tilde{L}$ for $\tilde{G}$; *Step 4*: Compute $d_G(b_{i1}, b_{i2}), \forall b_{i1}, b_{i2} \in B_i$ using $\tilde{L}$ for all partitions $\{G_i\}$ in parallel and insert them to $\{G_i\}$ to get $\{G_i'\}$. *Step 5*: Construct $\{L_i'\}$ on all $\{G_i'\}$ in parallel; *Step 6*: Construct L^* by aggregating $\tilde{L}$ and $\{L_i\}$.

In particular, Steps 1-3 construct the no-boundary index for *PMHL*, including $\tilde{L}$ and $\{L_i\}$. Steps 4 and 5 focus on building the post-boundary partition indexes $\{L_i'\}$, while Step 6 creates the cross-boundary index L^*. We next optimize the index construction and query processing by leveraging different boundary strategy as introduced before.

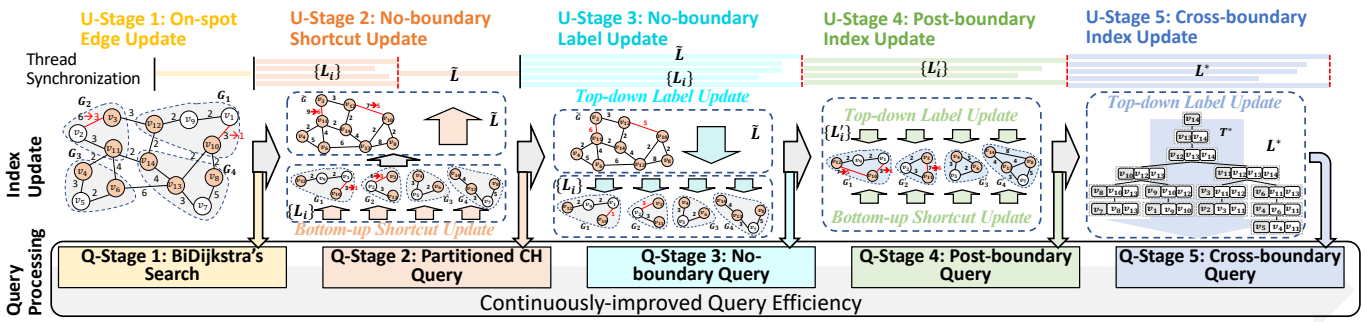


Fig. 7: Illustration of PMHL-based Solution

*1) **Optimize Index Construction**:* As introduced in Section III-C, no-boundary and post-boundary strategies leverage the partition index for the boundary shortcut computation, avoiding the time-consuming *Dijkstra*'s search. However, when applying them for *PMHL*, we find that the boundary shortcut computation is still time-consuming due to the low CH query efficiency. To tackle this issue, we optimize these two strategies by directly leveraging the shortcuts formed in the contraction of partition vertices for the overlay graph construction (Step 2). This optimization not only accelerates the index construction and maintenance 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 prove the index correctness after this optimization in Theorem 2.

Theorem 2. *The boundary shortcuts generated in the MDE of Step 1 preserve global distances for the overlay graph.*

Proof. For any boundary pair $\forall s,t \in B_i$, we prove $d_G(s,t)$ is preserved in the overlay graph by two 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. □

*2) **Optimized Query Efficiency**:* As shown in Figure 7, the query efficiency of *PMHL* is continuously enhanced in different query stages. *BiDijkstra's* (Q-Stage 1) and *PCH* (Q-Stage 2) are search-based algorithms with relatively slow efficiency. By contrast, Q-Stage 3-5 adopt hop-based solutions (based on various PSP strategies) with much faster query efficiency. We next discuss them from the query-type perspective.

Same-Partition Query. As discussed in Section III-C, the no-boundary query (Q-Stage 3) has to concatenate $\tilde{L}$ and $\{L_i\}$ for same-partition queries, resulting in slow query processing. However, this query type appears frequently in real-life applications. For instance, it corresponds to city-level queries on province-level road networks. To this end, we adopt the post-boundary strategy to fix $\{L_i\}$, obtaining $\{L'_i\}$. As such, the same-partition queries are efficiently answered solely by L'_i in Q-Stage 4 and 5, *i.e.*, $\forall s,t \in G_i, q(s,t) = d_{L'_i}(s,t)$.

Algorithm 1: Cross-boundary PMHL Construction

```
1  Input: Overlay index L̃, partition indexes {L_i}
   Output: Cross-boundary labels L* = {X*(v)|v ∈ V}
2  v ← the highest-order vertex; A ← φ; //A is ancestor vector
3  TreeAggregation(v, A);                          ▷ In a top-down manner
4  Function TreeAggregation(v, A):
5      X*(v).A ← A; //ch is the children set
6      if v ∈ B then
7          X*(v).ch ← X̃(v).ch ∪ X_i(v).ch \ B; //ch is children set
8          X*(v).dis ← X̃(v).dis; X*(v).pos ← X̃(v).pos;
9      else
10         X*(v).ch ← X_i(v).ch \ B;
11         Compute X*(v).dis via X_i(v).N;      ▷ Compute distance labels
12     A.push_back(v);
13     for u ∈ X*(v).ch do
14         TreeAggregation(u, A);
15     A.pop_back();
```

Cross-Partition Query. Both no-boundary and post-boundary query processing remains slow as they require distance concatenation for *cross-partition queries* (which can be regarded as cross-province queries). To solve this, we can utilize the *cross-boundary strategy* discussed in Section IV-A to eliminate the distance concatenation for faster query processing for cross-partition queries, *i.e.*, $\forall s \in G_i, t \notin G_i, q(s,t) = d_{L^*}(s,t)$.

However, the naive implementation introduced in Section IV-A fails to leverage the tree decomposition for pruning useless label entries, resulting in unsatisfactory query efficiency. Worse yet, we cannot utilize the *star-centric* paradigm of *DH2H* [33] to trace the label changes during index maintenance, leading to slow index update efficiency. To this end, we propose a *tree decomposition aggregation* method to integrate the overlay tree $\tilde{T}$ and partition trees $\{T_i\}$ for a new cross-boundary tree decomposition T^* and corresponding cross-boundary index L^*. The pseudo-code is presented in Algorithm 1. Starting from the highest-order vertex, we compute L^* in a top-down manner. In particular, for $\forall v \in G_i$, if it is an overlay vertex, we set its children set $X^*(v).ch$ as the union of $\tilde{X}(v).ch$ and $X_i(v).ch \setminus B$ (non-boundary vertices in $X_i(v).ch$); otherwise we set $X^*(v).ch$ as $X_i(v).ch \setminus B$ (Line 10). This integration strategy prioritizes the node relationships of $\tilde{T}$, ensuring that the LCA of the L^* aligns with $\tilde{L}$'s LCA for cross-partition queries. Besides, T^* also inherits the subordinate relationships of the overlay and partition trees. Therefore, we can directly inherit the distance

and position arrays from $\tilde{L}$ for the overlay vertices (Line 8). For non-boundary vertex v, its distance array is computed by using the neighbor set of v's partition index as the vertex separator (Line 14), *i.e.*, $L^*(v,c) = \min_{u \in X_i(v).N}\{|sc(v,u)| + d_{L^*}(u,c)\}, \forall c \in X^*(v).A$, where $|sc(v,u)|$ is the shortcut length from v to u on G_i.

Example 7. *We illustrate the tree aggregation by referring to Figure 6. Sourcing from the highest vertex v_{14}, we perform the aggregation in a top-down manner. For v_{14}, it has one child v_{13} according to the union of its child set in $\tilde{T}$ and T_4. For v_{10}, its has two children in T^*: v_8 from $\tilde{T}$ and v_9 from T_1.*

Theorem 3. *L^* based on tree decomposition aggregation satisfies the 2-hop cover property for cross-partition queries.*

The proof of Theorem 3 is elaborated in our complete version [62] due to limited space. With the above optimizations on query processing, we can answer the same-partition query with $\{L'_i\}$ and cross-partition query with L^* based on the 2-hop cover property during Q-stage 5 in *PMHL*.

D. PMHL Index Maintenance

We next introduce the index update of *PMHL*.

U-Stage 1: On-spot Edge Update. As shown in Figure 7, given a batch of updates, we directly modify the edge weights on G, enabling *BiDijkstra's search* for query processing.

U-Stage 2: No-boundary Shortcut Update. During this stage, we update the shortcut arrays of the no-boundary index by referring to the *bottom-up shortcut update mechanism* [33]. To begin with, we classify the edge updates into two types: inter-edge updates U_{inter} and intra-edge updates U_{intra}. If U_{intra} is not empty, we first identify the affected partitions $\{G_i\}$ and maintain the affected $\{L_i\}$ in parallel. Meanwhile, the affected boundary shortcuts are identified and inserted into U_{inter}. After shortcut updates of $\{L_i\}$, we next update the $\tilde{L}$ based on U_{inter}. The highest affected tree nodes of $\{L_i\}$ and $\tilde{L}$ are recorded for the next stage's update. After this stage, we conduct *Partitioned CH* [35] query by searching the union of shortcut arrays of $\{L_i\}$ and $\tilde{L}$.

U-Stage 3: No-boundary Label Update. Given the highest affected tree nodes as input, this stage concurrently maintains the distance arrays of $\tilde{L}$ and $\{L_i\}$ via the *top-down label update mechanism* [33]. We also store the vertices with updated labels in an affected vertex set V_A for U-Stage 5. After that, *no-boundary query processing* with faster query processing is utilized in *Q-Stage 3*.

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

U-Stage 5: Cross-boundary Index Update. We update the cross-boundary index L^*. In particular, we first identify the affected partitions G_A based on the affected vertex set V_A generated in U-Stage 3. For each affected partition $G_j \in G_A$, we select the highest-order vertex $v \in G_j$ as a representative. 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 L^* in a top-down manner. After that, *Q-Stage 5* offers the fastest *cross-boundary query processing*.

Example 8. *The thread synchronization is illustrated at the top of Figure 6, where the red dash lines indicate the synchronization points. For U-Stage 2-4, each affected partition (or overlay graph) is allocated with one thread. For U-Stage 5, each root vertices in V_R are allocated with one thread.*

Algorithm 2: TD-Partitioning

```
1  Function TDPartition(T, τ, k_e, β_l, β_u):
2      cN[v] ← 1, ∀v ∈ V; //cN is partition size vector
3      for X(v) ∈ T in a bottom-up manner do
4          for u ∈ X(v).ch do
5              cN[v] ← cN[v] + cN[u];
6      V_C ← φ; // V_C is a vector storing the root candidates
7      for v ∈ V in decreasing vertex order do
8          if β_l · |V|/k_e ≤ cN[v] ≤ β_u · |V|/k_e and |X(v).N| ≤ τ then
9              V_C.push_back(v);
10     for v ∈ V_C do
11         if ∀u ∈ V_R, u is not the ancestor of v then
12             V_R ← V_R ∪ v;
13     Get the partition result G̃, {G_i} based on V_R;
14     return G̃, {G_i};
```

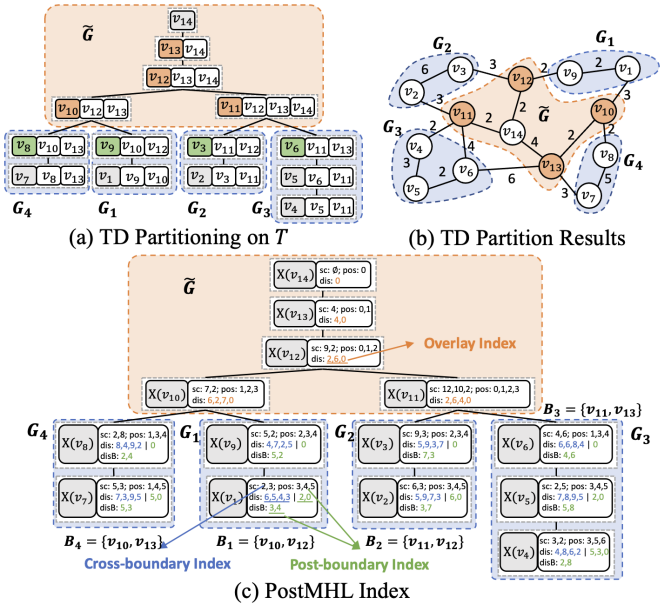
VI. Post-partitioned Multi-stage Hub Labeling

To further enhance query throughput, in this section, we propose a *Post-partitioned Multi-stage Hub Labeling (PostMHL)* based on a novel *tree decomposition-based graph partitioning*.

A. Tree Decomposition-based Graph Partitioning

As discussed in Section IV, the traditional graph partitioning methods are not designed towards good vertex ordering, thus failing to achieve efficient query processing. Conversely, can we obtain a suitable graph partitioning from a satisfactory vertex ordering? Observe that tree decomposition T generally produces a good vertex ordering for road networks, so why not leverage it to obtain graph partitioning? To this end, we design a *Tree Decomposition-based Partitioning (TD-Partitioning)* method to obtain partition results from a good vertex ordering. The 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. This is because $X(u).N$ can serve as a vertex separator (boundary vertices) for the descendants of $X(u)$ and all other tree nodes.

We present the pseudo-code of *TD-partitioning* in Algorithm 2. The input includes the tree decomposition $T = \{X(v)|v \in V\}$, bandwidth τ, expected partition number k_e, partition size imbalance ratio β_l and β_u. τ aims to limit the boundary vertex number for all partitions, as it can greatly affect the query efficiency of PSP index [35]. β_l and β_u constrain the imbalance of partition sizes such that workloads among threads are more balanced. Firstly, we calculate the



(a) TD Partitioning on T (b) TD Partition Results

(c) PostMHL Index

Fig. 8: TD Partitioning and PostMHL Index

descendent vertex number of each vertex $v \in V$ in a top-down manner from the root of T (Lines 2-5). Then, we calculate the root vertex candidates based on the constraints of bandwidth and partition size bounds (Lines 6-9). 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 that, we use the *minimum overlay strategy* to determine the root vertex for all partitions, obtaining the root vertex set V_R (Lines 12-12). This strategy aims to minimize the overlay graph size for faster overall index maintenance efficiency since overlay index maintenance cannot be paralleled. Lastly, we obtain the partition result 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 13). The *TD-partitioning* naturally fulfills the *boundary-first property* as the neighbor set of root vertex has higher vertex orders. Moreover, it allows the PSP index to utilize the vertex ordering generated by *MDE-based tree decomposition*, resulting in faster query efficiency.

Example 9. *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 $\{v_{10}, v_{12}\}$, $\{v_{11}, v_{12}\}$, $\{v_{11}, v_{13}\}$, and $\{v_{10}, v_{13}\}$. Figure 8-(b) shows the partition result with orange vertices representing the boundary vertices.*

B. PostMHL Index

The *TD-partitioning* method empowers the cross-boundary index with better query efficiency. However, updating PSP index is still laborious due to the sequentiality among PSP strategies and multiple tree decomposition. Fortunately, instead of designing the index from the partition's perspective like *PMHL*, which utilizes no-boundary or post-boundary strategies to guarantee correctness, Theorem 1 inspires us to re-design it reversely from the nonpartitioned index's perspective and further incorporate the PSP strategies for faster maintenance.

In particular, *PostMHL* amalgamates *overlay index*, *post-boundary index*, and *cross-boundary index* into one tree decomposition T. 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 8, 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. The post-boundary index is composed of the distance array entries to the in-partition ancestors and boundary arrays to the boundary vertices, 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 (as compared with *PMHL*) because the index construction and update of the post-boundary and cross-boundary indexes depend only on the overlay index, as proved in the following Theorem with its proof in our complete version [62] due to limited space.

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

Example 10. *Figure 8-(c) presents the PostMHL index with the overlay, post-boundary, and cross-boundary indexes in orange, green, and blue. The distance array of 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} has a distance array $X(v_{12}).dis = \{2, 6, 0\}$.*

To build the *PostMHL* index, we first utilize *MDE-based tree decomposition* [54] to obtain T and shortcut arrays for all tree nodes. After that, *TD-partitioning* is employed to obtain the graph partitions and the overlay index is built in a top-down manner like the *H2H* index. Then we build the post-boundary index for all partitions in parallel. In particular, given the partition G_i, we first compute the all-pair distances among the boundary vertices of G_i based on the overlay index and then store the results in a map table D for later reference. The post-boundary index is constructed in a top-down manner sourcing from the root vertex. 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. The specific pseudo-code is available in our complete version [62]. We analyze the space complexity and time complexity of *PostMHL* as follows:

Theorem 5. *The space complexity of PostMHL is $O(n \cdot h + n_p \cdot |B_{max}|)$ while the index construction complexity is $O(n \cdot (w^2 + log(n)) + n_o \cdot (\tilde{h} \cdot \tilde{w}) + \max_{i \in [1,k]}\{|V_i| \cdot (h \cdot w + |B_i| \cdot w)\})$, where h is the tree height, $|B_{max}|$ is maximal boundary vertex number of all partitions, n_o and n_p are the overlay*

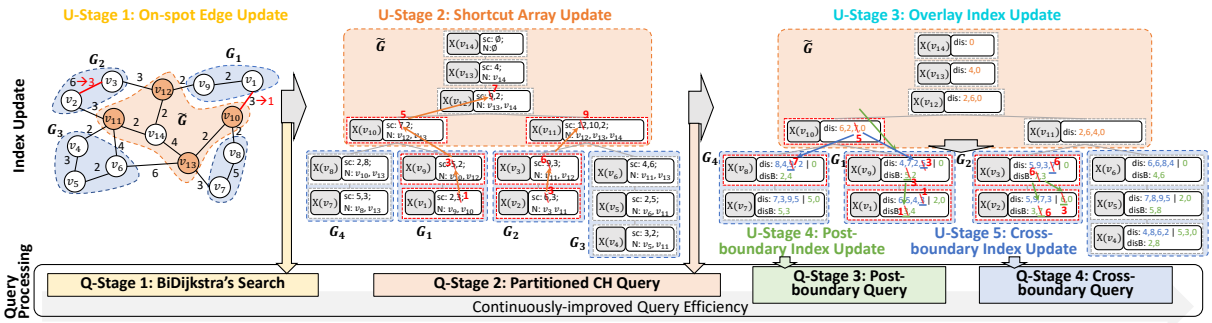


Fig. 9: Illustration of PostMHL-based Solution

and in-partition vertex number, $\tilde{h}$ is the maximum tree height of overlay vertices in T, w is treewidth. 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.

C. Framework of PostMHL

Now, we briefly introduce the framework of *PostMHL* with five update stages and four query stages as illustrated in Figure 9. The query processing of *PostMHL* is similar to the corresponding Q-Stages in *PMHL*. The U-Stage 1 and 2 of *PostMHL* are the same as those of *PMHL*. The U-Stage 3 is equivalent to the overlay label update of U-Stage 3 in *PMHL*. During 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 and cross-boundary index update. Based on Theorem 4, the post-boundary and cross-boundary index updates can be conducted in parallel after the overlay index, which further enhances the query throughput with the accelerated index update. Note that since it is more time-consuming to update the cross-boundary index than the post-boundary index due to a larger tree height of the overlay index, the post-boundary index is crucial for improving query throughput as it becomes available earlier.

Example 11. *We illustrate the PostMHL index update in Figure 9. U-Stage 1 directly refreshes edge weights when a batch of updates ($|e(v_2, v_3)| \rightarrow 3, |e(v_1, v_{10})| \rightarrow 1$) comes, and BiDijkstra's is used for query at first. In U-Stage 2, the in-partition shortcuts of affected partitions (G_1, G_2) are updated in parallel. Meanwhile, the affected overlay shortcuts are recorded and passed to the overlay update. Consequently, all affected shortcuts 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. Then 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.*

Remark 2. *PostMHL* further enhances the throughput of PMHL by two optimizations: 1) faster index maintenance by eliminating the no-boundary strategy and sharing the same tree decomposition for all indexes; 2) faster query processing by leveraging good vertex ordering obtained from TD-partitioning, achieving equivalent query efficiency with H2H.

TABLE I: Real-world Datasets

Name	Dataset	$\|V\|$	$\|E\|$	k	k_e	τ
NY	New York City[1]	264,346	730,100	8	32	100
GD	Guangdong[2]	938,957	2,452,156	8	32	100
FLA	Florida[1]	1,070,376	2,687,902	8	32	100
SC	South China[2]	1,326,091	3,388,770	32	64	200
EC	East China[2]	3,008,173	7,793,146	16	32	200
W	Western USA[1]	6,262,104	15,119,284	16	32	280
CTR	Central USA[1]	14,081,816	33,866,826	32	128	400
USA	Full USA[1]	23,947,347	57,708,624	32	128	400

[1] www.dis.uniroma1.it/challenge9/download.shtml; [2] https://www.navinfo.com/en;

TABLE II: Parameters (default values in bold)

Parameters	Values
Update Volume $\|U\|$	500, **1000**, 3000, 5000
Update Interval δt (s)	60, **120**, 300, 600
Query Response Time QoS R_q^* (s)	0.5, **1.0**, 1.5, 2

VII. Experiments

In this section, we evaluate the performance of the proposed methods. 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.

A. Experimental Setting

Datasets. We conduct experiments on eight real-world road networks from DIMACS [63] and NaviInfo [64]. Table I provides the dataset details and also shows the default value of partition number k of *PMHL* and bandwidth τ of *PostMHL*.

Algorithms. We compare our approaches (*PMHL* and *PostMHL*) with six baselines: *BiDijkstra* [11], *DCH* [32], *DH2H* [33], *N-CH-P* [35], *P-TD-P* [35], and *TOAIN* [37]. *DCH* and *DH2H* are state-of-the-art non-partitioned algorithms discussed in Section II. *N-CH-P*, *P-TD-P*, and *P-PTCP* are PSP indexes with exceptional index update and query processing performance. *TOAIN* is a throughput-optimizing index for dynamic kNN queries and can be used for SP computation by setting $k = 1$ for kNN. In particular, given a random query $q(s, t)$, we regard s as the query node while t is the nearest object. However, since *TOAIN* is designed for static road networks, we have to adapt it to dynamic networks by refreshing its shortcuts in updated networks. It is worth noting that to avoid system unresponsiveness, we also leverage index-free *BiDijkstra* to answer the queries for index-based baselines during index maintenance. We set $\beta_l = 0.1$, $\beta_u = 2$ for

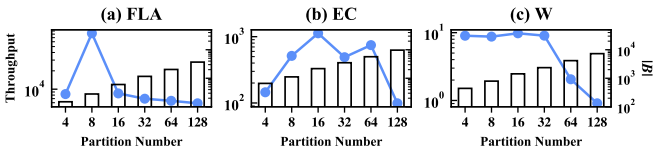


Fig. 10: Effect of Partition Number (Polyline: λ_q^*; Bar: $|B|$)

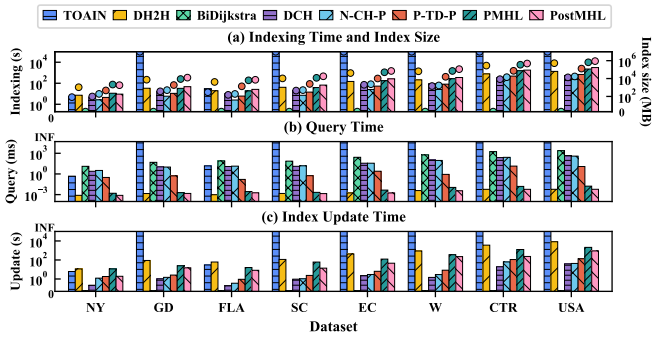


Fig. 11: Index Performance Comparison

PostMHL based on our preliminary experiments. The default thread number is set as 150 for all PSP algorithms.

Queries and Updates. We generate a query set Q in a Poisson process with the arrival rate of λ_q. We generate 10 batches of random updates for each dataset. For each update batch U, we follow [32], [39] to randomly select 10,000 edges and for each selected edge e, we decrease its edge weight to $0.5 \times |e|$ or increase to $2 \times |e|$. The default *update volume* $|U|$ and *update interval* δt are shown in bold in Table II.

Measurements. We use *average query response time* R_q^* as QoS and measure the *maximum average query throughput* λ_q^*. Similar to [37], we run the system on $10 \times \delta t$ seconds with a certain query arrival rate λ_q and gradually increase λ_q and repeat until QoS is violated or the system is overloaded (the updates U cannot be installed in δt). The default R_q^* is shown in Table II. We also report the average *index update time* T_u, *query time* T_q, *indexing time* T_c, and *index size* $|L|$.

B. Experimental Results

Exp 1: Effect of Partition Number. We first vary partition number k from 4 to 128 to evaluate its effect on *PMHL*. Figure 10 shows the results on SC, EC, and W, where both small and large k result in reduced throughput. $k = 8$ or 16 generally performs the best due to the balance between update workload and parallelization. A larger k can better exploit parallelization but increase the boundary number and tree height for the cross-boundary index, deteriorating the update efficiency of overlay and cross-boundary index. We also test the effect of bandwidth τ and k_e for *PostMHL*, which is in the Appendix [62] due to limited space. The default k, k_e, and τ are listed in the last three columns of Table I.

Exp 2: Index Performance Comparison. Next, we report the index performance $(t_c, |L|, t_q, t_u)$ in Figure 11. Note that we only report the results of TOAIN on NY and FLA since its index construction on other datasets exceeds 6 hours. As shown in (a), although t_c and $|L|$ of *PostMHL* are slightly larger than *DH2H* due to the addition of boundary arrays,

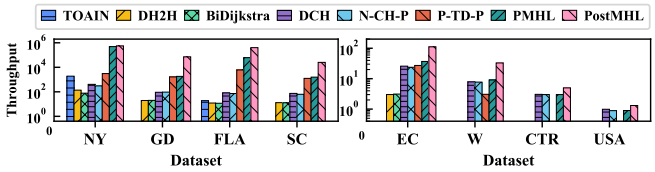


Fig. 12: Throughput Comparison on Different Datasets

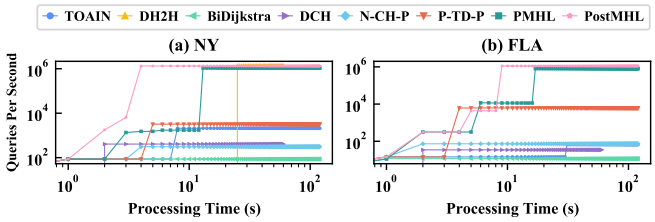


Fig. 13: Evolution of Queries Per Second

it still takes less than an hour to build *PostMHL* index. For query time, *PMHL* achieves up to $1000\times$ speedup over *P-TD-P* which only relies on the post-boundary strategy, validating the effectiveness of cross-boundary strategy. Besides, due to the thread parallelization among partition indexes, the index update efficiency of *PostMHL* is about $3.9-15.5\times$ faster than *DH2H*, while the query efficiency is almost the same.

Exp 3: Throughput Comparison. We next report the performance of throughput λ_q^* in Figure 12. It is worth noting that we especially set $\delta t = 600$ and $R_q^* = 5$ for CTR and USA since all algorithms suffer from almost zero throughput in the default setting. Nevertheless, we argue that in real-life applications, concurrent query processing (using multiple threads (workers) for query processing) could be leveraged to enhance the throughput and meet the real-time response for these super-large road networks (we also provide the preliminary experimental results in the Appendix [62]).

As shown in Figure 12, *PostMHL* and *PMHL* outperform all baselines with up to 2 orders of magnitude higher query throughput than the best applicable baselines. It is worth noting that even with state-of-the-art query efficiency and the help of *BiDijkstra*'s query processing during the index maintenance, the throughput of *DH2H* is still low due to its slow maintenance. *TOAIN* performs well on NY due to its adaptability in trading off query and update. *P-TD-P* and *DCH* are the best baselines in medium-sized or large road networks due to a good tradeoff between the index update and query time. Nevertheless, *PostMHL* always has the best throughput.

Exp 4: Evolution of Queries Per Second (QPS). To better understand why our methods have better throughput, we report the evolution of *QPS* (equals to $1/t_q$) during the update interval (120s) on NY and FLA. As shown in Figure 13, as time goes by, *PostMHL* (or *PMHL*) generally has the best query efficiency at any time point because of the three key designs: 1) *multi-stage scheme* to continuously enhance query efficiency; 2) *cross-boundary strategy* for exceptional query efficiency; 3) multi-thread parallelization for fast maintenance. Besides, *PostMHL* outperforms *PMHL* in most cases due to the faster maintenance and cross-boundary query. Since *PostMHL* generally has the best performance, we use it as the representative of our methods in the following experiment.

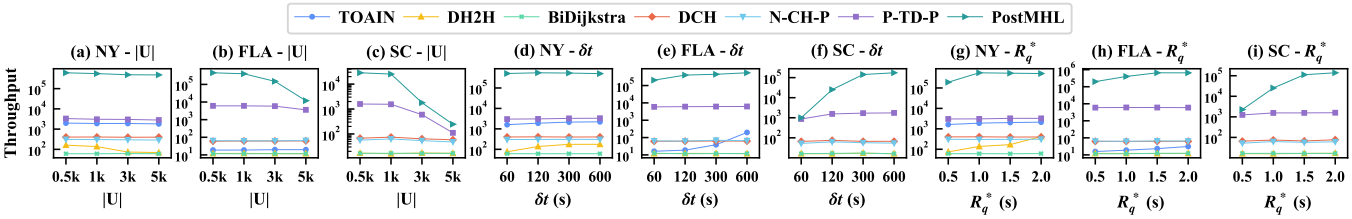


Fig. 14: Throughput Performance When Varying Parameters

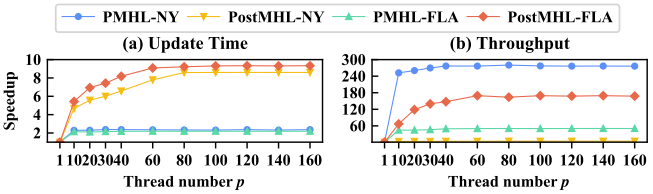


Fig. 15: Speedup When Varying Thread Number

Exp 5: Effect of Update Volume, Interval, and Response Time. We vary $|U|$, δt, and R_q^* to evaluate the effectiveness of *PostMHL* under different update scenarios. As shown in Figure 14 (a)-(c), with the increase of update volume $|U|$, the throughput of most SP indexes tends to decrease due to longer index maintenance. Nevertheless, *PostMHL* consistently outperforms all baselines with $2.2-182.9\times$ speed up. Figure 14 (d)-(f) show the throughput of most algorithms (*BiDijkstra*, *DCH*, *N-CH-P*) remains stable with the increase of update interval δt, demonstrating that they have reached their best performance. By contrast, the throughput of *PostMHL* rises dramatically for SC with a longer δt due to the engagements of faster query processing. Figure 14 (g)-(i) show the result of varying response time R_q^*. Similar to δt, the throughput of most baselines remains steady under larger R_q^*, while PostMHL generally has a significant throughput improvement. Overall, *PostMHL* generally outperforms baseline and exhibits remarkable throughput improvement in most cases.

Exp 6: Effect of Thread Number p. Figure 15 presents the performance speedup when varying p from 1 to 160, where each legend is a combination of an algorithm (*PMHL* or *PostMHL*) and a dataset (NY or FLA). The update speedup increases with the number of threads but reaches a plateau due to the unparalleled overlay index update and limited partition number. *PostMHL* has a higher speedup ($9.3\times$) than *PMHL* ($2.3\times$) since its partition number ($k_e = 32$) is larger than *PMHL*'s ($k = 8$). As shown in Figure 15-(b), the tendency of throughput speedup is similar to update time: rise dramatically first and then reach a plateau. In particular, the throughput of *PMHL* has up to $280\times$ speed up. This is because the initial throughput of *PMHL* (when $p = 1$) is quite low while a faster update time powered by more threads could vacate more time for efficient cross-boundary query processing.

VIII. Related Work

We discuss related works by classifying them into two types:

Shortest Path Algorithms. The index-free algorithms like *Dijkstra's* [10], A^* [65], and *BiDijsktra* [11] search the graph on the fly so they are naturally dynamic [12], [13], [66] but they are slow for query. To index-based methods were proposed [14]–[17], [22], [23], [43], [67]–[69] to achieve 1-5 orders of magnitude of improvement, with *H2H* [22] and *P2H* [23] performing best on road networks. To adapt to dynamic scenarios, index maintenance [25]–[36] has also been studied. Among them, *DCH* [32] and *DH2H* [33] are two state-of-the-art methods. It is worth mentioning that *DCH* [32] outperforms popular industrial solution *CRP* [44] in update time. However, most of these methods only focus on reducing query time or update time, ignoring that high throughput on dynamic networks requires both high query and update efficiency. It is worth noting that the throughput of kNN queries has also been studied in previous works [37], [38]. *TOAIN* [37] is an adaptive algorithm for optimizing the throughput of kNN queries based on a multi-level CH index called *SCOB*. However, they were designed for static networks and using them for random SP calculations is wasteful.

Partitioned Shortest Path Index. To scale up to large graphs, the PSP index [43]–[47], [49], [51], [58]–[60], [70], [71] has also been investigated and can be classified into three types based on partition structure [35]: the *Planar PSP* [44], [72], [73] treats all partitions equally on one level, such as *TNR* [72], *CRP* [44], *PCH* [35] and *PH2H* [35]. The *Hierarchical PSP* such as *G-Tree* [45] and *ROAD* [60] organizes the partitions hierarchically and each level is a planar partition. *Core-Periphery PSP* [36], [43], [74] treats the partitions discriminately by taking some important vertices as "Core" and the remaining ones as "Peripheries", which is more suitable for small-world networks. To adapt them to dynamic networks, [35] puts forward three general PSP strategies and a universal scheme for designing the dynamic PSP index. Nevertheless, these dynamic PSP indexes sacrifice the query efficiency, thus failing to achieve high throughput query processing.

IX. Conclusion

In this paper, we investigate high throughput query processing on dynamic road networks. We first propose a general *cross-boundary PSP strategy* and provide an insightful analysis of the *upper bound of PSP index query efficiency*. Then we propose the *PMHL* index to leverage thread parallelization and multiple PSP strategies for fast index maintenance and continuously improved query efficiency. Lastly, we propose *PostMHL* by adopting *TD partitioning* to achieve the same query efficiency with *DH2H* but with a much faster index update. The experiments demonstrate that *PostMHL* can yield up to 2 orders of magnitude throughput improvement compared to the state-of-the-art baselines, making hop-based solutions more practical for real-life applications.

REFERENCES

[1] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," *Algorithm engineering: Selected results and surveys*, pp. 19–80, 2016.

[2] H. Li, Y. Ge, R. Hong, and H. Zhu, "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*, 2016, pp. 975–984.

[3] B. Cao, L. Alarabi, M. F. Mokbel, and A. Basalamah, "Sharek: A scalable dynamic ride sharing system," in *2015 16th IEEE International Conference on Mobile Data Management*, vol. 1. IEEE, 2015, pp. 4–13.

[4] J. Ye, "Big data at didi chuxing," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2018, pp. 1341–1341.

[5] "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.

[6] U. B. of Transportation Statistics. Transportation statistics annual report. [Online]. Available: https://www.bts.gov/tsar

[7] Y. Zheng, Y. Liu, J. Yuan, and X. Xie, "Urban computing with taxicabs," in *Proceedings of the 13th international conference on Ubiquitous computing*, 2011, pp. 89–98.

[8] J. Zhang, C. Kamga, H. Gong, and L. Gruenwald, "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*, 2012, pp. 163–171.

[9] H. Qu and A. Labrinidis, "Preference-aware query and update scheduling in web-databases," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2006, pp. 356–365.

[10] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[11] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *The computer journal*, vol. 9, no. 3, pp. 275–280, 1966.

[12] L. Li, M. Zhang, W. Hua, and X. Zhou, "Fast query decomposition for batch shortest path processing in road networks," in *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 2020, pp. 1189–1200.

[13] M. Zhang, L. Li, W. Hua, and X. Zhou, "Stream processing of shortest path query in dynamic road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 5, pp. 2458–2471, 2020.

[14] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *International workshop on experimental and efficient algorithms*. Springer, 2008, pp. 319–333.

[15] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[16] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "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, 2011, pp. 230–241.

[17] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *European Symposium on Algorithms*. Springer, 2012, pp. 24–35.

[18] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu, "Hop doubling label indexing for point-to-point distance querying on scale-free networks," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, 2014.

[19] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong, "Is-label: an independent-set based labeling scheme for point-to-point distance querying," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, 2013.

[20] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao, "The exact distance to destination in undirected world," *The VLDB Journal*, vol. 21, pp. 869–888, 2012.

[21] F. Wei, "Tedi: efficient shortest path query answering on graphs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 99–110.

[22] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu, "When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 709–724.

[23] Z. Chen, A. W.-C. Fu, M. Jiang, E. Lo, and P. Zhang, "P2h: efficient distance querying on road networks by projected vertex separators," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 313–325.

[24] M. Farhan, H. Koehler, R. Ohms, and Q. Wang, "Hierarchical cut labelling-scaling up distance queries on road networks," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–25, 2023.

[25] T. Akiba, Y. Iwata, and Y. Yoshida, "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*, 2014, pp. 237–248.

[26] G. D'angelo, M. D'emidio, and D. Frigioni, "Fully dynamic 2-hop cover labeling," *Journal of Experimental Algorithmics (JEA)*, vol. 24, pp. 1–36, 2019.

[27] Y. Qin, Q. Z. Sheng, N. J. Falkner, L. Yao, and S. Parkinson, "Efficient computation of distance labeling for decremental updates in large dynamic graphs," *World Wide Web*, vol. 20, pp. 915–937, 2017.

[28] M. Zhang, L. Li, W. Hua, and X. Zhou, "Efficient 2-hop labeling maintenance in dynamic small-world networks," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 133–144.

[29] M. Zhang, L. Li, G. Trajcevski, A. Zufle, and X. Zhou, "Parallel hub labeling maintenance with high efficiency in dynamic small-world networks," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[30] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[31] V. J. Wei, R. C.-W. Wong, and C. Long, "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*, 2020, pp. 1841–1856.

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

[33] M. Zhang, L. Li, W. Hua, R. Mao, P. Chao, and X. Zhou, "Dynamic hub labeling for road networks," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 336–347.

[34] M. Farhan, Q. Wang, and H. Koehler, "Batchhl: Answering distance queries on batch-dynamic networks at scale," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2020–2033.

[35] M. Zhang, X. Zhou, L. Li, Z. Liu, G. Trajcevski, Y. Huang, and X. Zhou, "A universal scheme for partitioned dynamic shortest path index," *arXiv preprint arXiv:2310.08213*, 2023.

[36] X. Zhou, M. Zhang, L. Li, and X. Zhou, "Scalable distance labeling maintenance and construction for dynamic small-world networks," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024.

[37] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng, "Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 594–606, 2018.

[38] D. He, S. Wang, X. Zhou, and R. Cheng, "An efficient framework for correctness-aware knn queries on road networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1298–1309.

[39] M. Zhang, L. Li, and X. Zhou, "An experimental evaluation and guideline for path finding in weighted dynamic network," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2127–2140, 2021.

[40] T. Maps. Intermediate traffic api - introduction. [Online]. Available: https://developer.tomtom.com/intermediate-traffic-service/documentation/product-information/introduction?source_app=b2b&source_product=traffic-apis

[41] A. Sharma, V. Ahsani, and S. Rawat, "Evaluation of opportunities and challenges of using inrix data for real-time performance monitoring and historical trend assessment," 2017. [Online]. Available: https://lib.dr.iastate.edu/ccee_reports/24

[42] R. Xiaofeng. Connecting the real world: Algorithms and innovations behind amap. [Online]. Available: https://www.alibabacloud.com/blog/connecting-the-real-world-algorithms-and-innovations-behind-amap_596304

[43] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling up distance labeling on graphs with core-periphery properties," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1367–1381.

[44] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning in road networks," *Transportation Science*, vol. 51, no. 2, pp. 566–591, 2017.

[45] R. Zhong, G. Li, K.-L. Tan, and L. Zhou, "G-tree: An efficient index for knn search on road networks," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 39–48.

[46] S. Wang, X. Xiao, Y. Yang, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 61–72, 2016.

[47] L. Li, S. Wang, and X. Zhou, "Time-dependent hop labeling on road network," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 902–913.

[48] X. Zhou, K. Huang, L. Li, M. Zhang, and X. Zhou, "I/o-efficient multi-criteria shortest paths query processing on large graphs," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[49] L. Li, S. Wang, and X. Zhou, "Fastest path query answering using time-dependent hop-labeling in road network," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[50] Z. Yu, X. Yu, N. Koudas, Y. Liu, Y. Li, Y. Chen, and D. Yang, "Distributed processing of k shortest path queries over dynamic road networks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 665–679.

[51] Z. Liu, L. Li, M. Zhang, W. Hua, P. Chao, and X. Zhou, "Efficient constrained shortest path query answering with forest hop labeling," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1763–1774.

[52] J. W. Cohen, *The single server queue*. Elsevier, 2012.

[53] A. Berry, P. Heggernes, and G. Simonet, "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, 2003, pp. 58–70.

[54] J. Xu, F. Jiao, and B. Berger, "A tree-decomposition approach to protein structure prediction," in *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*. IEEE, 2005, pp. 247–256.

[55] M. A. Bender and M. Farach-Colton, "The lca problem revisited," in *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*. Springer, 2000, pp. 88–94.

[56] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, vol. 20, no. 1, pp. 359–392, 1998.

[57] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, "Graph partitioning with natural cuts," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 1135–1146.

[58] Z. Liu, L. Li, M. Zhang, W. Hua, and X. Zhou, "Fhl-cube: multi-constraint shortest path querying with flexible combination of constraints," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 3112–3125, 2022.

[59] Z. Liu, L. Li, M. Zhang, W. Hua, and X. Zhou, "Multi-constraint shortest path using forest hop labeling," *The VLDB Journal*, pp. 1–27, 2022.

[60] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian, "Road: A new spatial object search framework for road networks," *IEEE transactions on knowledge and data engineering*, vol. 24, no. 3, pp. 547–560, 2010.

[61] B. Zheng, Y. Ma, J. Wan, Y. Gao, K. Huang, X. Zhou, and C. S. Jensen, "Reinforcement learning based tree decomposition for distance querying in road networks," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1678–1690.

[62] High throughput shortest path query processing for large dynamic road networks (full version). [Online]. Available: https://www.dropbox.com/scl/fo/e2f57v59zsu74mrfovfzm/h?rlkey=05fc6tiinxy5k9qiknpxioo98&st=npfn7r48&dl=0

[63] "9th dimacs implementation challenge - shortest paths," www.dis.uniroma1.it/challenge9/download.shtml.

[64] "Navinfo," https://en.navinfo.com/.

[65] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[66] J. R. Thomsen, M. L. Yiu, and C. S. Jensen, "Effective caching of shortest paths for location-based services," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 313–324.

[67] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *ESA*, vol. 3669. Springer, 2005, pp. 568–579.

[68] T. Akiba, Y. Iwata, and Y. Yoshida, "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*, 2013, pp. 349–360.

[69] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling distance labeling on small-world networks," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1060–1077.

[70] Z. Li, L. Chen, and Y. Wang, "G*-tree: An efficient spatial index on road networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 268–279.

[71] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," *Journal of Experimental Algorithmics (JEA)*, vol. 21, pp. 1–49, 2016.

[72] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007, pp. 46–59.

[73] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling, "Fast point-to-point shortest path computations with arc-flags," *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, pp. 41–72, 2009.

[74] Y. Wang, Q. Wang, H. Koehler, and Y. Lin, "Query-by-sketch: Scaling shortest path graph queries on very large networks," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1946–1958.

X. Appendix

A. Index Update of No-boundary and Post-boundary Strategy

As introduced in Section III-C, the index update of no-boundary and post-boundary strategies are similar to their index construction by replacing the build with the update. Nevertheless, we introduce their update procedures as follows.

Index Update for No-boundary Strategy. Since the weight change of inter-edges does not affect the index for each subgraph, we divide index updates into two scenarios as shown in Figure 16. Scenario 1: Inter-edge weight change. When $e \in E_{inter}$ changes, only $\tilde{L}$ needs an update; Scenario 2: Intra-edge weight change. When $e \in E_{intra}$ ($e \in E_j$) changes, we first update L_j and compare the old and new weights of $e(b_{j1}, b_{j2})$ between boundary in G_j. If there is an edge weight update, we need to further update $\tilde{L}$.

Index Update for Post-boundary Strategy. It is similar to *No-Boundary* with an additional judgment and processing shown in Figure 16. Scenario 1: Intra-edge weight change. Suppose $e \in E_i$ changes, we update G_i, L_i and then update $\tilde{G}, \tilde{L}$ if any $d_{L_j}(b_{j1}, b_{j2})$ changes. Then we update $\{G'_i\}, \{L'_i\}$ if $d_{G'_i}(d_{i1}, d_{i2})$ and $d_{\tilde{L}}(d_{i1}, d_{i2})$ are different; Scenario 2: Inter-edge weight change. Suppose $e \in E_{inter}$ changes, we update $\tilde{G}, \tilde{L}$ and then update $\{G'_i\}, \{L'_i\}$.

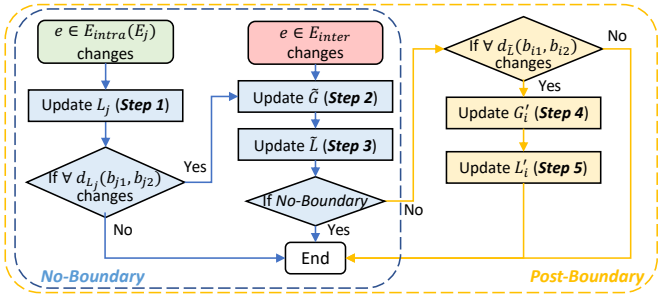


Fig. 16: Index Update of No-boundary and Post-boundary Strategy

B. Pseudo-code of PMHL Index Construction

The pseudo-code of *PMHL* index construction is presented in Algorithm 3. *PMHL* first leverages graph partitioning method *PUNCH* [57] to divide the original road network G into k subgraphs (Line 1). Given the partition results as input, a *boundary-first* vertex ordering method [58] 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 V-C for details).

C. Pseudo-code of PostMHL Index Construction

The pseudo-code of *PostMHL* index construction is presented in Algorithm 4. We first utilize *MDE-based tree decomposition* [54] to obtain T and shortcut arrays for all tree nodes (Line 1). After that, *TD-partitioning* is employed to obtain the graph partition results, and the overlay index is built in a top-down manner like the *H2H* 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 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 according to the *minimum distance property* [33], *i.e.*, $d(v,b) = \min_{u \in X(v).N}\{sc(v,u) + d(u,b)\}, \forall v \in G_i, b \in B_i \cup X(v).A$ (Lines 9-31). Lastly, the cross-boundary index is constructed parallelly in a top-down manner from each partition's root vertex just like the overlay index construction (Lines 32-33).

D. Proofs

Proof of Theorem 3. We first prove the distance array of L^* is correct in 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 distances from v to its ancestor $u \notin G_i$ computed by the top-down label construction are correct as it essentially uses B_i as the vertex separator. Meanwhile, the distances from v to its ancestor $u \in G_i$ are also accurate, as it uses the neighbor set of v's partition index as the vertex separator.

We next prove $\forall s \in G_i, t \in G_j, i \neq j$, the LCA of $X^*(s)$ and $X^*(t)$ is the vertex separator of s and t in 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 $sp_c =$

Algorithm 3: PMHL Index Construction

Input: Road network $G = \{V, E\}$
Output: PMHL index $L_{PMHL} = \{\tilde{L}, \{L_i\}, \{L'_i\}, L^*\}$
1 $\{G_i | 1 \leq i \leq k\} \leftarrow$ Partitioning G by PUNCH [57]; ▷ Get partition graphs
2 $r \leftarrow$ BoundaryFirstOrder($\{G_i\}$); ▷ Boundary-first vertex ordering
3 // No-boundary Index Construction
4 **parallel_for** $i \in [1, k]$
5 $\quad L_i \leftarrow$ MHLIndexing(G_i); ▷ Step 1: Build partition indexes $\{L_i\}$
6 $\tilde{G} \leftarrow$ OverlayGraphBuild($\{L_i\}$); ▷ Step 2: Build overlay graph
7 $\tilde{L} \leftarrow$ MHLIndexing($\tilde{G}$); ▷ Step 3: Build overlay index $\tilde{L}$
8 // Post-boundary Index Construction
9 **parallel_for** $i \in [1, k]$
10 $\quad G'_i \leftarrow$ GetExtendedGraph($\tilde{L}, G_i$); ▷ Step 4: Build extended partitions
11 $\quad L'_i \leftarrow$ MHLIndexing(G'_i); ▷ Step 5: Build post-boundary index $\{L'_i\}$
12 // Cross-boundary Index Construction, see Algorithm 1
13 $L^* \leftarrow$CrossIndexBuild($\tilde{L}, \{L_i\}$); ▷ Step 6: Build cross-boundary index L^*
14 **return** $L_{PMHL} = \{\tilde{L}, \{L_i\}, \{L'_i\}, L^*\}$;

Algorithm 4: 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 $L = \{X(v)|v \in V\}$

```
T ← TreeDecomposition(G);                          ▷ MDE-based tree decomposition
G̃, {G_i | i ∈ [1,k]} ← TDPartition(T, τ, k_e, β_l, β_u);  ▷ TD-partitioning
OverlayIndexing(G̃, L);                             ▷ Top-down overlay index construction
// Post-boundary index construction
parallel_for i ∈ [1,k]
    u ← root vertex of G_i;
    for b_1 ∈ X(u).N and b_2 ∈ X(u).N do
        D[b_1][b_2] ← Q(b_1, b_2);                 ▷ Get all-pair boundary distances
    for v ∈ G_i and X(v) ∈ T in a top-down manner do
        Suppose X(v).N = (x_1, x_2, ...);
        for j = 1 to |X(v).N| do
            X(v).pos[j] ←the position of x_j in X(v).A;
        // Compute boundary array
        for j = 1 to |X(u).N| do
            X(v).disB[j] ← ∞;
            for k = 1 to |X(v).N| do
                if x_k ∈ G̃ then d ← X(v).sc[k] + D[x_j][x_k];
                else d ← X(v).sc[k] + X(x_k).disB[j];
                X(v).disB[j] ← min{X(v).disB[j], d};
        // Compute distance array
        for j = 1 to |X(v).A| − 1 do
            if X(v).A[j] ∈ G̃ then continue;        ▷ Prune overlay ancestors
            c ← X(v).A[j]; X(v).dis[j] ← ∞;
            for k = 1 to |X(v).N| do
                if x_k ∈ G̃ then
                    l ← position of x_k in X(u).N;
                    d ← X(c).disB[l];
                else
                    if X(v).pos[k] > j then d ← X(x_k).dis[j];
                    else d ← X(c).dis[X(v).pos[k]];
                X(v).dis[j] ← min{X(v).dis[j], X(v).sc[k] + d};
        X(v).dis[|X(v).A|] ← 0;
parallel_for i ∈ [1,k]
    CrossPartiIndexing(G_i, L);                    ▷ Cross-boundary index construction
return L;
```

$\langle s = b_0, \ldots, b_n, t\rangle$ ($b_i \in B, 0 \le i \le n$). 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 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 $sp_c = \langle s, b_0, \ldots, b_n, t\rangle$ ($b_i \in B, 0 \le i \le n$). 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 $O(n \cdot h)$ [22] plus boundary array size $O(n_p \cdot |B_{max}|)$, where $|B_{max}| = \max_{i\in[1,k]}\{|B_i|\}$ is maximal boundary vertex number of all partitions, n_p is the in-partition vertex number. Hence, index size is $O(n \cdot h + n_p \cdot |B_{max}|)$. The indexing time of *PostMHL* consists of tree decomposition $O(n \cdot (w^2 + \log(n)))$, TD-partitioning $O(n \log(n))$, overlay indexing $O(n_o \cdot (\tilde{h} \cdot \tilde{w}))$, thread-parallel post-boundary and cross-boundary indexing $O(\max_{i\in[1,k]}\{n_i \cdot (h \cdot w + |B_i| \cdot w)\})$. As for index update, according to [33], 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 post-boundary and cross-boundary index updates can be conducted in parallel for all partitions. □

E. Additional Experimental Results

Exp 7: Effect of Partition Number k_e for PostMHL. We evaluate the effect of partition number k_e for *PostMHL*. Figure 17 presents the throughput λ_q^* and update time t_u on datasets FLA, EC, and W. Similar to PMHL, both small and large k_e result in reduced throughput. In general, $k_e = 32$ performs the best due to the balance between update workload and parallelization. A larger k_e can better exploit parallelization but increase the overlay vertex number due to smaller partition size, deteriorating the update efficiency of overlay and cross-boundary index.

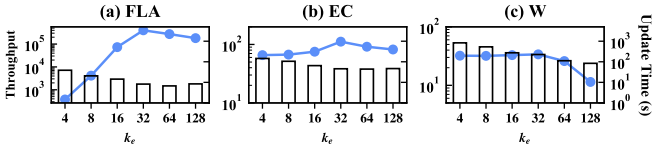


Fig. 17: Effect of k_e for PostMHL (Polyline: λ_q^*, Bar: t_u)

Exp 8: Effect of Bandwidth for PostMHL. We further demonstrate the effect of bandwidth τ *PostMHL*. Figure 18 presents the results on datasets NY and FLA. In particular, Figure 18 (a)-(b) shows the overlay vertex number and query time of Q-Stage 3. Observe that the increase in bandwidth leads to smaller overlay vertex numbers since a large τ leads to more vertices becoming the partition vertex. However, a large τ deteriorates the query efficiency of Q-Stage 3 (post-boundary query). This is due to 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 (*i.e.,* bandwidth). It is worth noting that the change of bandwidth could not affect the query efficiency of Q-Stage 1, 2, and 4 of PostMHL since the graph that *BiDijkstra* and *PCH* search on and the *cross-boundary index* do not change. Figure 18 (c)-(d) depicts the update time and query throughput. We can see that a small τ tends to deteriorate the update time and query throughput. This is because a small bandwidth leads to a larger overlay graph, whose index maintenance cannot be accelerated by multi-thread parallelization, thus reducing the index update

efficiency and throughput. Therefore, it is generally advisable to set a large bandwidth for PostMHL even though a large τ could undermine the post-boundary query processing.

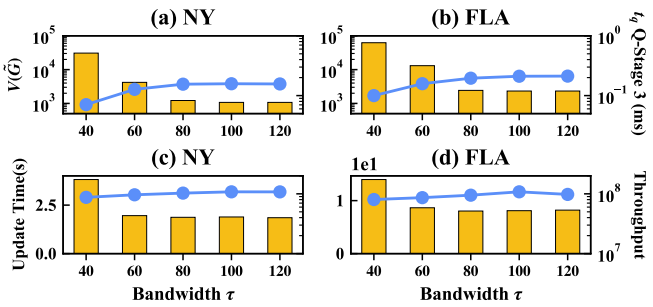


Fig. 18: Effect of τ for PostMHL (Bar: left axis, Polyline: right axis)