9 APPENDIX

9.1 Pseudo-code of MHL Index Construction

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

Algorithm 4: MHL Index Construction

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

9.2 Pseudo-code of PMHL Index Construction

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

Algorithm 5: PMHL Index Construction

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

9.3 Proofs

Proof of Lemma 1. As the vertex contraction process of CH and H2H is identical, the shortcuts produced by them are naturally equivalent, provided that they are executed in the same order (*e.g.,* using MDE [10, 58]). □

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4 Additional Experimental Results

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

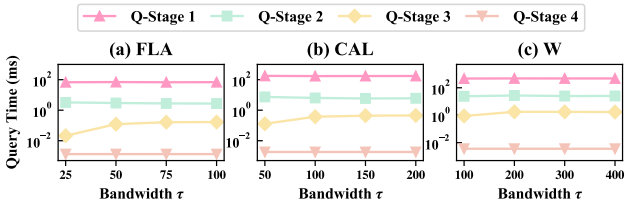

Figure 16: Effect of τ for PostMHL's Query Stages

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

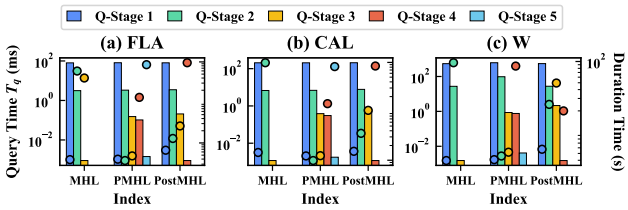

Figure 17: Performance of Different Query Stages