9 APPENDIX

9.1 Pseudo-code of VPL Index Construction

The pseudo-code of *VPL* construction is presented in Algorithm 3. Sepcifically, we adopt *PUNCH* [14] as the graph partitioning method to divide G into multiple subgraphs $\{G_i|1 \leq i \leq k\}$ (Line 1), because it generates the minimal boundary number on road networks [67] and its partition result complies well with the query clusters. Given the partition result, we first derive its order r via *tree height-aware PSP ordering* (will be introduced in Section 5.2). A tree decomposition T and the *shortcut arrays* of its tree nodes are then built based on r (Line 3). As for the label construction, we first build the overlay index for boundary vertices from the root of T (Lines 4-5) and then construct the labels for all partitions' non-boundary vertices in parallel (Lines 6-8). For each tree node $X(v)$, we first obtain the position array by identifying the position of any vertex $x_j \in X(v)$ in $X(v).A$ (Lines 12-13). The distance array is computed by using $X(v).N$ as the vertex separator while the boundary array is computed by inheriting corresponding label entries in $X(v).dis$.

Algorithm 3: VPL Index Construction

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

9.2 Pseudo-code of DVPL Shortcut Maintenance

Algorithm 4 presents the pseudo-code of mixed shortcut maintenance. We use the non-partitioned L to demonstrate the key idea of *DVPL* while it is easy to extend it to a partitioned one with parallelization. In particular, we use a min-heap set Q to record affected tree nodes, and a min-heap set $S(a)$ to record neighbors of a' affected shortcuts, ensuring that the updates are processed in a bottom-up manner. When Q is not empty, we iteratively pop the vertex a with the minimum order from Q, and handle its affected shortcuts $sc(a,b), \forall b \in S[a]$ (Lines 2-8). For each affected shortcut, we compute its new value d_{sc} and supportive number ϕ_{sc} via *minimum weight property* [62] (Line 5). For example, as shown in Figure 9-(a), suppose $sc(a,b)$ has two supportive vertices v_1, v_2 and the edge $e(a,b)$ decreases from 4 to 3, we have $d_{sc} = 3$ and $\phi_{sc} = 1$ since only edge $e(a,b)$ makes new $sc(a,b)$ holds after update. We compare $sc(a,b)$ with d_{sc} to figure out its update types, and assign Cw with old $sc(a,b)$ or new d_{sc} for the increase and decrease (Lines 6-7). After refreshing $sc(a,b)$ and $\phi_{sc}(a,b)$, we propagate the update in function SCUpdatePropagate, where we first identify the affected labels and then detect other affected shortcuts. We next take the decrease as an example (Lines 10-14). We first identify the affected distance labels in function DecUpdateCheck, *i.e.,* for each $u \in X(a).A$, if $L(a,u) > Cw + L(b,u)$ or $L(a,u) > Cw + L(u,b)$, we say the label $L(a,u)$ is affected and use $C(a,u)$ to record this information, where $C(a,u) > 0$ indicates increase update while $C(a,u) < 0$ represents decrease update. We recognize $X(a)$ as affected if there is any $L(a,u)$ that is affected. We also set the boolean value $X(a).ifU$ as true if there is any $L(a,ut)$ that is affected. As illustrate in Figure 9-(b), $L(a,u_1)$ is affected since $Cw + L(u_1,b) = 3+2 = 5 < L(a,u_1)$ while $L(a,u_2)$ is unaffected. We then check whether each neighboring shortcut $sc(b,u), u \in X(b).N$ or $sc(u,b), b \in X(u).N$ is affected in function DecSCCheck, where $sc(u,b)$ is recorded as affected if new $dis < sc(u,b)$ (Lines 20-22).

Algorithm 4: DVPL: Mixed Shortcut Maintenance

Input: G and its mixed updates U, L
Output: Shortcut Updated L
1 $Q.insert(a), S[a].insert(b), \forall e(a,b) \in U$; //$Q$ is a min-heap set, $S[a]$ is a set for a
2 **while** Q *is not empty* **do**
3 $a \leftarrow Q.begin(), Q.erase(a)$; //$a$ has the minimum vertex order in Q
4 **for** *each* $b \in S[a]$ **do**
5 Compute new shortcut d_{sc} and ϕ_{sc} by *minimum weight property* [62]
6 **if** $d_{sc} > sc(a,b)$ **then** $Cw \leftarrow sc(a,b)$; //increase update
7 **else if** $d_{sc} < sc(a,b)$ **then** $Cw \leftarrow d_{sc}$; //decrease update
8 $sc(a,b) \leftarrow d_{sc}; \phi_{sc}(a,b) \leftarrow \phi_{sc}$; SCUpdatePropagate($a, b, Cw$);
9 **Function** *SCUpdatePropagate*(a, b, Cw):
10 **if** $sc(a,b)$ *is decrease update* **then**
11 DecLabelCheck(a, b, Cw); ▷ Identify affected labels
12 **for** *each* $u \in X(a).N$ **do**
13 **if** $u \in X(b).N$ **then** DecSCCheck($b, u, a, Cw + sc(a,u)$);
14 **else if** $b \in X(u).N$ **then** DecSCCheck($u, b, a, Cw + sc(a,u)$);
15 **else if** $sc(a,b)$ *is increase update* **then**
16 **if** $X(a).FN[b] = true$ **then** IncLabelCheck(a, b, Cw);
17 **for** *each* $u \in X(a).N$ **do**
18 **if** $u \in X(b).N$ **then** IncSCCheck($b, u, Cw + sc(a,u)$);
19 **else if** $b \in X(u).N$ **then** IncSCCheck($u, b, Cw + sc(a,u)$);
20 **Function** *DecSCCheck*(u, b, a, dis):
21 **if** $dis = sc(u,b)$ and $u \notin S[a]$ **then** $\phi_{sc}(u,b) \leftarrow \phi_{sc}(u,b) + 1$;
22 **else if** $dis < sc(u,b)$ **then** $Q.insert(u), S[u].insert(b)$;
23 **Function** *IncSCCheck*(u, b, dis):
24 **if** $dis = sc(u,b)$ **then**
25 $\phi_{sc}(u,b) \leftarrow \phi_{sc}(u,b) - 1$;
26 **if** $\phi_{sc}(u,b) < 1$ **then** $Q.insert(u), S[u].insert(b)$;
27 **Function** *DecLabelCheck*(a, b, dis):
28 **if** $Cw = L(a,b)$ **then** $\phi_d(a,b) \leftarrow \phi_d(a,b) + 1, X(a).FN[b] \leftarrow true$;
29 **else if** $Cw < L(a,b)$ **then** $C(a,b) \leftarrow -1, X(a).ifU \leftarrow true$;
30 **for** *each* $u \in X(a).A$ **do**
31 **if** $L(a,u) > Cw + L(b,u), r(u) > r(b)$ or
 $L(a,u) > Cw + L(u,b), r(u) < r(b)$ **then** $C(a,u) \leftarrow -1$;
32 **Function** *IncLabelCheck*(a, b, Cw):
33 **for** *each* $u \in X(a).A$ **do**
34 **if** $L(a,u) = Cw + L(b,u), r(u) > r(b)$ *or*
 $L(a,u) = Cw + L(u,b), r(u) < r(b)$ *or* $u = b$ **then**
35 $\phi_d(a,u) \leftarrow \phi_d(a,u) - 1$;
36 **if** $\phi_d(a,u) < 1$ **then** $C(a,u) \leftarrow 1, X(a).ifU \leftarrow true$;

9.3 Proofs

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

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

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

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

Proof of Lemma 3. The lower bound of the number of affected queries is determined by *Case D1*, which is $\phi(a,b)$. *Case D2* determines the upper bound of the number of affected queries, which may affect all queries on G in the worst case. □

Proof of Lemma 4. As per the definitions of $\Omega_{\bar{A}}$, $L(s)$ and $L(t)$ are untouched during the index maintenance. Therefore, $Q(s,t)$ is an unaffected query and could be computed by the "old" index L. □

Proof of Lemma 5. $d(s,t) \cdot v_{max}$ is the longest spatial distance that $sp(s,t)$ could traverse since v_{max} is the maximum speed limit on G. Therefore, using $d(s,t) \cdot v_{max}$ as the major axis of the ellipse with foci at s and t is the largest spatial area $sp(s,t)$ can traverse. □

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

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

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

Proof of Theorem 3. No updates within G_i means that there is no inner change source, while $d(b_1,b_2), \forall b_1, b_2 \in B_i$ remain unchanged after indicates that there is no outer change source triggering the same-partition query. Therefore, G_i is inner-unaffected. □

Proof of Theorem 6. $C(u,b) = 0$ means that $L(u,b)$ is unaffected. There are three cases for $L(u,b)$.

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

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

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

Proof of Theorem 7. Mixed shortcut maintenance algorithm leverages a set Q to record all affected tree nodes, which incurs $O(\Delta_{sc}^T \cdot \log(\Delta_{sc}^T))$. The access of affected shortcuts in $S[a]$ needs $O(\Delta_{sc} \cdot \log(w))$. The computation of all new shortcuts and shortcut update propagation both are $O(\Delta_{sc} \cdot w)$ while the label check during propagation needs $O(\Delta_{sc} \cdot h)$. Therefore, the time complexity is $O(\Delta_{sc}^T \cdot \log(\Delta_{sc}^T) + \Delta_{sc} \cdot (w + \log(w) + h))$. Mixed label maintenance algorithm checks all tree nodes with $O(\Delta^T)$ while only deal with the affected labels $O(\Delta_d)$ with the computation of new label $O(w)$, the update propagation $O(deg_{\mathcal{G}^-}^{max})$. Therefore, the complexity is $O(\Delta_d \cdot (w + deg_{\mathcal{G}^-}^{max}) + n)$. The space complexity consists of three parts: 2-hop labels and ancestors $O(n \cdot h)$, shortcut size $O(n \cdot w)$, and the downward neighbors $O(n \cdot deg_{\mathcal{G}^-}^{max})$. Therefore, the space complexity is $O(n \cdot (w + h + deg_{\mathcal{G}^-}^{max}))$. □

9.4 Additional Experimental Results

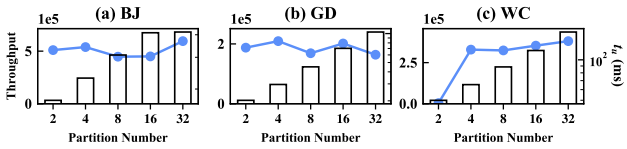

Figure 16: Effect of Region Number $\mathcal{K}$ of VPL

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