TraSS: Efficient Trajectory Similarity Search Based on key-value data stores –Details of Hausdorff, DTW and Range query

A THE ENLARGED ELEMENT

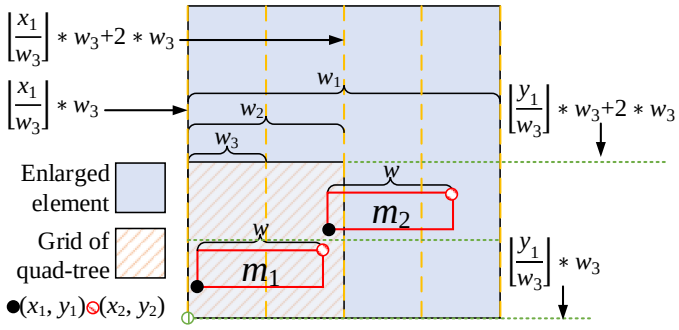


Figure 1: The Comfortable Resolution of The MBR.

Without losing generality, we normalize the entire space range to an interval of 0-1. That is, the width and height of the entire space are both normalized to 1. We use $((x_1, y_1), (x_2, y_2))$ to represent the lower left and upper right points of a MBR, and w and h are the width and height. s stands for the quadrant sequence of the most appropriate enlarged element for the MBR, $|s|$ denotes its length and implies the appropriate resolution. Then, the width and height of that enlarged element are both $2 * 0.5^{|s|}$.

Lemma 1. *The most appropriate quadrant sequence s of a MBR $((x_1, y_1), (x_2, y_2))$ has a length of*

$$|s| = l \text{ or } l + 1, \tag{1}$$

where, $l = \lfloor log_{0.5}(max\{x_2 - x_1, y_2 - y_1\}) \rfloor$.

Proof. We let $w = x_2 - x_1$ and $h = y_2 - y_1$. Without loss of generality, we presume $w \geq h$.
(1) The width of an enlarged element at l-resolution is

$$w_1 = 2 * 0.5^l = 2 * 0.5^{\lfloor log_{0.5}(w) \rfloor} \geq 2 * w. \tag{2}$$

The lower-left corner of the MBR is located in the lower-left grid of the enlarged element, and $w \leq w_1/2$, so that completely contained in that enlarged element, e.g., as shown in Figure 1, m_1 and m_2 have a width of w and locate in the orange grid. (2) The width of an enlarged element of $(l + 1)$-resolution is

$$w_2 = 2 * 0.5^{l+1} \geq 2 * 0.5^{log_{0.5}(w)+1} = w. \tag{3}$$

Thus, some MBRs with width w ($w \leq w_2$) can completely be covered in an enlarged element at $(l + 1)$-resolution, such as in Figure 1, the orange grid is also an enlarged element at $(l + 1)$-resolution, and the MBR m_1 can be completely contained by the orange grid.
(3) The width of an enlarged element of $(l + 2)$-resolution is

$$w_3 = 2 * 0.5^{l+2} < 2 * 0.5^{log_{0.5}(w)+1} = w. \tag{4}$$

Thus, any one enlarged element at $(l + 2)$-resolution can not contain the object with width w entirely.

Therefore, the appropriate resolution (also the length of the corresponding quadrant sequence) for the non-point spatial object with width w is l or $l + 1$. □

We cannot confirm the befitting resolution for the MBR only according to Lemma 1. As shown in Figure 1, m_1 and m_2 with the same width w ($w_3 < w \leq w_1/2$). However, they are contained by $(l + 1)$-resolution and l-resolution, respectively. We divide the enlarged element at l-resolution in x-axis into four equal parts according to gold dotted lines in Figure 1. Obviously,

objects which intersect more than two parts must be covered in l-resolution because any enlarged element at $(l+1)$-resolution contains only two parts. Therefore, we calculate the appropriate resolution for the object by the following inequality,

$$\lfloor \frac{x_1}{w_3} \rfloor * w_3 + 2 \cdot w_3 \geq x_2. \tag{5}$$

If Inequality (5) is "false", the object must be covered in l-resolution. At the same time, y-axis also satisfies Inequality (5). Thus, the object can be covered at $(l+1)$-resolution, only if the Inequality (5) of x-axis and y-axis are both "true". Intuitively, in Figure 1, the Inequality (5) of m_2 is "false", and the Inequality (5) of m_1 in x-axis and y-axis are both "true". Therefore, m_1 is covered in the enlarged element at $(l+1)$-resolution and m_2 is contained in l-resolution completely.

B OTHER SIMILARITY MEASURES

B.1 Hausdorff

B.1.1 Definition.

Definition 1. (*Hausdorff*).

$$D_H = \max\{\max_{i=1}^{n} \min_{t \in T} d(q_i, t), \max_{j=1}^{m} \min_{q \in Q} d(q, t_j)\}, \tag{6}$$

where $t \in T$ and $q \in Q$ is a point of T and Q, representatively.

B.1.2 Pruning strategies. It is easy to know that the Hausdorff distance satisfies Lemma 3. Because, $D_H \geq d(t, Q)$ and $D_H \geq d(q, T)$. All other Lemmas proposed in Section 5.3 are based on Lemma 3, so that these Lemmas can be directly used in the Hausdorff distance.

B.2 DTW

B.2.1 Definition.

Definition 2. (*DTW*).

$$D_D(Q^n, T^m) = \begin{cases} \sum_{k=1}^{m} d(q_1, t_k) & \text{if } n = 1 \\ \sum_{k=1}^{n} d(q_k, t_1) & \text{if } m = 1 \\ d(q_n, t_m) + \min\{D_D(Q^{n-1}, T^m), & \\ D_D(Q^n, T^{m-1}), D_D(Q^{n-1}, T^{m-1})\} & \end{cases}, \tag{7}$$

B.2.2 Pruning strategies. **Global Pruning.** Both Lemmas proposed in Section 5.3 can be directly used in the DTW distance. In addition, we can further prune index spaces the following Lemma:

Lemma 2. *if*

$$\sum_{bbox \in Q.B} d(index\_space, bbox) > \varepsilon,$$

then $D_D > \varepsilon$, where $d(index\_space, bbox)$ is the minimum distance of the index space and a bbox.

Proof. Because the trajectory T is completely covered by the index space $IndS$, so that the distance of each point of T to the bbox is greater than or equal to the distance of $IndS$ to the bbox. Thus, $D_D(Q^n, T^m) \geq D_D(Q^n.B, T^m.B) \geq D_D(Q^n.B, T^m.MBR) \geq D_D(Q^n.B, IndS) = \sum_{bbox \in Q.B} d(IndS, bbox)$. □

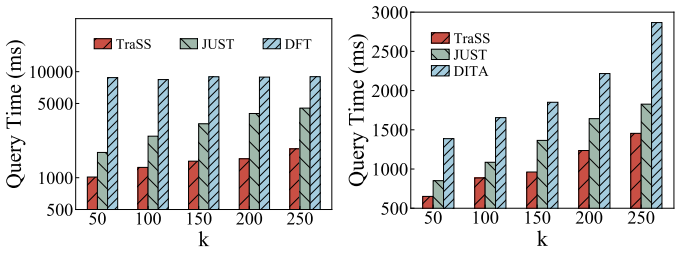
Local Filtering. Both Lemmas proposed in Section 5.4 can be directly used in the DTW distance. In addition, we can further prune index spaces the following Lemmas:

Lemma 3. *if $D_D(Q.B, T.B) > \varepsilon$, then $D_D > \varepsilon$, where $D_D(Q.B, T.B)$ is the DTW distance of the bboxes of Q and T.*

Proof. Analogy to the proof of Lemma 2. □

B.3 Efficiency.

DITA does not support the Hausdorff distance and *DFT* does not support the DTW. Figure 2 shows the efficiency of our framework. We can intuitively observe that in the Hausdorff and DTW metrics *TraSS* is more outstanding than others.



(a) Hausdorff (Tdrive). (b) DTW (Tdrive).

Figure 2: The Effect of Other Measures.

C RANGE QUERY

We use a suitable indexed space to represent the non-point object, and number the corresponding quadrant sequence and position code as an integer value, namely XZ+ value. Then, we can use the XZ+ value as the spatial key and extract the geometry representation of the object as the value. After that, we store the spatial key and value of the object into the database. Then, given a query window, we use the query processing algorithm to efficiently generate range scans of related spatial keys and extract corresponding non-point spatial objects.

The goal of query processing is to generate spatial key range scans selectively and accurately, automatically filtering out many unnecessary objects. This section focuses on the spatial range query because it is the most basic spatial query, and other queries, e.g., kNN query can be easily implemented by extending the spatial range query.

C.0.1 Generating Range Scan. Given a query window QW, we can efficiently find the objects that intersect with QW without accessing plenty of unnecessary data. We use a Breadth-First Search to generate spatial key range scans of the intersecting indexed spaces from resolution 1 to the maximum resolution r. We first check the spatial relationship between the query window and each enlarged element at 1-resolution. There are three spatial relationships:
(1) $contains$, if the enlarged element is completely contained in the query window, all indexed spaces with different resolutions covered by this enlarged element must be in the query window. Therefore the range scan contained in the enlarged element is from the corresponding XZ+ value $C(s, 1)$ to $C(s, 1) + N_{is}(|s|) - 1$, where s is the corresponding quadrant sequence of the enlarged element;
(2) $intersects$, if the query window intersects the enlarged element, the range scan is decided by the intersecting position codes, which are determined by the lower-left corner of the query window;
(3) $disjoint$, enlarged elements that do not intersect the query window are ignored.

After checking all the enlarged elements at 1-resolution, then we recursively check the spatial relationship ((1), (2) or (3)) between the elements in the remaining queue and the query window until there are no elements in the remaining queue. Then, we obtain final range scans and extract the objects whose spatial keys are covered by these range scans from the database. Finally, we use their actual boundaries to remove objects that do not intersect with this query window.

D INSTANTIATION

INSTANTIATION. We implement our framework on HBase. Figure 3 gives the architecture of $TraSS$. The *storing* component includes *calculating* XZ* index values, *extracting* DP-features, and *building puts*. Firstly, trajectories are indexed using appropriate index spaces. Then, we use Douglas-Peucker (DP) algorithm to pre-calculate DP features of trajectories to speed up query processing. After that, we convert trajectories to *puts*, and insert all *puts* into HBase table regions. When executing the *quering*, we first extract the DP features of a given trajectory. Then, we push down global pruning (*G-Pruning*) and local filtering (*L-Filtering*) into the coprocessor of HBase.

The *storing* component includes *calculating* XZ* index value, *extracting* DP-features, and *building pu*. Firstly, trajectories are indexed using appropriate index space and encoded using XZ* index value (cf. Section 4). Then, we use Douglas-Peucker (DP) algorithm to pre-calculate DP features of the trajectory to speed up query processing. After that, to convert the trajectory as a *put* of HBase, $TraSS$ combines the index value and the id of the trajectory as the *rowkey* and records the other information (e.g., DP-features) as columns. Then all *puts* are inserted into HBase table regions. The *rowkey* of $TraSS$ for storing is combined

Algorithm 1: Spatial range query: $query(QW)$

Input: A query window: QW.
Output: Range scans: $scans$.

```
foreach element ∈ RootElements.children do
    remaining.add(element)
l = 1; remaining.add(LevelTerminator);
while remaining ≠ ∅ do
    e = remaining.poll;
    if e = LevelTerminator then
        if remaining ≠ ∅ then
            l = l + 1;
            remaining.add(LevelTerminator);
    else
        if QW contains e then
            min = sequenceCode(e.x1, e.y1, l, 1);
            max = min + Nis(l) − 1;
            scans.add((min, max));
        else
            if QW intersects e then
                ps = positionCodes(QW, e);
                sc = sequenceCode(e.x1, e.y1, l);
                max = min − ps.min + ps.max − 1;
                scans.add((sc, ps));
                if l < g then
                    foreach subElement ∈ e.children do
                        remaining.add(subElement);
scans.merge;
return scans;
```

as follows,

$$rowkey = shards + index\ value + tid,$$

where *shards* is a hash number to decentralize trajectories, which can avoid data skew problem; *indexvalue* represents the spatial information of the trajectory; *tid* is the identifier of the trajectory.

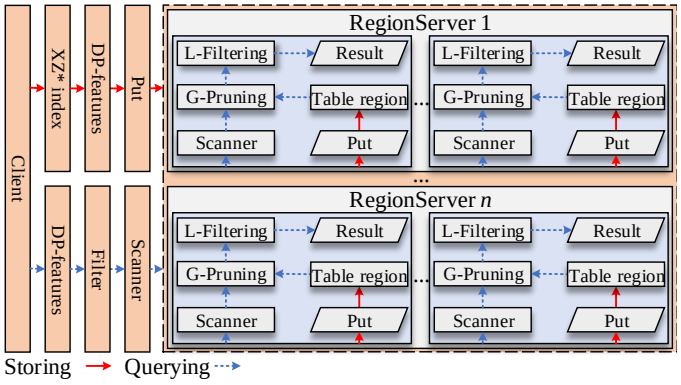


Figure 3: Architecture of *TraSS*.