K-Nearest Neighbor Temporal Aggregate Queries

Yu Sun [†1], Jianzhong Qi [†2], Yu Zheng [‡3], Rui Zhang [†4]

[†] *Department of Computing and Information Systems, University of Melbourne, Victoria, Australia*
{[1] sun.y, [2] jianzhong.qi, [4] rui.zhang}@unimelb.edu.au

[‡] *Microsoft Research, Beijing, P.R.China*
[3] yuzheng@microsoft.com

ABSTRACT

We study a new type of queries called the *k-nearest neighbor temporal aggregate* (kNNTA) query. Given a query point and a time interval, it returns the top-k locations that have the smallest weighted sums of (i) the spatial distance to the query point and (ii) a temporal aggregate on a certain attribute over the time interval. For example, *find a nearby club that has the largest number of people visiting in the last hour*. This type of queries has emerging applications in location-based social networks, location-based mobile advertising and social event recommendation. It is a great challenge to efficiently answer the query due to the highly dynamic nature and the large volume of the data and queries. To address this challenge, we propose an index named TAR-tree, which organizes locations by integrating the spatial and temporal aggregate information. We perform a detailed analysis on the cost of processing kNNTA queries using the TAR-tree. The analysis shows that the TAR-tree results in much fewer node accesses than alternatives. Furthermore, we propose two enhancements for the kNNTA query: (i) an algorithm suggesting the least amount of weights to be adjusted to explore different query results and (ii) a collective processing scheme to share index traversal among a batch of queries. We conduct extensive experiments using real-world data sets. The results validate the accuracy of the cost analysis and show that the TAR-tree outperforms alternatives by up to ten times in node accesses. The results also show that the weight adjustment algorithm and collective processing scheme outperform their baselines by significant margins.

1. INTRODUCTION

Location-based services (LBSs) have a large market and this market is growing rapidly. A well-known global market research company MarketsandMarkets forecasts in a recent report that the LBSs market will grow from $8.12 billion in 2014 to $39.87 billion in 2019. Location-based social networks (LBSNs) [31] have been a driving force for the growth of LBSs. Many emerging applications enable users to explore their neighborhood with rich social information in a highly customized fashion. For example, using the functionality *Places Nearby* (e.g., in Facebook or Foursquare), users may want to find nearby attractions that have the most visits recently or find a nearby club that is gathering the most people in the last hour; using the functionality *Explore* (e.g., in Flickr or Instagram), users may want to browse photos taken nearby and have the most *likes* lately.

These applications require ranking locations (or geotagged media contents) based on two criteria: (i) the spatial distance and (ii) a temporal aggregate on a certain attribute (e.g., the visits or likes). The spatial distance indicates the degree of closeness while the temporal aggregate reflects the social opinion in a certain period. These applications exhibit three key characteristics, which create a highly dynamic environment: (i) The visits or likes happen continuously, making the aggregate data grow rapidly. For instance, there were 3 million check-ins per day in Foursquare by May 2014. The number of the aforementioned requests is also very large. (ii) The time interval a user interested in is highly customized, which may vary from hours (e.g., for retrieving current events) to years (e.g., for long term analyses). (iii) The users may adjust their weighting on the two criteria widely to explore results of different preferences.

The skyline operator [6] can support multi-criteria decision problems. However, the skyline operator is computationally expensive even for static data and queries. The highly dynamic environment and the large volume of requests and objects generated in LBSNs make it prohibitive to use the skyline operator. Moreover, users are not given the flexibility in determining their preference over the two criteria. Following existing studies [9][15][22], we rank the locations using a weighted sum of the spatial distance and the temporal aggregate. We formulate the problem as the *k-nearest neighbor temporal aggregate* (kNNTA) query (formally defined in Section 3). Apart from the above applications in LBSNs, kNNTA queries are useful in many other applications in urban computing [32] where the spatial distance and a temporal aggregate are considered simultaneously, such as location-based mobile advertising and social event recommendation.

The kNNTA query requires quick response since users usually use the query to browse locations or geotagged media contents in the neighborhood. Due to the dynamic nature and the huge volume of the data and queries, having an efficient solution to this type of queries is challenging. Existing indexing structures cannot manage the locations effectively

based on both spatial closeness and temporal aggregate information simultaneously (detailed discussion in the related work, Section 2). To efficiently process the kNNTA query, we propose a novel index named the TAR-tree, in which the locations are organized by integrating the spatial and temporal aggregate information. We perform a detailed analysis on the cost of query processing using the TAR-tree. The analysis shows that the TAR-tree results in much fewer node accesses than alternatives that organize the locations based on only the spatial or the temporal aggregate information. The analysis can also be used as a cost model for query optimization. Furthermore, we propose two enhancements for the kNNTA query: (i) To help users explore results of different preferences, we propose an efficient algorithm suggesting the least amount of weights to be adjusted between the two criteria so that the query results will change. (ii) To handle large number of queries, we propose a collective processing scheme to share index traversal among a batch of queries. In summary, the main contributions of this paper are as follows.

- We propose a query called the k-nearest neighbor temporal aggregate (kNNTA) query to address emerging applications that requires ranking locations on both (i) the spatial distance and (ii) a temporal aggregate on a certain attribute.
- We propose a novel index named the TAR-tree to efficiently process the kNNTA query. We perform a detailed analysis on the cost of query processing using the TAR-tree, which shows that the TAR-tree results in much fewer node accesses than alternatives.
- We propose two enhancements for the kNNTA query: (i) an algorithm suggesting the least amount of weights to be adjusted to explore different query results and (ii) a collective processing scheme to share index traversal among a batch of queries.
- We conduct extensive experiments using real-world data sets. The results validate the accuracy of the cost analysis, and show that the TAR-tree outperforms alternatives by up to ten times in node accesses. The results also show that the weight adjustment algorithm and collective processing scheme outperform their baselines by significant margins.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formalizes the kNNTA query. Section 4 presents the TAR-tree. Section 5 discusses grouping strategies. Section 6 provides the analysis. Section 7 gives two enhancements. Section 8 reports the experiment results and Section 9 concludes the paper.

2. RELATED WORK

Queries. Previous spatial aggregate queries focus on the *range aggregate* [25], which returns the summarized information of POIs falling in a hyper rectangle (e.g., find the maximum or minimum weight among POIs intersecting the query rectangle). Temporal range aggregate queries [26] have also been studied, which add the temporal dimension to range aggregate queries (e.g., return the number of cars in the city center *during the last hour*). The kNNTA query differs from these queries in that (i) it returns the POIs rather than the aggregate value (e.g., the number of cars) and (ii) its aggregate is over the history of individual POIs (e.g., the check-in history) rather than spatial regions. Spatial keyword queries [9] retrieve the top-k objects such that their locations are close to the query point and their textual descriptions are relevant to the query keywords. The kNNTA query differs from spatial keyword queries in that instead of the keywords query time intervals are given, and a dynamic aggregate attribute (e.g., the count of check-ins) rather than the textual relevance is considered. Given two data sets P and Q (queries), an aggregate nearest neighbor (aNN) query [19] retrieves the points in P that have the smallest aggregate distances to the points in Q. The aNN query aggregates on the distances of a group of points, different from the kNNTA query which aggregates in the time dimension. Therefore, the algorithms for aNN queries cannot apply. Many other types of queries aggregating on different objects such as moving objects [10][16], data streams [29] or locations [13][20][21] are also studied. These queries are all different from the kNNTA query, and hence the algorithms for them cannot apply.

Indexes. Indexes such as aR-tree [17] and aP-tree [25] were proposed to process range aggregate queries. They cannot be adapted to process the kNNTA query because only one aggregate is maintained. The kNNTA query requires the temporal aggregate over various time intervals. Papadias et al. [26] proposed the aRB-tree to process temporal range aggregate queries. The aRB-tree combines the R-tree and B-tree, making each entry of the R-tree point to a B-tree which stores historical aggregates of the entry over each timestamp. To address the *distinct counting problem* in aRB-tree, i.e., an object will be counted multiple times if it remains in the query rectangle for more than one timestamp, Tao et al. [24] proposed the *sketch index* which is similar to the aRB-tree but with the B-tree storing historical counting sketches of the regions in its subtree. The aRB-tree and sketch index cannot be adapted to process the kNNTA query when the epochs are of varied lengths, since the B-tree cannot index time intervals. Even if the epochs are of equi-length, the aRB-tree and sketch index pay no attention to entry grouping strategies and group the entries based on only spatial extents, which, as will be shown in our analysis and experiments, is not effective for processing the kNNTA query. Sun et al. [23] divided the space into regular grid and proposed an adaptive multi-dimensional histogram (AMH) to answer temporal range aggregate queries. AMH cannot be adapted to answer the kNNTA query either, since the histogram buckets only maintain the aggregate and cannot retrieve individual POIs. Even if we use extremely fine granularity such that each cell in the grid only contains one POI, the buckets are grouped mainly by the aggregate dimension which, as will be shown, is also an ineffective strategy. Cong et al. [9] proposed the IR-tree for spatial keyword queries by integrating the R-tree and inverted indexes. Variants of the IR-tree, such as DIR-tree, group the R-tree entries by minimizing a weighted sum of the spatial closeness and text similarities, which is not optimal since it introduces another parameter, precludes existing optimization techniques for R-tree and makes it difficult to estimate the query processing cost. When designing the TAR-tree, our main focus is to develop a robust and effective grouping strategy. Many other spatial indexes [14][30] for nearest neighbor queries are also proposed. These indexes cannot be adapted to process the kNNTA query as they only focus on the spatial dimensions and are unable to tackle the temporal aggregate.

3. PROBLEM FORMULATION

3.1 Query Definition

The locations, which may have spatial extents, are hereafter termed as points-of-interest (POIs). The visits, likes, and so on are termed as *checked-ins*. A *k-nearest neighbor temporal aggregate* (kNNTA) query returns the top-k POIs based on a weighted sum of (i) the spatial distance to the query point and (ii) a temporal aggregate on the check-ins over a time interval. More precisely, we rank the POIs by a function f that computes the ranking score of a POI p as

$$f(p) = \alpha_0 d(p,q) + \alpha_1 (1 - g(p, \mathcal{I}_q)), \quad (1)$$

where $\alpha_i > 0$ (a constant) is the weight, $0 \leq d(p,q) \leq 1$ is the normalized Euclidean distance between p and the query point q, and $0 \leq g(p, \mathcal{I}_q) \leq 1$ is the normalized temporal aggregate of p over a query time interval $\mathcal{I}_q$. We use the weighted sum due to its simplicity and common usage in the literature [9][15][22], although the same result can be achieved by any monotonic function on the two criteria. We normalize the spatial distance $d(p,q)$ and temporal aggregate $g(p, \mathcal{I}_q)$ by dividing each by its *range* (i.e., maximum − minimum), so that the value is in the range $[0, 1]$. The normalization prevents one criterion from overpowering the other if it has a relatively large value. Without loss of generality, we let $\alpha_0 + \alpha_1 = 1$, since the ranking does not change if α_0 and α_1 is multiplied by a positive constant. The smaller the ranking score is, the higher p ranks and the better it suits the query.

The temporal aggregate can be *count*, *min*, *max*, *sum* or *average* (i.e., $\frac{sum}{count}$). In this paper, we focus on the aggregate that counts the number of check-ins at a POI, but the methods easily extend to other aggregates. In the rest of this paper, we omit "temporal" when the context is clear and simply use "aggregate" to refer to the "temporal aggregate". Let t_0 be the starting of the application and t_c be the current time. We discretize the time axis in to *epochs*. Each epoch may be a second, an hour or of varied lengths (e.g., one hour, two hours, four hours, eight hours and so on) depending on the application. The aggregate $g(p, \mathcal{I}_q)$ is computed by adding up the number of check-ins at p whose epoch intersects $\mathcal{I}_q$. We summarize the definition of the kNNTA query as follows.

DEFINITION 1. **K-Nearest Neighbor Temporal Aggregate (kNNTA) Query.** *Given a query point and a time interval, a k-nearest neighbor temporal aggregate query returns a set $\mathcal{R}$ of k POIs with the minimum ranking scores computed by the ranking function f given by Equation 1, i.e., $\forall p \in \mathcal{R}$ and $p' \in \mathcal{P} \setminus \mathcal{R}$, $f(p) \leq f(p')$.*

3.2 A Straightforward Approach

Figure 1 gives an example. The circles are the POIs. Table 1 presents the number of check-ins that each POI has in epochs $[t_0, t_1)$, $[t_1, t_2)$ and $[t_2, t_c]$, respectively. A kNNTA query is issued with a query point q denoted by the small square, a time interval $[t_0, t_c]$, $\alpha_0 = 0.3$ ($\alpha_1 = 0.7$) and $k = 1$. The ranking score of e is computed by $f(\mathsf{e}) = 0.3 \cdot \frac{2.24}{15.6} + (1 - 0.3) \cdot (1 - \frac{2}{12}) = 0.626$, where 2.24 is the Euclidean distance between e and q, 15.6 is the maximum distance between any two points in the space, 2 is the aggregate at e over $[t_0, t_c]$ and 12 is the maximum aggregate among all POIs. We obtain f as the query result, whose spatial distance to q equals 3 and aggregate equals 3+5+4 = 12. The ranking score of f is $0.3 \cdot \frac{3}{15.6} + (1-0.3) \cdot (1 - \frac{12}{12}) = 0.058$.

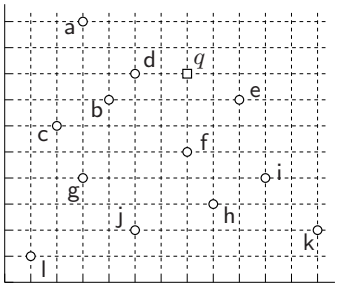


Figure 1: POIs and the query point

POI	$t_0 \rightarrow$	$t_1 \rightarrow$	$t_2 \rightarrow$
a	1	1	0
b	1	0	1
c	2	2	2
d	2	0	0
e	1	1	0
f	3	5	4
g	2	3	1
h	1	1	0
i	2	2	2
j	2	0	0
k	1	0	1
l	1	0	1

Table 1: Aggregate distribution

To handle the kNNTA query, a straightforward approach is sequential scan. Assume that the check-ins have already been counted within each epoch (as shown in Table 1). We first add up the number of check-ins in each epoch in the query time interval and obtain the aggregate for each POI. We then compute the ranking score of each POI, and return the top-k POIs. The time complexity is $\mathcal{O}(m'\mathcal{N} + \mathcal{N} \log m + k \log \mathcal{N})$, where m' is the number of epochs in the query time interval, $\mathcal{N}$ is the number of POIs, m is the number of epochs in $[t_0, t_c]$ (e.g., 3 in the above example) and k is the number of returned POIs. Both $\mathcal{N}$ and m' are very large in real social networks. For instance, $\mathcal{N} = 60,000,000$ in the LBSN Foursquare, and $m' = 8,760$ if the query time interval is one year and each epoch is one hour. The high cost makes this approach inapplicable in real applications.

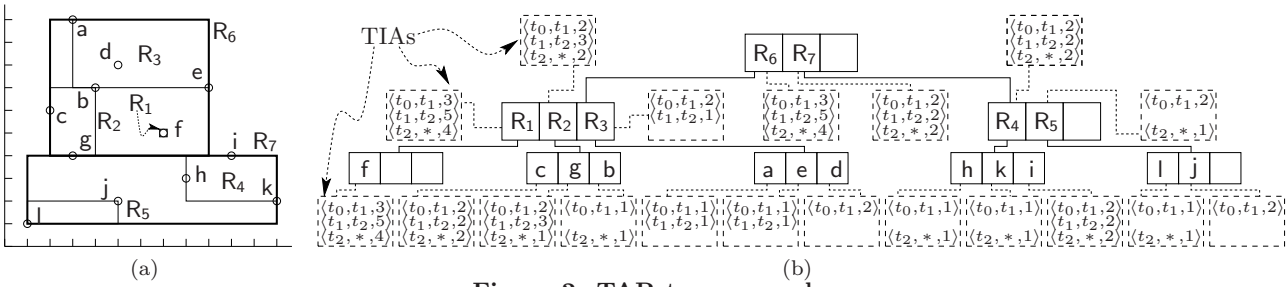
4. INDEX DESIGN

We design an index called the *temporal aggregate R-tree* (TAR-tree) to efficiently process the kNNTA query.

4.1 Index Structure

The TAR-tree is a variant of the R-tree.The algorithms for indexing the spatial extents of the POIs remain the same. A leaf entry is a minimum bounding rectangle (MBR) enclosing a POI. A leaf node contains a number of leaf entries. An entry in an internal node points to a child node (leaf node or internal node), and has an MBR enclosing the MBRs contained in the child node.

The difference between the TAR-tree and R-tree is that each entry of the TAR-tree also points to a temporal index. The temporal index stores the non-zero aggregate (at least one check-in) over each epoch, and keeps each record as a triple $\langle t_s, t_e, agg \rangle$, where t_s is the start time and t_e is the end time of the epoch, and agg is the aggregate value during the epoch. For brevity, we refer to the temporal index as the TIA (temporal index on the aggregate). The TIA of a leaf entry stores the aggregate of the POI it contains. The TIA of an internal entry stores the *largest* aggregate value of the TIAs in the child node for each epoch. For example, if two TIAs are in the child node and they store records $\{\langle t_0, t_1, 2\rangle, \langle t_1, t_2, 2\rangle, \langle t_2, *, 2\rangle\}$ and $\{\langle t_0, t_1, 2\rangle, \langle t_1, t_2, 3\rangle, \langle t_2, *, 1\rangle\}$, respectively, then the TIA of the internal entry pointing to this node stores the records $\{\langle t_0, t_1, \max\{2,2\}\rangle, \langle t_1, t_2, \max\{2,3\}\rangle, \langle t_2, *, \max\{2,1\}\rangle\}$. Any temporal index can be used to implement the TIA. We have used the disk-based multi-version B-tree [2] in our implementation as it has been proven to be



(a) (b)

Figure 2: TAR-tree example

asymptotically optimal.

In most applications, the aggregate update (i.e., inserting check-ins) is much more frequent than the spatial update (i.e., inserting POIs). We maintain the spatial and aggregate information in different components to enable quick digestion of new check-ins. Figure 2 presents an example of TAR-tree indexing the POIs shown in Figure 1. Figure 2(a) shows the MBRs of the entries. Figure 2(b) shows the index structure. The temporal records indexed by TIAs are enclosed by dashed lines. Empty lines in the TIAs mean that no records are stored for the epoch due to a zero aggregate. As we will see, the most important aspect for TAR-tree to efficiently process the kNNTA query is the strategy to group the entries. We will discuss the entry grouping strategy in Section 5.

4.2 Index Maintenance

We briefly discuss how to insert check-ins and POIs. Deletion is the same as R-tree and hence omitted.

Inserting Check-ins. When an epoch ends, we compute the aggregate of each POI by the check-ins (in this epoch), and then insert the non-zero aggregates in a batch fashion. Specifically, starting from the root node of TAR-tree, if an entry contains a POI whose aggregate is non-zero, we traverse the sub-tree rooted at the entry recursively. When reaching a leaf node, we store the non-zero aggregate into the POI's TIA, and return the largest aggregate in this node to the parent. Such an update procedure is efficient, since we only traverse part of the R-tree (which can be kept in main-memory) and insert only one record into the TIA.

Inserting POIs. When we insert a POI, the inserted path in TAR-tree is determined by the entry grouping strategy (which will be discussed in Section 5). For each entry in the inserted path, we update its MBR to include the POI, and update its TIA if in an epoch the aggregate of the POI is larger. If the insertion causes some POIs to be reinserted, we first remove these POIs from the TAR-tree, update the MBRs and TIAs in the inserted path, and then insert these POIs as described above. If the insertion causes some node to split, we redistribute the entries in the node by the entry grouping strategy.

4.3 Query Processing

We use the *best-first* search (BFS) [12] for query processing, which works as follows: (i) the entries in the root node are first inserted into a priority queue, in which the priority is determined by the entry's ranking score (detailed in the next paragraph), and then (ii) the front entry of the queue is ejected. If the entry is a leaf entry, the POI it contains is added to the result list; otherwise, each of its child entries is inserted into the queue. (iii) Step (ii) is repeated until k POIs are obtained.

The ranking score of an entry e is the weighted sum of the spatial distance from the query point to the MBR of e and the aggregate computed by the TIA of e. Given a query time interval $\mathcal{I}_q$, the TIA returns the records whose time interval $[t_s, t_e]$ is contained in $\mathcal{I}_q$. We obtain the aggregate over $\mathcal{I}_q$ by adding up the *agg* field of each returned record.

According to [12], the BFS produces correct query results as long as the entry's priority is computed by a *consistent* function. For the TAR-tree, the consistence can be expressed as: if e_c is an entry in the node pointed by entry e, then $f(e) \leq f(e_c)$. We prove the consistency of the ranking function f as follows.

PROPERTY 1. *Given any query point q and query time interval $\mathcal{I}_q$, we have $f(e) \leq f(e_c)$, where e_c is a child entry of entry e in the TAR-tree.*

PROOF. We have $f(e) = \alpha_0 d(e,q) + \alpha_1(1 - g(e, \mathcal{I}_q))$. Due to the TAR-tree design, it follows that $d(e,q) \leq \min_{e_c \in e} d(e_c, q)$ and $g(e, \mathcal{I}_q) \geq \max_{e_c \in e} g(e_c, \mathcal{I}_q)$. Therefore,

$$f(e) \leq \alpha_0 \min_{e_c \in e} d(e_c, q) + \alpha_1(1 - \max_{e_c \in e} g(e_c, \mathcal{I}_q))$$
$$\leq \alpha_0 d(e_c, q) + \alpha_1(1 - g(e_c, \mathcal{I}_q)) = f(e_c) \ \ \forall e_c \in e,$$

i.e., $f(e) \leq f(e_c)$. □

5. ENTRY GROUPING STRATEGIES

We now discuss the strategies for grouping the TAR-tree entries. As proved above, the BFS will provide the correct query results on the TAR-tree no matter which grouping strategy is used. The BFS has been proven to be optimal per TAR-tree instance in that only the TAR-tree nodes that intersect the *search region* will be accessed by the BFS [4]. However, different entry grouping strategies may result in different TAR-tree instances and hence vastly different number of node accesses. The performance of the BFS on the TAR-tree is roughly proportional to the number of accessed nodes, since similar operations are performed on each accessed node and the TAR-tree is most likely disk resident due to its large size as we discussed in Section 4.1. Therefore, we aim at minimizing the *node extents* in the TAR-tree so that fewer nodes are accessed by the BFS.

5.1 Two Straightforward Strategies

Since the TAR-tree is a variant of the R-tree, one straightforward strategy is to group the entries based on the spatial extents as R-tree does. Here we briefly review the grouping method of R*-tree [3]. When inserting a POI, we choose the entry that has the least overlap with other entries after containing the POI, if the entry points to a leaf node. If the entry points to an internal node, we choose the one that has the least area enlargement after including the POI. When a

node incurs overflow and this is the first time overflow happens in this level, we reinsert several entries of the node. When a node splits, we first choose a split axis, along which the sum of all possible new MBR margins is minimized. We then redistribute the entries (along the chosen split axis) such that the two new nodes have the minimum overlap.

Another straightforward strategy is to group the entries that have similar aggregate distributions. The similarity or distance between two aggregate distributions can be measured by the Manhattan distance (or Earth mover's distance and the like). For example, in Table 1, the distance between the TIA of c and TIA of g equals $0+1+1=2$, while the distance between the TIA of c and TIA of l equals $1+2+1=4$. When a POI is added, we insert the POI into the node that has the smallest distance to it. When a node splits, we redistribute the entries such that the distance between the two new nodes is maximized.

5.2 Integral 3D Strategy

As our analysis and experiments will show, the above two entry grouping strategies are not effective. We propose to group the entries by integrating the spatial and aggregate information to minimize the node extents. Specifically, we group the entries as 3-dimensional bounding boxes, in which two are the spatial dimensions and the third is a dimension capturing the aggregate information. As the aggregate information is distributed as aggregate values in many epochs. Here the trick lies in how to sufficiently represent the aggregate information as a single value (i.e., the coordinate of the third dimension). We have designed the third dimension as the following value

$$\widehat{\lambda}_p = \frac{1}{m}\sum_{i=1}^{m} v_i,$$

where m is the number of epochs in $[t_0, t_c]$ and v_i is the aggregate value in the i^{th} epoch (and as usual the bounding box of an internal entry encloses the bounding boxes of its child entries). This value is an estimate of the expected number of check-ins at the POI p contained by the leaf entry in an epoch (because we can model the number of check-ins at a POI in an epoch using the Poisson distribution). If two entries have similar such values, they may also have similar aggregates over the query time interval. It can significantly reduce the node extents if we group the entries having both similar spatial distances to the query point and similar aggregates over the query time interval.

Since the two types of information are of very different nature and do not have a unified domain range, when using this strategy, we normalize the spatial and aggregate dimensions by the ranges of their domains, respectively. In particular, to align with the ranking function, the normalized coordinate z_p of the third dimension for a leaf entry equals $z_p = 1 - \frac{\widehat{\lambda}_p}{\max_p \widehat{\lambda}_p}$. Note that only when we group the entries they are treated as 3-dimensional bounding boxes. When processing the kNNTA query, the spatial extents of the entry are obtained from the MBR and the aggregate from the TIA.

6. COST ANALYSIS AND COMPARISON OF GROUPING STRATEGIES

In this section, we analyze the query processing cost using the TAR-tree (with our proposed integral 3D entry grouping strategy). Through the cost analysis, we show that the TAR-tree results in much fewer node accesses than alternatives that use the other two grouping strategies. The analysis can also be used as a cost model for query optimization purposes. As mentioned before, we measure the cost by the number of node accesses. In the BFS, the accessed nodes are those intersecting the query search region, which is in turn determined by the data distribution. Therefore, we first analyze the distribution of the aggregate data in Section 6.1, and then estimate the search region and the number of node accesses in Sections 6.2 and 6.3, respectively. We compare the three entry grouping strategies in Section 6.4.

Table 2: Powerlaw fitting

Data	n	$\hat{\beta}$	$\hat{x}_{min}$	p-value
NYC	72,273	3.20	31	0.68
LA	45,591	3.07	16	0.18
GW	1,280,969	2.82	85	0.29
GS	182,968	2.19	59	0.21

6.1 Distribution of the Aggregate Data

Like many other types of data in real life [8], we observe that the aggregate value (i.e., the number of POIs having a certain aggregate value) follows the power-law distribution very well. Let the discrete random variable X be the count aggregate over a certain time interval, among the aggregates of all POIs, the probability that X has an observed value x is computed by

$$p(x) = \Pr(X = x) = Cx^{-\beta},$$

where C is a normalization constant. The power-law indicates that a small number of the POIs having a large proportion of the check-ins (roughly 80% of the check-ins are at 20% of the POIs). We test the power-law hypothesis on four real LBSN data sets (detailed at the beginning of Section 8) with the method in [8]. We list in Table 2 the results from the fitting of a power-law to each of the data sets, where n is the number of the tested POIs, $\hat{\beta}$ is the estimated scaling parameter, $\hat{x}_{min}$ is the estimated lower-bound to the power-law behavior and p-value is the goodness-of-fit indicator. It is suggested in [8] that the power-law hypothesis is ruled out if p-value is less than or equal to 0.1. Since the p-values of the four data sets are all clearly larger than 0.1, we argue that they all follow the power-law very well.

6.2 Estimation of the Query Search Region

Similar to the k-nearest neighbor query, the search region of the kNNTA query is determined by the ranking score of the k^{th} POI, which is denoted by $f(p_k)$. For ease of exposition, we describe the ranking score and search region in a normalized 3-dimensional unit cube, where two are the spatial dimensions and the third is the aggregate dimension. Figure 3 illustrates the ranking score with the query example in Section 3.2. The line segment $\overline{q\mathsf{g}'}$ represents the normalized spatial distance and $\overline{\mathsf{g}\mathsf{g}'}$ represents the normalized aggregate of g. The ranking score of g equals $\alpha_0|\overline{q\mathsf{g}'}| + \alpha_1|\overline{\mathsf{g}\mathsf{g}'}|$.

In the 3-dimensional unit cube, the query search region is of a cone shape. Its height and base radius, denoted by h_l and r_0, are computed by

$$r_0 = \frac{f(p_k)}{\alpha_0} \quad \text{and} \quad h_l = \frac{f(p_k)}{\alpha_1},$$

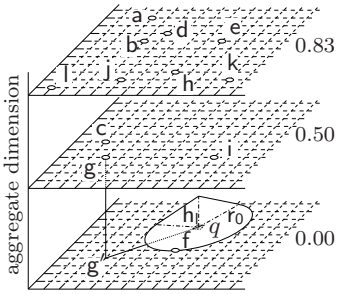


Figure 3: Cost analysis example

respectively. For example, in Figure 3, the cone illustrates the search region. Recall that in the query example we have $\alpha_0 = 0.3$, $\alpha_1 = 0.7$ and $f(p_k) = 0.058$, which implies that $r_0 = 0.192$ and $h_l = 0.082$. By definition, k POIs are in the search region. For instance, in the above example $k = 1$ and only f is in the search region. If $k = 2$, the search region will expand until it reaches a second POI. We use this property to estimate the size of the search region.

We observe that in the 3-dimensional unit cube, the POIs are only on a few *layers* at a specific height. Moreover, the number of such layers is countable. This is because the aggregate values (before normalization) are integers representing the number of check-ins. For example, in Figure 3, the POIs are only on three layers: a, b, d and so on have an aggregate value 2, and thus are on the layer at height $1 - \frac{2}{12} = 0.83$; c, g and i are on the layer at height $1 - \frac{6}{12} = 0.5$; and f and the query point q are on the layer at height 0. For simplicity, we denote each layer by the aggregate value x. By the power-law distribution, the probability $p(x)$ that a POI has an aggregate value x is computed by

$$p(x) = \frac{x^{-\beta}}{\zeta(\beta, x_{min})},$$

where

$$\zeta(\beta, x_{min}) = \sum_{i=0}^{\infty} (i + x_{min})^{-\beta}$$

is the Hurwitz zeta function [8]. The expected number of POIs on layer x, which is denoted by $\mathcal{N}(x)$, is computed by

$$\mathcal{N}(x) = \mathcal{N} \cdot p(x),$$

where $\mathcal{N}$ is the total number of POIs. Let the horizontal cross-section of the search region cut by layer x be $\mathcal{D}(q, r_x)$. The radius r_x of $\mathcal{D}(q, r_x)$ is computed by

$$r_x = \frac{h_l - h_x}{h_l} \cdot r_0,$$

where h_x is the height of layer x. Assume that the POIs are uniformly distributed on each layer. We can estimate the expected number of POIs in $\mathcal{D}(q, r_x)$ by $\mathcal{N}(x) \cdot \pi r_x^2$. However, the *boundary effects* cannot be neglected. Boundary effects represent the problem that some parts of the search region lie out of the 3-dimensional unit cube (e.g., when $k = 2$ in the above example). Taking the boundary effects into account, according to [4], the expected number of POIs bounded by $\mathcal{D}(q, r_x)$ is computed by

$$\mathcal{N}(x) \cdot E[S_{\mathcal{D}(q,r_x) \cap U_x}],$$

where $E[S_{\mathcal{D}(q,r_x) \cap U_x}]$ is the expected area that $\mathcal{D}(q, r_x)$ intersects layer x. Assuming that the query point is uniformly distributed, according to [27], we can approximate

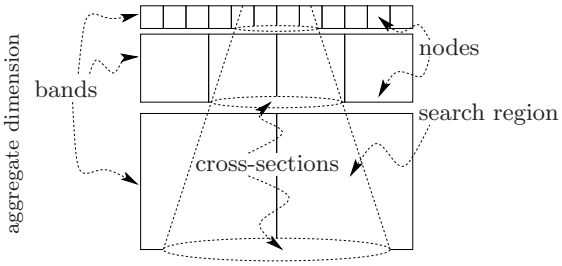


Figure 4: Node accesses estimation example

$E[S_{\mathcal{D}(q,r_x) \cap U_x}]$ by

$$\begin{cases} \left(\sqrt{\pi} \cdot r_x - \frac{\pi r_x^2}{4}\right)^2, & \sqrt{\pi} \cdot r_x < 2 \\ 1, & otherwise. \end{cases}$$

Adding up the number of POIs bounded by the cross-section on each layer, $f(p_k)$ can be estimated by solving the following equation:

$$k = \sum_{x=\Omega}^{\infty} \mathcal{N}(x) \cdot E[S_{\mathcal{D}(q,r_x) \cap U_x}],$$

where Ω is the minimum aggregate value.

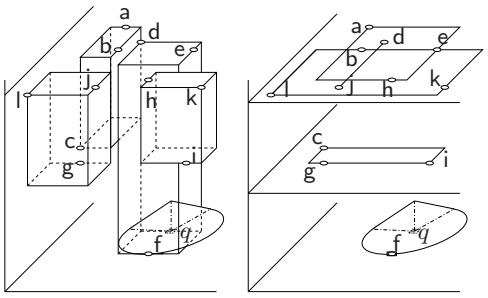
6.3 Estimation of the Number of Node Accesses

We estimate the number of node accesses by computing the number of nodes intersecting the search region. Without loss of generality, we only estimate the number of leaf nodes intersecting the search region since the number of internal nodes is much smaller than the number of leaf nodes. Also, the following analysis applies to internal nodes straightforwardly. The main challenge in the estimation is that the node extents are not uniform along the aggregate dimension due to the power-law distribution. The unit cube is divided into several *bands* along the aggregate dimension (computing the range of each band is detailed below). For example, in Figure 4, each square represents the extents of a node. The squares are small among higher layers and large among lower layers. The nodes of different extents form three bands. We first estimate the node extents and then the number of node accesses in each band.

Following existing cost analyses on the R-tree [12][5][27], we assume that the leaf nodes are of a cubic shape. We estimate the node extents by the extent along the aggregate dimension and the extents along the spatial dimensions. Starting from the top layer x, we proceed downward along the aggregate dimension. When we reach layer y, the node height equals $\Delta h = h_x - h_y$. Meanwhile, according to [5] the node extents along the spatial dimensions equal

$$S_y = \left(1 - \frac{1}{f}\right) \left(\min\left\{\frac{f}{\sum_{i=x}^{y} \mathcal{N}(i)}, 1\right\}\right)^{\frac{1}{2}},$$

where f is the fanout (the average number of entries in a node which typically equals 69% of the node capacity [28]). We obtain the node extents by solving the equation $S_y = \Delta h$ (or $S_y - \Delta h < \epsilon$). We refer to the space from layer x to layer y as a *band* (as shown in Figure 4). We then compute the expected number of nodes accesses in this band. The probability P_y that a node in a band intersects the search region is computed by the Minkowski sum [5] of the node extents S_y and the cross-section $\mathcal{D}(q, r_y)$ cut by the layer y (as illustrated in Figure 4). Taking the boundary effects



(a) (b)

Figure 5: Entry grouping examples

into account, according to [27], P_y can be estimated by

$$P_y = \begin{cases} \left(\frac{4L_y - (L_y + S_y)^2}{4(1 - S_y)}\right)^2, & L_y + S_y < 2, \\ 1, & otherwise, \end{cases}$$

where

$$L_y = \left[\sum_{i=0}^{2} \left(\binom{2}{i} \cdot S_y^{2-i} \cdot \frac{\sqrt{\pi^i}}{\Gamma(i/2+1)} \cdot r_y^i \right) \right]^{\frac{1}{2}}.$$

The expected number of node accesses NA_y in this band is thus computed by

$$NA_y = \frac{\sum_{i=x}^{y} \mathcal{N}(i)}{f} \cdot P_y,$$

where $\frac{\sum_{i=x}^{y} \mathcal{N}(i)}{f}$ is the number of nodes in this band. We then proceed with $x = y + 1$ and repeat the above steps until all layers are processed. The expected number of leaf node accesses, denoted by $NA(\alpha, k)$, equals the sum of the number of node accesses computed in each band, i.e.,

$$NA(\alpha, k) = \sum_{y} NA_y.$$

6.4 Comparison of Entry Grouping Strategies

Based on the above analysis (which is validated by our experiments), we qualitatively compare the three grouping strategies (discussed in Section 5).

If we use the spatial extents to group the entries, the nodes have weak pruning power in the aggregate dimension. The reason is that the nodes will be of a hyper-rectangle shape due to the power-law distribution. For example, in Figure 5(a) the hyper-rectangles represent the nodes. The lower part of such a node may intersect the search region with a high probability. The entries at the top of the unit cube are less likely to contain query results, however, they are accessed if the lower part of the node intersects the search region. The power-law indicates that 80% of the entries are at the top of the unit cube, and hence many nodes will be accessed unnecessarily.

If we use the aggregate distribution to group the entries, the nodes have weak pruning power in the spatial dimensions. This is because the nodes will cover a large space in the spatial dimensions since the spatial proximity is not considered. For example, Figure 5(b) shows the rectangles on each layer representing the nodes. We can see that they have large extents in the spatial dimensions and will be accessed with a high probability provided the height of the search region is greater than the layer containing the node.

The above drawbacks can be avoided when we use the integral 3D strategy. The node extents will follow a power-law-like distribution as shown in Figure 4. The nodes hence retain the pruning power of both spatial and aggregate dimensions. Therefore, the TAR-tree results in much fewer node accesses than alternatives that organizes entries using only the spatial proximity or the aggregate distribution.

7. ENHANCEMENTS FOR THE QUERY

In this section, we propose two enhancement techniques for the kNNTA query. In Section 7.1, we present an algorithm suggesting the least amount of weights to be adjusted that can cause the query results to be changed. In Section 7.2, we present a collective processing scheme to share the index traversal among a batch of queries.

7.1 Suggesting the Minimum Weight Adjustment

New users of the kNNTA query may have difficulty in setting the weights between the spatial distance and aggregate properly. They may adjust the weights to explore different results. It is discouraging if the results remain the same after the weights have been changed. We tackle this problem by suggesting the users the *minimum weight adjustment* that can change the current results (here, changing the results refers to changing the POIs in the kNNTA answer set).

A few existing studies proposed algorithms retaining the top-k results instead of changing the results. For example, Mouratidis et al. [15] proposed an algorithm that computes the *immutable regions* which is defined as the widest range of α_i that preserves the top-k results (assuming that the other weight α_{1-i} is kept constant). Soliman et al. [22] studied finding the maximal hypersphere centered at the weight vector $[\alpha_0, \alpha_1]^T$ such that each vector in the hypersphere preserves (including the order) the top-k results. These algorithms do not apply since they cannot compute the weight adjustment to change the top-k results.

To solve this problem, we first rewrite the ranking function $f(p)$. Let the POIs be ranked in a list. We denote the i^{th} ranked POI by p_i and rewrite the ranking function of p_i by $f(p_i) = \alpha_0 s_{i,0} + \alpha_1 s_{i,1}$, where $s_{i,0} = d(p_i, q)$ and $s_{i,1} = 1 - g(p_i, \mathcal{I}_q)$. For simplicity, we focus on the adjustment of α_0 (since $\alpha_1 = 1 - \alpha_0$). Given a top-k POI p_i ($i \leq k$) and a lower ranked POI p_j ($j > k$), where $f(p_i) < f(p_j)$, we obtain a value range of α_0 such that for any α_0' in the range, a ranking function $f'(p)$ defined by α_0' satisfies $f'(p_i) > f'(p_j)$. For example, in the ranking list in Table 3, we have $\alpha_0 = \alpha_1 = 0.5$ and $k = 2$. To let $f'(p_1) > f'(p_3)$, we need $\alpha_0' > \frac{5}{6}$. To let $f'(p_1) > f'(p_6)$, we need $\alpha_0' < \frac{1}{8}$. We refer to the boundary of the range as the *weight adjustment*, denoted by $\gamma_{i,j}$. Let $\delta_{i,j,t} = s_{i,t} - s_{j,t}$ for $t = 0, 1$. When $\delta_{i,j,0} \cdot \delta_{i,j,1} < 0$, $\gamma_{i,j}$ is computed by

$$\gamma_{i,j} = \frac{\delta_{i,j,1}}{\delta_{i,j,1} - \delta_{i,j,0}}.$$

When $\delta_{i,j,0} \cdot \delta_{i,j,1} \geq 0$, we cannot achieve $f'(p_i) > f'(p_j)$ since p_i *dominates* p_j (i.e., $s_{i,t} < s_{j,t}$ for $t = 0, 1$). The minimum weight adjustment (MWA) is the weight adjustments that are nearest to the current weight, i.e., the $\max\{\gamma_{i,j}\}$ or $\min\{\gamma_{i,j}\}$ when $\gamma_{i,j}$ is less or greater than the current weight. For example, in the ranking list in Table 3, to let $f'(p_1)$ be greater than $f'(p_3)$, $f'(p_5)$, $f'(p_6)$, we need $\alpha_0' > \frac{5}{6}$, $\alpha_0' > \frac{20}{29}$, $\alpha_0' < \frac{1}{8}$, and to let $f'(p_2)$ be greater than $f'(p_4)$,

Table 3: Ranking list

POI	$s_{i,0}$	$s_{i,1}$	POI	$s_{i,0}$	$s_{i,1}$
p_1	0.25	0.10	p_4	0.35	0.25
p_2	0.10	0.30	p_5	0.025	0.60
p_3	0.20	0.35	p_6	0.60	0.05

$f'(p_5)$, $f'(p_6)$, we need $\alpha'_0 < \frac{1}{6}$, $\alpha'_0 > \frac{4}{5}$, $\alpha'_0 < \frac{1}{3}$, respectively. The MWA of α_0 is either $\alpha'_0 < \frac{1}{3}$ or $\alpha'_0 > \frac{20}{29}$, since $\frac{1}{3}$ and $\frac{20}{29}$ and are nearest to the current weight 0.5 when the weight adjustment is less and greater than 0.5, respectively. More precisely, the MWA for α_0 comprises two values Γ_l and Γ_u that are computed by:

$$\begin{cases} \Gamma_l = \max\{\gamma_{i,j}\} \text{ for } \delta_{i,j,0} < 0, i \leq k, j > k, \\ \Gamma_u = \min\{\gamma_{i,j}\} \text{ for } \delta_{i,j,0} > 0, i \leq k, j > k. \end{cases}$$

The MWA will change exactly one of the top-k POIs and keeps the other top-k POIs (the order within top k may change). For example, if we change α_0 to 0.75 in the above example, the new top-2 POIs will be the current p_2 and p_5.

A straightforward way to compute the MWA on the TAR-tree is as follows: After finding the top-k POIs, for each of the top-k POIs p, we continue the BFS until the queue is empty. If the ejected entry e is dominated by p, we continue. Otherwise, we compute and update the (tentative) MWA if e is a leaf entry, or continue the BFS if e is an internal entry. This approach may incur significant cost since it enumerates each of the top-k POIs and has a very weak pruning power on the lower ranked POIs (by checking the dominance).

To overcome this drawback, we propose an approach that makes use of the skyline queries. We observe that: when $\delta_{i,j,0} > 0$ and $\delta_{i,j,0} < 0$, the weight adjustment computed from an entry gives an upper and lower bound on the weight adjustments computed from the child entries, respectively. The reason is, when $\delta_{i,j,0} < 0$ and $\delta_{i,j,0} > 0$,

$$\gamma_{i,j} = \frac{\delta_{i,j,1}}{\delta_{i,j,1} - \delta_{i,j,0}} = \frac{1}{1 - \frac{\delta_{i,j,0}}{\delta_{i,j,1}}} = \frac{1}{1 - \frac{s_{i,0} - s_{j,0}}{s_{i,1} - s_{j,1}}}$$

increases and decreases with the decrease of $s_{j,1}$ or $s_{j,0}$, respectively. Therefore, to compute the MWA, we only need to consider the weight adjustments when interchange the POIs on (i) the skyline of the lower ranked POIs and (ii) the skyline of the top-k POIs with the dominating condition reversed (i.e., p_i dominates p_j if $s_{i,t} > s_{j,t}$ for $t = 0, 1$). Therefore, after finding the top-k POIs, we propose to (i) first compute the skyline of the top-k POIs with the dominating condition reversed, and then (ii) compute the skyline of the lower ranked POIs and (iii) obtain the MWA by the weight adjustments interchanging the POIs on the two skylines. Note that although the proposed TAR-tree is designed for the kNNTA query, it also enables efficient answering of the skyline query, since many skyline algorithms are based on the R-tree (e.g., [18]). It is not difficult to extend the algorithm to compute the weight adjustment that leads to multiple top-k POIs being changed.

7.2 Collective Query Processing

To achieve high scalability when processing multiple kNNTA queries simultaneously, we propose to process kNNTA queries in a batch fashion.

Let c be the number of queries in a batch. We use c priority queues for the BFS of the c queries. In the BFS, we access a node when the front entry is an internal entry. Some front entries in the c queues may be the same (pointing to the same node). For example, if $c = 5$, after we insert a root node containing two entries R_1 and R_2, the front entries of the c queues may be R_1, R_1, R_2, R_2 and R_1. To reduce the number of node accesses, we process the c queues greedily, i.e., the queues containing the most frequent front entry are processed first, which makes the accessed node be shared by the most queries. For instance, in the above example, the node R_1 will be retrieved and the three queues having R_1 as the front entry will be processed first. To further share the aggregate computation on the TIAs in the accessed node, we group the queries together if they have the same query time interval (i.e., the same start time and length) and process the queries as a batch. Such grouping method is effective because in real applications users are usually given only a few options for the query time interval (e.g., one day or one week from now) by default.

Table 4: Data Set

Name	Time	Locations	Check-ins
NYC	05/2008-06/2011	72,626	237,784
LA	02/2009-07/2011	45,591	127,924
GW	02/2009-10/2010	1,280,969	6,442,803
GS	01/2011-07/2011	182,968	1,385,223

8. EXPERIMENTS

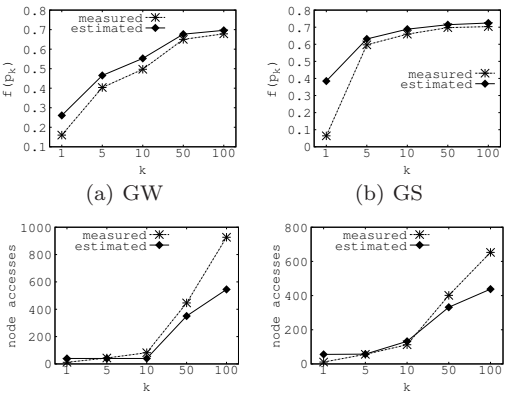
In this section, we empirically evaluate the cost analysis, the TAR-tree and two enhancements for the kNNTA query.

Experiments Setup. We use four real-world data sets: NYC, LA, GW and GS. NYC and LA [1] are two LBSNs for the New York City and Los Angeles, respectively (generated from Foursquare tips), GW [7] is the LBSN Gowalla and GS [11] is the LBSN Foursquare (generated from check-ins posted on Twitter). The details of the data sets are listed in Table 4. We implement the R-tree with the R*-tree [3] and the TIA with the Multi-version B-tree. Given the vast memory capacity of modern computers, the R-tree is kept in memory and each TIA is assigned a maximum of 10 buffer slots. To simulate real scenarios, unless otherwise specified, the R-tree node size is set to 1024 bytes (and hence the node capacities are 50 and 36 for 2- and 3-dimensional entries respectively), the epoch length is set to 7 days, and a location needs to have 15, 10, 100 and 50 check-ins for the four data sets respectively to be treated as an effective public POI and indexed. For each data set, we generate 1,000 queries with the query point uniformly sampled from the data set and the query time interval uniformly sampled from $2^0, 2^1, \ldots, 2^9$ days. By default $k = 10$ and $\alpha_0 = 0.3$.

The experiments are conducted on a 64-bit Windows desktop computer with a 3.40GHz Intel(R) Core(TM) i7-2600 CPU and 16GB RAM. All algorithms are implemented in Java. For all sets of experiments (except the validation of the cost analysis), we measure the CPU time and number of node accesses. All presented results are averaged over the 1,000 queries. Due to the space limitation, we only present the results of GW and GS. The results of NYC and LA are consistent with those of GW and GS, and hence are omitted.

8.1 Validation of the Cost Analysis

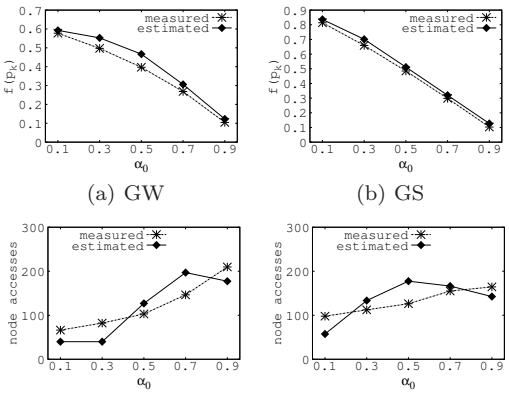
In this set of experiments, we evaluate the cost analysis by comparing the **estimated** $f(p_k)$ and number of leaf node accesses with the **measured** ones.



(a) GW (b) GS

(c) GW (d) GS

Figure 6: Cost analysis validation by varying k



(a) GW (b) GS

(c) GW (d) GS

Figure 7: Cost analysis validation by varying α_0

Varying k. We first evaluate the analysis by varying k from 1 to 100. The results are plotted in Figure 6. Figures 6(a) and 6(b) show the comparison of the estimated and measured $f(p_k)$. We can see that $f(p_k)$ increases with the increase of k as expected. The estimates are very close to the actual values when $k \geq 5$. The estimate is less accurate when $k < 5$, especially on GS, which is due to the large variance of $f(p_k)$ when $k < 5$ and only a few POIs have large aggregate values. Figures 6(c) and 6(d) present the comparison of the estimated and measured number of node accesses. We can see that with the increase of k the number of leaf node accesses increases and the estimates exhibit the same growing trend as the actual values. When $k \leq 50$, the estimates approximate the measured values very well. When $k > 50$, the estimation is slightly inaccurate due to that the number of POIs computed by the power-law is less accurate when x is close to $\hat{x}_{min}$ and it happens more frequently when k is larger. This problem can be addressed by collecting more data or introducing a more complex piece-wise fitting.

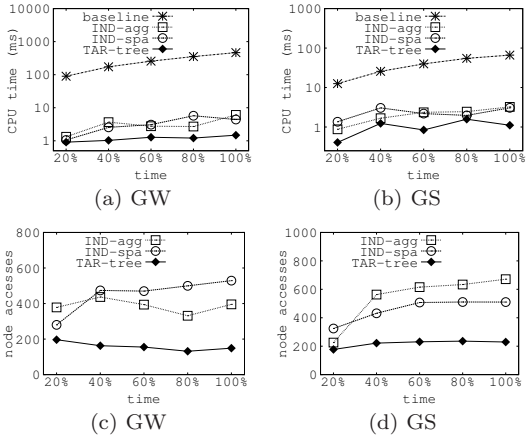
Varying α_0. Next, we evaluate the analysis by varying α_0 from 0.1 to 0.9. The results are plotted in Figure 7. Figures 7(a) and 7(b) depict the comparison of the estimated and measured $f(p_k)$. For all values of α_0, the estimates are almost identical to the actual values. Figures 7(c) and 7(d) illustrate the comparison of the estimated and measured number of node accesses. We can see that the number of node accesses increases moderately with the increase of α_0. The estimates fluctuate closely around the actual values. When α_0 is close to 0.9, the estimates show an opposite growing trend to the actual values. This is due to the same problem caused by the error of the power-law fitting when x is close to $\hat{x}_{min}$, and can also be addressed by the same approaches. Overall, the cost analysis is accurate and can strongly indicate the query processing cost.

8.2 Performance of the TAR-tree

In this set of experiments, we evaluate the performance of the TAR-tree. We compare the **TAR-tree** with the two alternatives (discussed in Section 5) using the spatial extents and the aggregate distribution to group the entries, respectively. We refer to the two alternatives as the **IND-spa** and **IND-agg**, respectively. We also compare the TAR-tree with the straightforward approach (scanning the aggregate values and POIs) to measure the query processing speed up, which is referred to as the **baseline**.

Effect of the LBSN Growing with Time. First, we evaluate the performance by simulating the growth of the LBSN with time. We take a snapshot on each data set at $20\%, 40\%, \ldots, 100\%$ of the time. The results are plotted in Figure 8. Figures 8(a) and 8(b) show the CPU time of different approaches. We can see that the TAR-tree runs several times faster than the IND-spa and IND-agg. The TAR-tree also runs greatly faster than the baseline. With the growth of the LBSN, the performance of the TAR-tree may slightly fluctuate (as shown in Figure 8(b)). This is due to that the TAR-tree does not adjust promptly to adapt to the current LBSN. To address the problem, we can reinsert part of the entries periodically or rebuild the TAR-tree when the performance degrades below some threshold.

Figures 8(c) and 8(d) present the number of node accesses against the growth of the LBSN. The TAR-tree consistently has the smallest number of node accesses and outperforms the IND-spa and IND-agg by a significant margin on both data sets. On GW, the TAR-sap incurs the largest number of node accesses, while on GS the IND-agg is the worst. This indicates that unlike the TAR-tree, the performance of IND-spa and IND-agg is unstable across different data sets. On GW, the number of node accesses in the TAR-tree marginally decreases with the growth of the LBSN. This implies that the TAR-tree performs better when there are sufficient aggregate information.

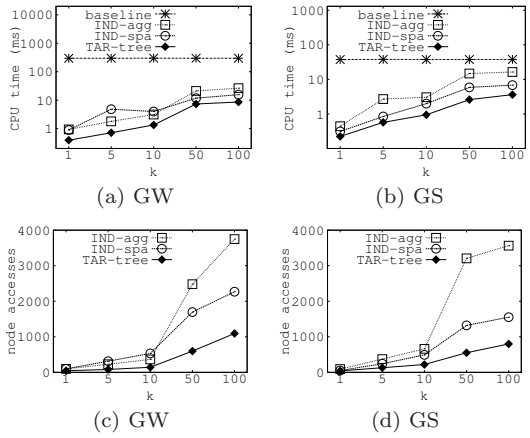


(a) GW (b) GS

(c) GW (d) GS

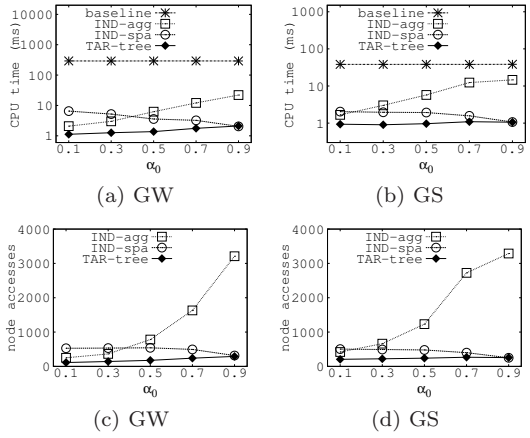
Figure 8: TAR-tree evaluation by simulating the growth of the LBSN

Effect of k. Next, we evaluate the performance the TAR-tree by varying k from 1 to 100. The results are presented in Figure 9. We can see that the TAR-tree constantly outper-



(a) GW (b) GS (c) GW (d) GS

Figure 9: TAR-tree evaluation by varying k

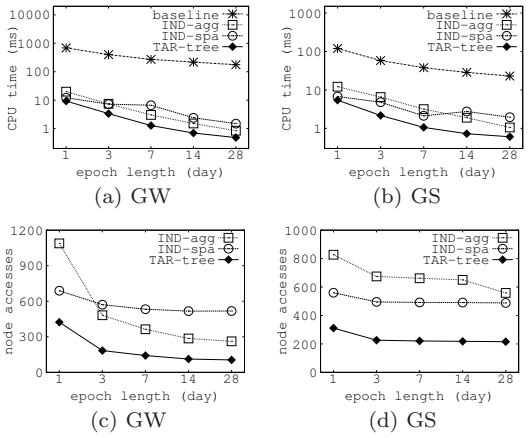


(a) GW (b) GS (c) GW (d) GS

Figure 10: TAR-tree evaluation by varying α_0

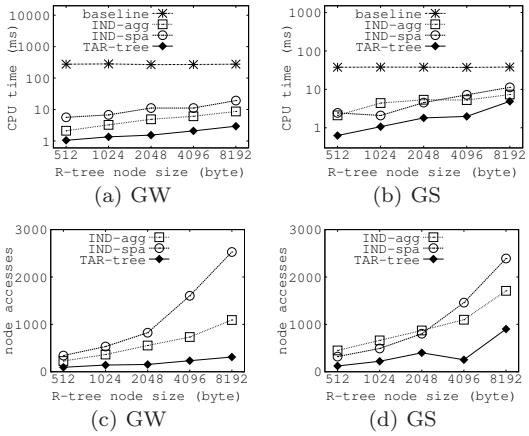
forms all the other approaches. With the increase of k, the CPU time and number of node accesses of all approaches increase. This, as indicated by the cost analysis, is because the search region expands with the increase of k and hence accesses more nodes. When $k \geq 50$, as Figure 9(b) shows, the performance of the IND-agg deteriorates rapidly and its CPU time is comparable to that of the baseline. Figures 9(c) and 9(d) show the number of node accesses. We can see that when $k > 10$, the IND-spa and IND-agg have a much larger growth rate than that of the TAR-tree. This also confirms the cost analysis (in Section 6.4) that the expanding of the search region has a larger impact on IND-spa and IND-agg.

Effect of α_0. Figure 10 plots the performance of different approaches when the value of α_0 is varied from 0.1 to 0.9. When α_0 approaches 1, the importance of the spatial distance increases in the kNNTA query. Figures 10(a) and 10(b) show the CPU time and we can see that when α_0 approaches 1, the performance of the IND-spa and IND-agg decreases and increases, respectively. This is because the IND-spa and IND-agg are optimized for the spatial and aggregate dimensions, respectively. The performance of the TAR-tree is almost unaffected by the changing of α_0 and the TAR-tree keeps running the fastest. Even when $\alpha_0 = 0.1$ and 0.9, for which the IND-agg and IND-spa are supposed to have a good advantage, the TAR-tree still performs no worse than the IND-agg and IND-spa, respectively. Figures 10(a) and 10(b) present similar results on the number of node accesses. We can see that when α_0 approaches 1, the number of node accesses in the IND-agg increases radically. This is because the height of the search region grows rapidly with the increase of α_0, and for the IND-agg, a node is accessed (with a high probability) as long as the layer containing the node is less than or equal to the height of the search region.



(a) GW (b) GS (c) GW (d) GS

Figure 11: TAR-tree evaluation by varying the epoch length

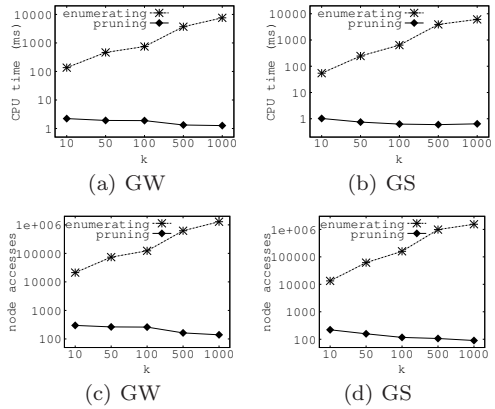


(a) GW (b) GS (c) GW (d) GS

Figure 12: TAR-tree evaluation by varying the R-tree node size

Effect of the Epoch Length. We now proceed to evaluate the performance by varying the parameters of the TAR-tree. First, we vary the epoch length from 1 to 28 days and present the results in Figure 11. Figures 11(a) and 11(b) show that the CPU time of all approaches (including the baseline) decreases with the increase of the epoch length. This is because fewer values are added up to obtain the aggregate. Figures 11(c) and 11(d) show the number of node accesses and we can see that the longer the epoch length is, the fewer node accesses the TAR-tree needs to process the query. The reason is that a longer epoch length strengthens the pruning power of the TAR-tree because the aggregate of a parent node is closer to the maximum aggregate computed from its child nodes. For all epoch lengths, the TAR-tree outperforms the other approaches both in the CPU time and number of node accesses.

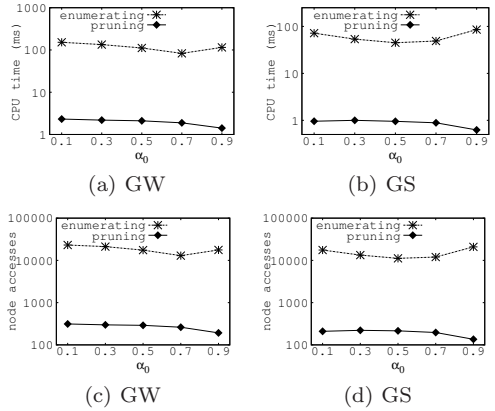
Effect of the R-tree Node Size. Next, we vary the R-tree node size from 512 to 8192 bytes and present the results in Figure 12. As shown in Figures 12(a) and 12(b), the CPU time of the TAR-tree increases almost linearly with the increase of the node size. This is because the number of entries in a node grows linearly as the node size grows and similar operations are performed on each entry. Fig-



(a) GW (b) GS

(c) GW (d) GS

Figure 13: Computing the MWA by varying k



(a) GW (b) GS

(c) GW (d) GS

Figure 14: Computing the MWA by varying α_0

ures 12(c) and 12(d) show that the number of node accesses of all approaches increases with the growth of the node size. The IND-spa has the largest growth rate and the TAR-tree has the smallest growth rate. The reason is that with the increase of the node size, the node represent a larger spatial region, and thus has a relatively weak pruning power in spatial extents. Under all settings, the TAR-tree consistently outperforms all the other approaches.

8.3 Performance of the Weight Adjustment Algorithm

In this set of experiments, we compare the performance of the proposed weight adjustment algorithm with the straightforward approach. We refer to the proposed algorithm and the straightforward approach as **pruning** and **enumerating**, respectively.

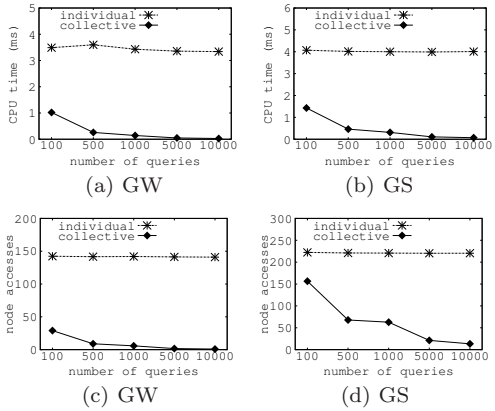
Varying k. We first evaluate the algorithm by varying k from 10 to 1000. We plot the results in Figure 13. From Figures 13(a) and 13(b), we can see that pruning runs orders of magnitude faster than enumerating. The performance of enumerating degrades rapidly as k grows. This is because each top-k result is enumerated and computed against the lower ranked POIs. The CPU time of the pruning algorithm decreases marginally with the increase of k. This is because computing the skyline of the lower ranked POIs takes much less time and pays off the time spent on computing the skyline of the top-k POIs. Figures 13(c) and 13(d) show consistent results on the number of node accesses except that the number of node accesses decreases marginally faster than the CPU time since it incurs no node accesses to compute the skyline of the top-k POIs.

Varying α_0. Next, we evaluate the algorithm by varying α_0 from 0.1 to 0.9. The results are presented in Figure 14. As Figures 14(a) and 14(b) show, the CPU time of enumerating first decreases and then increases as α_0 grows. Since checking the dominance is the only pruning technique used by this approach, it indicates that the pruning power of the technique is weakest when α_0 is around 0.1 or 0.9. The CPU time of the pruning algorithm has an opposite growing trend to enumerating. This indicates that it is more efficient to compute the skyline when the weight is skewed. Figures 14(c) and 14(d) present consistent results on the number of node accesses. In all settings, the proposed algorithm outperforms the baseline by a significant margin.

8.4 Performance of the Collective Processing Scheme

In the last set of experiments, we evaluate the collective processing scheme (**collective**) against the approach to processing the query individually (**individual**). To investigate the effect of memory buffering on processing the query individually, we assign no buffer to the TIAs.

Varying the Number of Queries. Figure 15 presents the CPU time and the node accesses as a function of the number of queries. From Figures 15(a), 15(b) and Figures 15(c), 15(d), we can see that for the collective processing scheme, the more queries are processed collectively, the shorter the average processing time is and the smaller number of node accesses we need, respectively. This is because more queries share the index traversal. We can also see that when processing the query individually, changing the number of queries has little effect on either the CPU time or the number of node accesses. The collective processing scheme constantly outperforms processing the query individually by a big margin.



(a) GW (b) GS

(c) GW (d) GS

Figure 15: Collective processing by varying the number of queries

Varying the Number of Query Types. Next, we evaluate the collective processing scheme by varying the number of query time intervals (i.e., query types) from 1 to 100. Figure 16 presents the results. Since the queries are grouped by the query time interval, the efficiency of the collective processing scheme will decline when the number of query time interval increases. As Figures 16(a) and 16(b) show, the efficiency of the collective processing scheme does not degrade too much when there are more than 10 types of queries. The collective processing scheme keeps running several times faster than processing the queries individually. Figures 16(c)

and 16(d) present consistent results on the number of node accesses. In all settings, the collective processing scheme outperforms the baseline by a significant margin.

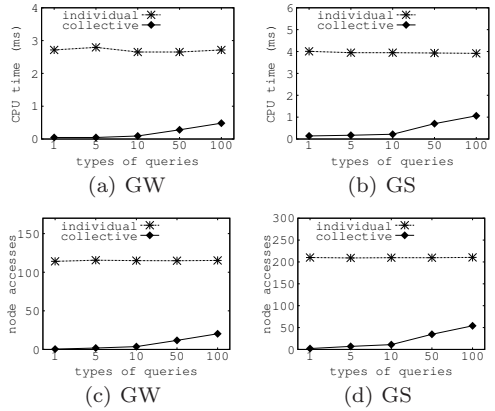


Figure 16: Collective processing by varying the number of query types

9. CONCLUSIONS

We proposed a new type of queries called the k-nearest neighbor temporal aggregate query, which provides highly customized POI retrieval by integrating the spatial distance and a temporal aggregate on a certain attribute. We designed a novel index called the TAR-tree by integrating both types of information to effectively group the entries, and therefore can support efficient processing of the kNNTA query. We performed a detailed analysis on the cost of query processing using the TAR-tree. The analysis shows that the TAR-tree has a stronger pruning power than alternatives. The accuracy of the cost analysis is validated by empirical experiments. Furthermore, we proposed two enhancements for the kNNTA query: (i) To assist users explore different results, we devised an efficient algorithm suggesting the minimum weight adjustment that can change the query results. (ii) To handle large number of queries, we proposed an effective collective processing scheme to share the index traversal among a batch of queries. We conducted extensive experiments on real-world data sets. The results validate the efficiency of the TAR-tree and the effectiveness of the two enhancements for the query.

Acknowledgments. This work is supported by Australian Research Council (ARC) Discovery Project DP130104587 and Australian Research Council (ARC) Future Fellowships Project FT120100832.

10. REFERENCES

[1] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *SIGSPATIAL*, pages 199–208, 2012.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDBJ*, 5(4):264–275, 1996.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*, pages 78–86, 1997.

[5] C. Böhm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[7] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, pages 1082–1090, 2011.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[9] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[10] H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Continuous aggregate nearest neighbor queries. *GeoInformatica*, 17(1):63–95, 2013.

[11] H. Gao, J. Tang, and H. Liu. gscorr: Modeling geo-social correlations for new check-ins on location-based social networks. In *CIKM*, pages 1582–1586, 2012.

[12] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[13] J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, and Z. He. Top-k most influential locations selection. In *CIKM*, 2011.

[14] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *TODS*, 30(2):364–397, 2005.

[15] K. Mouratidis and H. Pang. Computing immutable regions for subspace top-k queries. *PVLDB*, 6(2):73–84, Dec. 2012.

[16] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[17] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, 2001.

[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[19] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *TODS*, 30(2):529–576, 2005.

[20] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The min-dist location selection query. In *ICDE*, pages 366–377, 2012.

[21] J. Qi, R. Zhang, Y. Wang, A. Y. Xue, G. Yu, and L. Kulik. The min-dist location selection and facility replacement queries. *World Wide Web*, 17(6):1261–1293, 2014.

[22] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: Semantics and sensitivity measures. In *SIGMOD*, 2011.

[23] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present, and the future in spatio-temporal databases. In *ICDE*, pages 202–213, 2004.

[24] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.

[25] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *TKDE*, 16(12):1555–1570, 2004.

[26] Y. Tao and D. Papadias. Historical spatio-temporal aggregation. *TOIS*, 23(1):61–102, 2005.

[27] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *TKDE*, 16(10):1169–1184, 2004.

[28] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS*, pages 161–171, 1996.

[29] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.

[30] R. Zhang, J. Qi, M. Stradling, and J. Huang. Towards a painless index for spatial objects. *TODS*, 39(3):19, 2014.

[31] Y. Zheng. Location-based social networks: Users. In *Computing with Spatial Trajectories*, pages 243–276. 2011.

[32] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: concepts, methodologies, and applications. *Transaction on Intelligent Systems and Technology*, 2014.