Spatio-Temporal Aggregation Using Sketches

Yufei Tao[§] George Kollios[‡] Jeffrey Considine[‡] Feifei Li[‡] Dimitris Papadias[†]

[§]*Department of Computer Science*
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

[‡]*Department of Computer Science*
Boston University
Boston, MA, USA
{gkollios, jconsidi, lifeifei}@cs.bu.edu

[†]*Department of Computer Science*
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Abstract

Several spatio-temporal applications require the retrieval of summarized information about moving objects that lie in a query region during a query interval (e.g., the number of mobile users covered by a cell, traffic volume in a district, etc.). Existing solutions have the *distinct counting problem*: if an object remains in the query region for several timestamps during the query interval, it will be counted multiple times in the result. The paper solves this problem by integrating spatio-temporal indexes with sketches, traditionally used for approximate query processing. The proposed techniques can also be applied to reduce the space requirements of conventional spatio-temporal data and to mine spatio-temporal association rules.

1. Introduction

Despite the vast spatio-temporal literature that aims at retrieving individual objects satisfying various query predicates, most related applications are interested in aggregate information about the qualifying objects. Some examples include traffic supervision systems monitoring the number of vehicles in a district and mobile computing applications allocating bandwidth depending on the usage of each cell. Although summarized results can be obtained using conventional operations on individual objects (i.e., by accessing every single record qualifying the query), as suggested in [PTKZ02], there are several motivations for specialized aggregation methods: (i) in some cases personal data should not be stored due to legal or privacy issues; (ii) individual data may be irrelevant or unavailable; and (iii) individual data may be highly volatile and involve extreme space requirements, while the aggregate information is usually more stable over long periods, thus requiring considerably less space for storage. For example, although the distinct cars in a city area usually change rapidly, their number at each timestamp may not vary significantly, since the number of objects entering the area is similar to that exiting.

Given a rectangle *qr* and an interval *qt* (whose ending time is in the past or the present), a *spatio-temporal aggregate query* retrieves summarized information about objects that appeared in *qr* during *qt*. We consider two cases of such queries: (i) the *spatio-temporal count* that returns the total number of qualifying objects and (ii) the *spatio-temporal sum* which, assuming that each object is associated with a *measure*, outputs the sum of the measures of the qualifying objects. For instance, a measure of interest for the mobile computing scenario is the number of phone calls, in which case the corresponding spatio-temporal sum query will return the total number of phone calls made by all users in region *qr* during interval *qt*. Obviously, the spatio-temporal count is a special case of the spatio-temporal sum, where the measure of each object is 1.

The only directly-related, previous work [PTKZ02] proposes several multi-tree structures based on R-trees and B-trees (for details see Section 2.2). The shortcoming of that approach is the so-called *distinct counting problem*, i.e., if an object remains in the query region for several timestamps during the query interval, it will be counted (or summed) multiple times in the result. Since the structures of [PTKZ02] store only summarized data, information about individual objects is lost and the problem cannot be solved by duplicate elimination techniques. An effective solution to distinct counting is crucial for several applications because it enables a much richer range of decision-making queries. A natural question in traffic analysis, for example, is to ask how many cars are present in a district. The techniques of [PTKZ02] can be used to find the average number of cars per timestamp during a time interval. However, the average is not adequate in analyzing the traffic volume. For example, one might find a parking garage and a busy highway with the same average number of cars, and then assert that they have similar traffic characteristics. If we allow distinct counting queries, we can easily distinguish between the two cases. First, the highway will have a much higher turnover rate of cars entering and leaving, which will be directly reflected in a higher number of distinct cars over time. Second, this turnover rate can be quantified by comparing this number against the average. That is, if the number of distinct cars increases by x, while the average stays constant, x cars must have entered and x must have left within the time period in question.

The difficulty for supporting distinct counting queries is due to the fact that there is no way to exactly summarize distinct objects substantially better than by simply enumerating all of them. As this is impractical in most settings, we are forced to consider approximate methods. In this paper we combine *sketches*, a common approximation method, with spatio-temporal aggregate indexes. Although our main goal is to solve the distinct counting problem, the proposed techniques can also be used for alternative tasks, such as reducing the space requirements of conventional spatio-temporal databases and mining spatio-temporal association rules. The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 formally defines our problem, and describes the proposed methods. Section 4 discusses related processing tasks made possible by our techniques, while Section 5 contains an extensive experimental evaluation. Finally, Section 6 concludes the paper with directions for future work.

2. Related Work

We first review the FM algorithm for approximate counting as it constitutes part of our solution. Then, assuming basic knowledge of R-trees [G84, SRF87, BKSS90], we describe the existing approaches for aggregate processing in multi-dimensional spaces, focusing on spatio-temporal applications.

2.1 The FM algorithm and counting sketch

Estimating the number of distinct objects in a dataset has received considerable attention (see [PGF02, GGR03] and the references therein). Many methods in the literature are based on the FM algorithm developed by Flajolet and Martin [FM85] (referred to as FM in the sequel). FM requires a hash function h which takes as input an object id o, and outputs a pseudorandom integer $h(o)$ with a geometric distribution, that is, $\mathrm{Prob}[h(o)=v] = 2^{-v}$ for $v>=1$. A *sketch* consists of r bits, whose initial values are set to 0 (an appropriate choice of r is discussed later). For every object o in the dataset, FM sets the $h(o)$-th bit (of the sketch) to 1. After processing all objects, FM finds the first bit of the sketch that is still 0. Let the position of this bit be k; then the number of distinct objects is estimated as $n=1.29\times 2^k$. This is actually an *unbiased* estimate, i.e., $E[n]=1.29\times 2^{E[k]}$ where E[.] denotes the expected value of a random variable, but the variance of k is approximately 1.12. Hence, the estimate of n is frequently off by a factor of two or more [FM85].

In order to remedy this, Flajolet and Martin propose using m independent sketches, each with its own independent hash function, and averaging the resulting values. Let k_1, k_2, …, k_m be the positions of the first 0 in the m sketches respectively. The new estimate of n is 1.29×2^{k_a}, where $k_a=(1/m)\sum_{i=1}^{m}(k_i)$. This is also an unbiased estimate, but with an expected standard error of $O(m^{-1/2})$ [FM85]. However, the expected processing cost of each object increases from $O(1)$ to $O(m)$. Flajolet and Martin solve this problem using *Probabilistic Counting with Stochastic Averaging* (PCSA). PCSA applies a second hash function to choose one of the m sketches and only inserts the object into that sketch. This reduces the expected insertion cost back to $O(1)$. As a result, each sketch is responsible for approximately n/m (distinct) objects, resulting in a new formula for estimation, $1.29m\times 2^{k_a}$, with expected standard error $O(m^{-1/2})$. Figure 2.1 shows the pseudo-code of FM with PCSA.

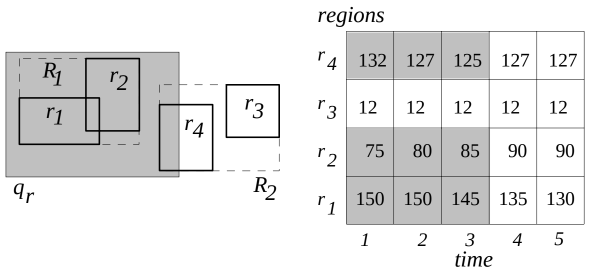
```
algorithm FM_PCSA (DS, h, m, r)
/* DS is a dataset; h is a random function such that, given an object $o\in DS$, $\mathrm{Prob}[h(o) = v] = 2^{-v}$; m is the number of sketches used; r is the number of bits in each sketch */
1.  initialize m sketches $s_1, s_2, \ldots, s_m$, each with r bits set to 0
2.  for each object o in DS
3.      randomly pick a sketch $s_i$ ($1\le i\le m$)
4.      $s_i[h(o)] = 1$
5.  $k=0$
6.  for $i=1$ to $m$
7.      for $j=1$ to $r$
8.          if $s_i[j] = 0$ then
9.              $k = k + j$;
10.             break // go to the next sketch
11. return ($1.29m\cdot 2^{k/m}$)
end FM_PCSA
```

Figure 2.1: FM Algorithm

As shown in [FM85], a proper value for r (i.e., the number of bits in a sketch) is $O(\log_2 n)$, where n is the number of distinct objects. The resulting space consumption of the FM algorithm is $O(m\cdot\log_2 n)$ when m sketches are used. Although we use FM because it is fast and accurate, our techniques can be applied with any sketch allowing union operations.

2.2 The aRB-tree

Assume the set of regions r_1, r_2, …, r_4 of Figure 2.2a and consider that at each timestamp, the number of objects in the region is given in Figure 2.2b, where the horizontal/ vertical dimension corresponds to time/region id. For instance, r_1 contains 150 objects during the first two timestamps, 145 objects the third timestamp and so on. A *spatio-temporal count query q* retrieves the total number of objects in a window qr during an interval qt, e.g., the query with qr equal to the shaded area of Figure 2.2a and *qt*=[1,3] returns the sum of the shaded values in Figure 2.2b (i.e., the cells corresponding to r_1, r_2, r_4 during the first three timestamps).



(a) Region extents (b) 2D view of the aggregates

Figure 2.2: Regions and their aggregate data

Papadias et al. [PTKZ02] propose the aRB- (aggregate R-B-) tree for the efficient processing of spatio-temporal count queries. In the aRB-tree, the extents of all regions (in this case r_1, r_2, r_3, r_4) are stored in an R-tree. Each entry of the R-tree is associated with a pointer to a B-tree that stores historical aggregate data about the entry. Figure 2.3 illustrates the aRB-tree for the example of Figure 2.2. Note that Figure 2.2a also includes the minimum bounding rectangles (MBRs) of the intermediate tree entries R_1, R_2. The B-tree of r_1, for example, contains 4 leaf entries (in the format <*time*, *agg*> (with *time* being the index key), corresponding to its 4 aggregate changes in history (i.e., no change at time 2). Intermediate B-tree entries follow the same format. For instance, the first root entry <1,445> in the B-tree for r_1 indicates that the total number of objects in r_1 during interval [1,3] is 445, while the second entry <4,265> shows that for interval [4,5] the number is 265. The B-tree of an intermediate R-tree entry summarizes the aggregated data about regions in its branch; e.g., the first leaf entry of the B-tree for R_1 <1,225> denotes that the number of objects in r_1, r_2 (i.e., the child node of R_1) at timestamp 1 is 225. Similarly the first entry of the top node <1,685> denotes that this number during the interval [1,3] is 685.

The aRB-tree facilitates aggregate processing by eliminating the need to descend nodes that are totally enclosed by the query. As an example, consider the query in Figure 2.2a (with interval *qt*=[1,3]). Search starts from the root of the R-tree. Entry R_1 is totally contained inside the query window and the corresponding B-tree is retrieved. Since the entries of the root node in this B-tree contain the aggregate data of interval [1,3] (and [4,5]), the next level of the B-tree does not need to be accessed and the contribution of R_1 (i.e., the contribution of r_1, r_2) to the query result is 685. The second root entry R_2 of the R-tree partially overlaps the query rectangle *qr*; hence, the algorithm visits its child node, where only entry r_4 intersects q_s, and thus its B-tree is retrieved. The first root entry suggests that the contribution of r_4 for interval [1,2] is 259. In order to complete the result, we have to descend the second entry and retrieve the aggregate value of r_4 for timestamp 3 (i.e., 125). The total number of objects in these regions during the interval [1,3] is the sum 685+259+125 (i.e., the numbers in the shaded cells of Figure 2.3). Nevertheless, the aRB-tree does not take into account multiple object occurrences. For example, if an object remains in r_1 at timestamps 1, 2, 3, it will be counted three times in the result. Therefore, aRB-trees are not directly applicable for applications that require distinct counting.

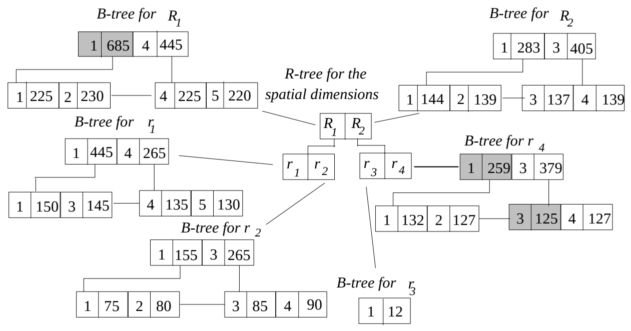


Figure 2.3: The aRB-tree

2.3 Other techniques for aggregate processing

Most work in the spatial database literature considers the following problem: given a set of objects (points, rectangles, etc.) and a rectangular query window *q*, return the number of objects intersecting *q*. Efficient structures for point data include the aP-tree [TPZ02], and the CRB-tree [GAA03], which achieves optimal performance, i.e., logarithmic query time and linear space in the 2D space. For 2D interval objects, the MVSB-tree [ZMT+01] solves the problem in logarithmic query time. Zhang et al. [ZTG02] develop a set of structures for general rectangle data with good worst-case bounds. Further, several authors [JL99, LM01, PKZT01] suggest augmenting a conventional R-tree with summarized information in the intermediate entries (in a way similar to aRB-trees) to accelerate aggregate queries for data with arbitrary extents and dimensionality. As with the aRB-tree, however, these approaches do not support distinct counting, and hence cannot be applied for our problem.

Finally, approximate query answering in conventional databases and stream management systems has been addressed using various techniques such as histograms [TGIK02], sampling [CDD+01], randomized data access [HHW97], function-fitting [CR94], etc. All these methods, however, assume a single "snapshot" of the database, and do not support spatio-temporal temporal (historical) data. The only histograms with a temporal aspect focus on spatio-temporal prediction [TSP03].

3. Distinct Spatio-Temporal Aggregation

In Section 3.1, we formally define the problem and overview the proposed methods. Then, Section 3.2 proposes the *sketch index* and Section 3.3 describes the processing of distinct spatio-temporal count queries. Section 3.4 extends the solution to spatio-temporal sum processing.

3.1 Problem definition and solution overview

We consider a set of R 2D static regions $r_1, r_2,.., r_R$ as the finest aggregation granularity (e.g., cells in a mobile phone network, road segments), and a set of n moving objects with *distinct* ids $o_1, o_2, \ldots, o_n$ (n>>R). The extents of two or more regions may overlap. Let $o(t)$ be the location of object o at time t and assume that $o(t)$ cannot be measured accurately. Instead, we know the set of objects that fall in each region every timestamp, which is indeed the case in many practical applications. For example, although the exact location of every mobile user is not usually known, it is possible to decide the cell that services the user.

Given an aggregate query q with window qr and time interval qt, we define the set $M(q)$ of *matching objects* as:

$$M(q) = \{o_i \mid \exists \text{region } r_j \text{ \& time } t \in qt \text{ such that, } o_i(t) \in r_j \text{ and } r_j \text{ intersects } qr\}$$

Note that $M(q)$ is defined through the spatial regions (i.e., the finest aggregation granularity). In particular, an object $o_i \in M(q)$ if it appears in some region that qualifies the query, even though the object itself does not lie in qr during qt. Now we are ready to define the query types considered in this paper.

Problem 3.1: A <u>*distinct* (*spatio-temporal*) *count*</u> (DC, for short) query q (with window qr and time interval qt) returns the number of matching objects, or more formally: $DC(q) = |M(q)|$.

Assuming that each object o_i carries a measure w_i (invariant with time), then a <u>*distinct sum*</u> (DS) query q retrieves the sum of the measures of all matching objects: $DS(q)=\sum w_i$, where $o_i \in M(q)$. ■

The exact computation of $DC(q)$ requires working space at least linear to the number of distinct objects. This lower bound also applies to $DS(q)$ since DC queries are a special case of DS with measure w_i=1.Thus, if n is the number of distinct objects and T is the total number of timestamps in history, any solution that solves DC/DS queries precisely needs $\Omega(n \cdot T)$ space. This is prohibitive in practice since both n and T may be very large. To overcome this problem, we develop a structure that answers DC/DS queries *approximately*, consuming $O(m \cdot R \cdot T \cdot \log n)$ space, where R is the number of regions, and m is an adjustable constant that, as explained shortly, determines the tradeoff between overhead and approximation accuracy.

Using the FM algorithm discussed in Section 2, for each region r_i ($1 \le i \le m$) and timestamp t we maintain a sketch $s_i(t)$ that captures the (ids of) objects in r_i at t. Figure 3.1 presents the system for distinct aggregation. At each timestamp, every object reports its id (or measure, for DS queries) to the region that covers its location. The region has a *sketch generator* that creates the corresponding sketches based on the object information, and transmits them to the database. To obtain sketches for DC queries, the generator simply performs the PCSA algorithm described in Section 2.1. The algorithm for DS is more complex and discussed separately in Section 3.4.

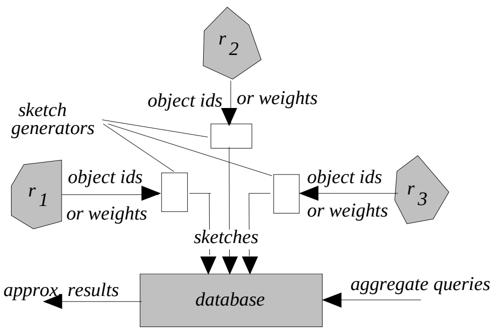
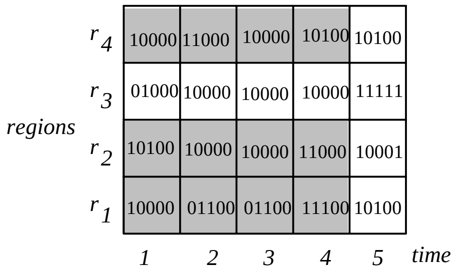


Figure 3.1: System architecture

The sketches received by the database can be stored in a two dimensional array shown in Figure 3.2 (assuming R=4 regions). This is similar to the storage scheme in Figure 2.2b, except that each cell now contains sketches instead of the actual number of objects in each region per timestamp.



regions					
r_4	10000	11000	10000	10100	10100
r_3	01000	10000	10000	10000	11111
r_2	10100	10000	10000	11000	10001
r_1	10000	01100	01100	11100	10100
	1	2	3	4	5 time

Figure 3.2: Conceptual sketch storage model

Lemma 3.1: Using the sketch $s_i(t)$ in cell (r_i, t), we can estimate the distinct number $n_i(t)$ of objects in r_i at timestamp t. ■

This lemma results from the straightforward application of the FM estimation algorithm. Namely, by identifying the position k of the left most 0 in $s_i(t)$, we can estimate $n_i(t)$ as 1.29×2^k (note that the sketches do not correspond to the numbers in Figures 2.2a). The following lemma illustrates another important property, which constitutes the rationale underlying the proposed method.

Lemma 3.2: Using the OR of the sketches of c cells (r_1, t_1), (r_{x2}, t_2), …, (r_{xc}, t_c), (two cells may have the same region *or* timestamp), we can estimate the number of distinct objects o that appear in some region r_{xj} at time t_j.■

The correctness of this lemma is due to the well-known fact [FM85] that the sketch (generated by FM) for the union of several datasets is identical to the OR of the individual sketches of each dataset. In other words, let D_S be the set of objects that satisfy the condition stated in Lemma 3.2; then, the sketch of D_S is *exactly the same* as the $\text{OR}_{i=1}^{c} s_{xi}(t_i)$.

Lemma 3.2 indicates a simple algorithm for approximate DC processing. Consider, for example, a DC query with window *qr* intersecting r_1, r_2, r_4 and *qt*=[1,4]. The goal is to compute the OR of the sketches in the shaded cells. In this case, the result is 11100, and since the left-most 0 is at position 4, the (approximate) result equals 1.29×2^4=10.32. This algorithm, however, is "conceptual", meaning that it does not consider the actual access paths for retrieving the necessary sketches. Efficient sketch retrieval cannot be achieved using conventional data warehouses because there is no natural ordering on the region axis; therefore, regions intersecting *qr* may not be consecutive. Furthermore, there is no pre-defined hierarchy on regions, rendering traditional group-by techniques inapplicable. Motivated by this, in the next section we introduce the *sketch index* to accelerate the sketch retrieval.

3.2 Sketch index structure

The sketch index is similar to the aRB-tree in terms of structure, but differs in the query algorithms. Figure 3.3 shows an example for the data in Figure 3.2. An R-tree indexes the regions r_1, r_2,…, r_4. Each R-tree entry is associated with a B-tree that records the historical sketches of the corresponding region (or regions in its sub-tree). For example, the B-tree of region r_1 consists of 4 leaf entries (in the format <*time*, *sketch*>), indicating its 4 sketch changes in history (i.e., no change at time 3). The sketch 11100 of the first root entry in this B-tree equals the OR of all the sketches (i.e., 10000, 01100) in its subtree. The same rule applies to all intermediate B-tree entries.

Consider the first leaf entry <1,10100> in the B-tree of R_1. Its sketch 10100 equals the OR for sketches of r_1 and r_2 (i.e., 10000, 10100, respectively) at time 1. In general, for each intermediate R-tree entry R_i, its sketch at any time t, is the OR of sketches of all the regions in the subtree (of R_i) at t. The sketch index is a dynamic structure, and its incremental maintenance algorithms follow those of the aRB-tree due to the similarity of the structures.

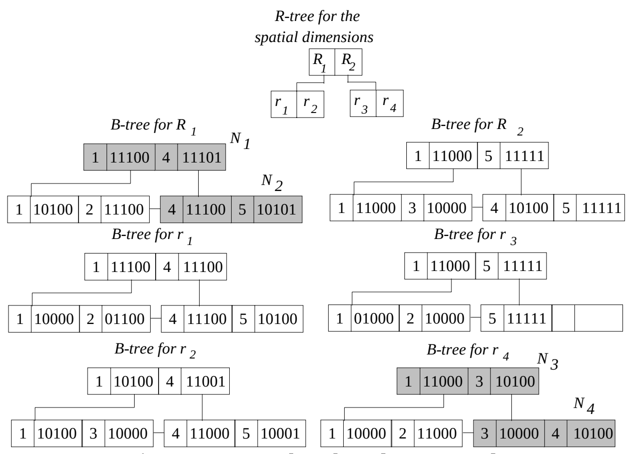


Figure 3.3: A sketch index example

3.3 Query processing using the sketch index

A straightforward algorithm for answering DC queries using the sketch index is to perform the search in a way similar to that in the aRB-tree. To illustrate this, we assume, for simplicity, the same extents of regions (r_1, r_2,…, r_4) and intermediate R-tree entries (R_1, R_2) as those in Figure 2.2a. Consider again the query q with window *qr* (shown in Figure 2.2a) and interval *qt*=[1,4]. The search algorithm initiates a *result sketch RS* with all bits set to 0, and gradually updates it. Specifically, the search starts from the root of the R-tree. Since R_1 is contained in *qr*, we fetch the root N_1 of its B-tree, where the first entry <1,11100> indicates that the OR of all sketches in its sub-tree during [1,3] is 11100, which becomes the new value of *RS*. The child node N_2 of the second root entry must be searched. Inside this node, entry <4,11100> qualifies *qt*, and thus its sketch is OR-ed with *RS* (which, however, incurs no change to *RS*). Next the algorithm backtracks to the R-tree and, since R_2 partially intersects *qr*, accesses its child node, in which the only entry intersecting *qr* is r_4. Hence, it visits N_3 and N_4 producing the final sketch *RS*=11100. In Figure 3.3, the visited B-tree nodes are shaded.

The above algorithm applies spatial and temporal conditions (using *qr* and *qt* respectively), but completely ignores the pruning power of the sketches themselves. Notice that in the previous example *RS* is already set to 11100 (i.e., the final result) at a very early stage of the search process (i.e., after accessing the root of the B-tree of R_1). In other words, sketches of subsequent nodes do not affect the final result at all. This motivates the following pruning heuristic.

Heuristic 3.1: Let *RS* be the current result sketch, and e an intermediate B-tree entry whose associated sketch is s_e. Then, the sub-tree of e can be pruned if (s_e OR *RS*) = *RS*. ■

According to this rule, the processing of the above query can avoid visiting node N_4 because the sketch (10100) of its parent entry <3,10100> satisfies 10100 OR RS = RS. The implication is that, in order to maximize the effectiveness of Heuristic 3.1, we should first try to maximize the 1's in RS, before descending intermediate (B-tree) entries. In general, we should "postpone" visiting nodes that may be pruned later as more bits of RS are set. The next question is: which node accesses are avoidable, and which ones are necessary?

To answer this question, let S_{RE} be the set of R-tree entries whose B-trees must be accessed. Equivalently, each entry e in S_{RE} satisfies the following conditions: (i) its MBR is covered by query rectangle qr (or its MBR intersects qr if e is a leaf) and (ii) none of its ancestor entries satisfies (i). In the example of Figure 3.3, S_{RE}={R_1,r_4}. Evidently, accesses to the roots of their respective B-trees are unavoidable[1]. Hence we visit all of them (in Figure 3.3, nodes N_1 and N_3), and examine the entries therein. Some of these entries allow us to set (possibly many) bits of RS without any further node access. The first entry of N_1 has *lifespan* [1,3] (3 is derived from the timestamp of the next entry 4), which is *contained* in the query interval qt. So we can immediately update RS to its sketch 11100. Similarly, the lifespan [1,2] of the first entry in N_3 is also contained in qt; hence its sketch 11000 is also taken into account, but does not change RS.

Now let us consider the remaining entries in N_1 and N_3, namely, <4,11101> and <3,10100>. Although, their lifespans are not contained in qt=[1,4], Heuristic 3.1 eliminates <3,10100>. Nevertheless, <4,11101> is not pruned by the heuristic because 11101 OR RS = 11101 ≠ RS. However, recall that our objective is not to retrieve the complete final RS. Instead, we are interested in the position of the left-most bit that is still 0. What is the *possible* left-most position (of the final RS) in this case? Given the current RS=11100 and entry <4,11101>, the answer is 4 (i.e., the left-most 0 must be at the 4-th bit), since the first 3 bits of both RS and the entry's sketch are all 1. Therefore, the access to the child node of this entry *can also be avoided*, because (even if we actually visit it) the only possible change to RS is to set the 5-th bit to 1, which does not affect our estimation. This observation leads to another heuristic.

Heuristic 3.2: Let S_U be the OR of the sketches of the entries whose sub-trees cannot be pruned so far. If p is the position of the left-most 0 in (RS OR S_U), then the sub-tree of an intermediate (B-tree) entry e can be pruned if its sketch s_e satisfies the following condition:

$$\left(RS \text{ AND } \underbrace{1...1}_{p-1}0...0 \right) = \left(RS \text{ OR } s_e \text{ AND } \underbrace{1...1}_{p-1}0...0 \right) \quad ■$$

Heuristic 3.2 subsumes 3.1 by providing a more general condition. Specifically, instead of requiring all bits of RS and (RS OR s_e) to be identical, it prunes s_e if only the first p–1 bits (of the these sketches) are the same (where p is decided by RS and S_U together). The third heuristic indicates a good access order for the child nodes of entries not pruned by Heuristic 3.2.

Heuristic 3.3: Given a set of qualifying entries, we visit their child nodes in descending order of the number of 1's in their sketches. ■

We use a heap to manage the entries which cannot be pruned yet, using the numbers of 1's in their sketches as the sorting keys. As an example, consider another query whose (i) rectangle qr intersects all regions (r_1, r_2, r_3, r_4), and contains the MBR of R_1 but not R_2, and (ii) interval is qt=[1,4]. In this case, the algorithm first visits the roots of the B-trees of R_1, r_3, r_4, after which RS=11100, and the heap contains two entries <1,11111> (from the root of r_3's B-tree) and <4,11101> (the second entry in the B-tree of R_1), both of which cannot be pruned. The algorithm will visit the child node of <1,11111> next since it has more 1's. Figure 3.4 illustrates the pseudo-code of the improved algorithm (referred to as *sketch-prune* in the sequel).

```
algorithm sketch_prune (qr, qt)
1.   initiate a "max" heap H accepting entries of the form
     <B-tree entry e, key>; set all bits of RS to 0
2.   obtain the set $S_{RE}$ of R-tree entries whose B-trees must be
     searched
4.   for each of entry e in $S_{RE}$
5.       for each entry e' in the root of e.btree
6.           process_intermediate(e', $S_{RE}$, H) }
7.   while (H is not empty)
8.       $S_U$ = the OR of the sketches of the entries in H
9.       p = the position of the left-most 0 of $S_{RE}$ OR $S_U$
10.      remove the top entry <e, key> from H; let the sketch
         of e be $s_e$
11.      let s be a sketch whose left-most (p–l) bits are 1
         while the others are 0
12.      if (RS OR $s_e$ AND s) ≠ (RS AND s)
13.          for each entry e' in e.child (its sketch $s_{e'}$)
14.              if (e.child is leaf) and (e'.lifespan intersects qt)
15.                  RS=$s_{e'}$ OR RS
16.              if (e' is an intermediate node)
17.                  process_intermediate(e', $S_{final}$, H)
18.  let k be the position of the left-most 0 in RS
19.  return $1.29 \times 2^k$
end sketch_prune

Algorithm process_intermediate (e, $S_{final}$, H)
/* e is an intermediate entry in the B-tree with sketch $s_e$; RS is
the current result sketch; qt is the query interval; H is the heap*/
1.   if e.lifespan is contained in $q_T$ then RS=RS OR $s_e$;
2.   else if (e.lifespan intersects $q_T$)
3.       insert <e, number of "1" in $s_e$> into H
end process_intermediate
```

Figure 3.4: The *sketch-prune* algorithm

[1] Unlike the aRB-tree, we do not store sketches in the R-tree entries because this would decrease the node fanout.

Heuristic 3.3 provides a reasonably "good" access order, but other more sophisticated and potentially better access orders exist. For instance, the order may be decided according to the number of additional bits in *RS* that may be set (to 1) by this entry. Specifically, assume *RS*=11000, and two sketches 11100 and 00110; then according to this order, the second sketch will be processed first (although it has fewer 1's) since it may set two bits of RS (while the first sketch can set only one bit). This, however, requires adjusting the sorting keys of the entries in the heap as the algorithm proceeds (and *RS* changes), which may be expensive if the heap size is large.

The description so far assumes that only one sketch is maintained per B-tree entry; however, the *sketch-prune* algorithm can be easily modified to support multiple sketches (which, as discussed in Section 2.1, leads to higher accuracy) as follows. First, Heuristics 3.1 and 3.2 are applied individually for each sketch to prune the entries that qualify the heuristic conditions in all sketches. Then, Heuristic 3.3 determines the access order with respect to the total number of 1's in all the sketches of an entry. The storage of the sketch index at each timestamp is linear to (i) the number R of regions, (ii) the length $\log_2 n$ of each sketch, and (iii) the number m of sketches used. As a result, the total space complexity (for all T timestamps in the history) is $\mathcal{O}(m \cdot R \cdot T \cdot \log n)$

3.4 Supporting distinct sum queries

The proposed method for DC (distinct count) processing can be applied to DS (distinct sum) queries, by modifying the sketches of the leaves. The resulting sketches are then indexed and queried in exactly the same way as described in the previous section. Hence, it suffices to illustrate the specialized algorithm for creating the sum sketches. Specifically, the problem is stated as follows: given a dataset with (possibly duplicate) tuples in the form (*object* o, *measure* w), estimate the sum of measures of the *distinct* objects. That is, if an object appears with the same measure several times, its measure is added only once.

We solve this problem by reducing it to DC processing. Given an input record (o, w), we simulate the FM sketch generation algorithm by inserting w *different* elements $(o,\theta_{1,w})$, $(o,\theta_{2,w})$,…, $(o,\theta_{w,w})$, where $\theta_{i,w}$ are special symbols to distinguish these elements. Consequently, the estimated "count" using FM is actually the sum of the w's of distinct records (o, w) (comparing both o and w) in the original problem[2]. The disadvantage of this approach is that, if w is large, inserting w different elements will be expensive. Here we briefly describe an alternative algorithm for generating sum-sketches that remedies this problem (more details and proofs may be found in [CLKB04]). The main idea is to leverage the observation of [FM85] that the first few (say x) bits are (almost) definitely 1. Since the FM estimator is only concerned with the first 0 in the final sketch, we only need to consider the part (of the sketch) starting at the (x+1)-th bit, or in other words, we can *ignore* the insertion of those elements (let their number be y) that will set the first x bits. Recall that, since the hash function (used by FM) has the property that, the probability of setting the i-th bit equals 2^{-i}, each element has probability $\sum_{i=1}^{x}(2^{-i})$ to set (any of) the first x bits. Hence, y follows the Binomial distribution[3] Bin(w, $\sum_{i=1}^{x}(2^{-i})$). As a result, (in order to decide how many bits *after* the x-th one is set) we only need to insert w–y elements, and obtain the resulting sketch. Let the left-most 0 of this sketch be at position k'; then the corresponding position k in the sketch of inserting all w elements equals $x+k'$.

There remains only one question: what is a good value for x? The analysis of [FM85] observes that inserting w distinct items sets the first $x = \log_2 w - 2\log_2\log_2 w$ bits of the resulting sketch to 1 with high probability. This value is adopted in our implementation. Finally, we note that this method can also be combined with PCSA to improve accuracy, as shown in Figure 3.5.

```
algorithm sum_PCSA ($DS$, $h$, $m$, $r$)
/* dataset $DS=\{(o_1,w_1),(o_2,w_2),\ldots\}$; $h$ is a random function such
that, Prob[$h(o,w)=v$]=$2^{-v}$; $m$ is the number of sketches used; $r$ is
the number of bits in each sketch */
1.  init $m$ sketches $s_1, s_2, \ldots, s_m$, each with $r$ bits, all set 0
2.  for each $(o, w)$ in $DS$ do
3.      randomly pick a sketch $s_i$ ($1 \le i \le m$)
4.      $x = \log_2 w - 2\log_2\log_2 w$;
5.      for j=1 to $x$
6.          $s_i[j] = 1$;
7.      for $j$=1 to $w - \mathrm{Bin}(w, \sum_{i=1}^{x}(2^{-i}))$
8.          $s_i[x+h(o,j)] = 1$;
9.  $k$=0
10. for $i$=1 to $m$ do
11.     for $j$=1 to $r$ do
12.         if $s_i[j] = 0$ then
13.             $k = k + j$;
14.             break; // go to the next sketch
15. return ( $1.29m \cdot 2^{k/m}$ )
end sum_PCSA
```

Figure 3.5: Sketch generation and estimation for DS

4. Extensions

In this section we present the application of the proposed techniques to related spatio-temporal problems. Section 4.1 uses sketches to reduce the size of general spatio-temporal databases and enhance the performance of

[2] An alternative approach is to insert elements of the form (o,θ_i), in which case the estimated "count" is the sum of the maximum w's for each distinct o.

[3] For Binomial distribution x~Bin(n,p), the probability Prob[x=m] is $\binom{n}{m}p^n(1-p)^{n-m}$.

aggregate processing. Section 4.2 applies sketches to mine spatio-temporal association rules.

4.1 Approximating general moving data

The discussion in Section 3 assumes a set of regions that constitute the finest aggregation granularity, which may not be the case for the conventional spatio-temporal databases. In this scenario, each object o reports its location (x,y) at each timestamp t to the database, which maintains a tuples in the form $<o,x,y,t>$. Evidently, the size of the database table grows *continuously*, so that eventually it becomes prohibitively large (especially if the number of monitored objects is high). In addition to the space complexity $O(n \cdot T)$ (where n is the number of objects, and T is the number of timestamps in the history), this deteriorates query performance. In the sequel, we show that, if the goal is to support aggregate queries, we can reduce the size and query overhead significantly, at the trade-off of some small error (around 15% as shown in our experiments).

We *manually* impose a $res \times res$ regular grid over the data space (i.e., each cell of the grid has length $1/res$ of the total axis extent), where res is a parameter called *resolution*. Then, the sketch index is directly applicable by treating the grid cells as the finest aggregate granularity. It is worth mentioning that if the number of cells is relatively small (i.e., low resolution), the approximation tends to over-estimate the actual result because an object, which does not fall in the query rectangle qr, but in a cell intersecting qr, will also be counted. This problem can be alleviated by setting res to a sufficiently large value (e.g., 50 in our implementation). It is easy to verify that the space complexity is $O((res)^2 \cdot T \cdot \log n)$, or $O(T \cdot \log n)$ when res is a constant. As a further improvement, observe that we can actually remove the R-tree from the sketch index, because the cells indexed by the R-tree are regular. Specifically, it suffices to introduce a hierarchical decomposition as shown in Figure 4.1, where the grid at level i has resolution 2^i, and the maximum level equals $\log_2 res$.

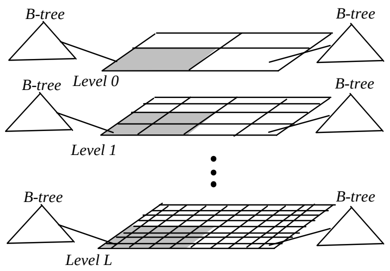


Figure 4.1: Grid-based approximation

Note that, this hierarchy implicitly defines the parent-child relation among cells of different levels (e.g., the shaded cell at level 0 is the ancestor of all the shaded cells in the lower levels). As in sketch indexes, each grid cell is associated with a B-tree managing the historical sketches about objects in its extent (cells in intermediate levels resemble intermediate entries in the R-tree of a sketch index). Given a rectangle qr, we can easily decide the set of cells (in a particular grid) that (i) partially intersect or (ii) are contained in qr. As with the R-tree of a sketch index, descending the hierarchy is only necessary for case (i), because in case (ii) the B-tree is accessed directly. It can be proven that, given the finest resolution res, the algorithm accesses $O(res \cdot h_B)$ pages (for any query), where h_B is the maximum height of the B-tree.

4.2 Mining spatio-temporal association rules

Consider a user in region r_i at time t. What is the probability p that this user will appear in region r_j by time $t+T$? We denote such a spatio-temporal association rule with the syntax $(r_i,T,p) \Rightarrow r_j$. Inferring such rules is important in practice. For example, in mobile computing, they can identify trends in user movements and lead to better allocation of antenna bandwidth to cater for potential network congestions in the near future. Additional constraints, such that r_i and r_j must be within certain distance, may also be specified.

By maintaining the sketches of all regions at each timestamp as in Figure 3.2, we can answer the following question easily: given specific r_i, r_j, and a timestamp t, how many users that are in r_i at t, appear in r_j at *any* of the following T timestamps (i.e., $t+1, \ldots, t+T$)? Let $s_i(t)$ be the sketch of r_i at time t, and $s_j(t), \ldots, s_j(t+T)$ be the sketches of r_j at the subsequent T timestamps. We first estimate the number n_1 of objects at r_i at time t (using $s_i(t)$), and the number n_2 of objects at r_j during time interval $[t+1, t+T]$ (using $\mathrm{OR}_{i=t+1}^{t+T}(s_j(t+i))$). Next, we estimate the total number n_3 of objects that appeared either in r_i (at time T) or in r_j during $[t+1, t+T]$ (using $\mathrm{OR}_{i=t+1}^{t+T}(s_j(t+i))$ OR $s_i(t)$). Then, the number of objects that appear in r_i at time t and then appear in r_j during $[t+1, t+T]$ equals $n_1+n_2-n_3$. This idea naturally leads to a simple brute-force algorithm for discovering the association rules, which as shown in Figure 4.2, checks all possible instances of (r_i, r_j, t).

```
algorithm associate_rule_mining ($T$, $p$, $c$)
/* $T$ is the horizon; $p$ is the appearance probability; $c$ is the confidence factor */
1.  for each region $r_i$
2.      for each region $r_j$
3.          $sample$=0; $witness$=0
4.          for each timestamp $t$ in history
5.              sample++
6.              $s' = s_j(t+1)$ OR $s_j(t+2)$ OR ... OR $s_j(t+T)$
7.              $n_1$=FM estimate from $s_i(t)$; $n_2$=FM estimate from $s'$;
                $n_3$=estimate from $s_i(t)$ OR $s'$
8.              if $(n_1+n_2-n_3)/n_1>p$ then $witness$++
9.          if ($witness/sample>c$) then output rule $(r_i,T,p)\Rightarrow r_j$
end associate_rule_mining
```

Figure 4.2: Algorithm for mining association rules

5. Experiments

This section experimentally evaluates the proposed methods. First, Section 5.1 examines the efficiency of the sketch-index in answering aggregate queries. Then, Section 5.2 studies the effect of approximating spatio-temporal data, while Section 5.3 presents preliminary results for mining association rules.

5.1 Performance of sketch-indexes

Due to the lack of real spatio-temporal datasets we generate synthetic data in a way similar to [SJLL00, TPS03] aiming at simulation of air traffic. We first adopt a real spatial dataset [Tiger] that contains 10k 2D points representing locations in the Long Beach county (the data space is normalized to unit length on each dimension). These points serve as the "airbases". At the initial timestamp 0, we generate 100k air planes, such that each plane (i) is associated with a number of passengers uniformly generated in [200,300], (ii, iii) a source and a destination that are two random different airbases, and (iv) a speed uniformly distributed in [0.02, 0.04] (the velocity direction is determined by the orientation of the line segment connecting its source and destination airbases). At the subsequent 100 timestamps, all planes move continually according to their velocities. Once a plane reaches its destination, it flies towards another (randomly selected) airbase with a new velocity (also uniform in [0.02, 0.04]). At each timestamp, every plane reports to its nearest airbase, or specifically, the database consists of tuples in the form <*time t*, *airbase b*, *plane p*, *passenger # a*>, specifying that plane *p* with *a* passengers is closest to base *b* at time *t*.

A spatio-temporal count/sum query has two parameters: the length *qrlen* of its query (square) window and the number *qtlen* of timestamps covered by its interval. The actual extent of the window (interval) distributes uniformly in the data space (history, i.e., timestamps [0,100]). A count query retrieves the number of *distinct* air planes that report to airbases in *qr* during *qt*, while a sum query returns the sum of these planes' passengers. A *workload* consists of 100 queries with the same parameters *qrlen* and *qtlen*.

The disk page size is set to 1k in all cases (the relatively small page size simulates situations where the database is much more voluminous). Since there does not exist any specialized method for distinct spatio-temporal aggregation, we compare the sketch-index to the following *relational approach* that can be implemented in a DBMS. Specifically, we index the 4-tuple table <*t*,*b*,*p*,*a*> using a B-tree on the *time t* column. Given a count query (with window *qr* and interval *qt*), we issue:

SELECT <u>distinct</u> *p*

FROM <*t*,*b*,*p*,*a*>

WHERE $t \in qt$ & *b* contained in *qr*.

The performance of each method is measured as the average number of page accesses (per query) in processing a workload. For the sketch-index, we also report the average (relative) error of the workload. Specifically, let act_i and est_i be the actual and estimated results of the *i*-th query in the workload; then the error equals $(1/100)\sum_{i=1}^{100}|act_i - est_i|/act_i$. For sketch-indexes we set the number of bits in each sketch to 24, and vary the number of sketches.

The first experiment evaluates the space consumption. Figure 5.1 shows the sketch index size as a function of the number of sketches used (count- and sum-indexes have the same results). As expected, the size increases when more sketches are included, but is usually considerably smaller than the database size (e.g., for 16 signatures, the size is only 40% the database size).

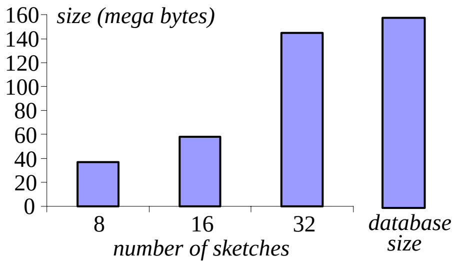
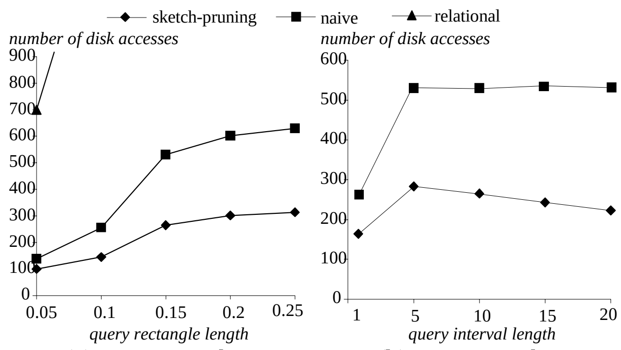


Figure 5.1: Size comparison

Next we demonstrate the superiority of the proposed *sketch-pruning* query algorithm, with respect to the *naïve* one that applies only spatio-temporal predicates. Figure 5.2a illustrates the costs of both algorithms for count-workloads with *qtlen*=10 and various *qrlen* (the index used in this case has 16 sketches). For comparison, we also illustrate the performance of the relational method, which, however, is clearly incomparable (for *qrlen*≥0.1, it is worse by an order of magnitude); hence in the sequel we omit this technique.

Sketch-pruning always outperforms *naïve* (e.g., eventually two times faster for *qrlen*=0.25). The improvement increases with *qrlen*, since queries returning larger results tend to set bits in the result sketch more quickly, thus enhancing the power of Heuristics 3.1 and 3.2. In Figure 5.2b, we compare the two methods by fixing *qrlen* to 0.15 and varying *qtlen*. Similar to the findings of [PTKZ02][4], both algorithms demonstrate "step-wise" growths in their costs, while *sketch-pruning* is again significantly faster. The experiments with sum-workloads lead to the same observations, and therefore we evaluate sketch-indexes using *sketch-pruning* in the rest of the experiments.

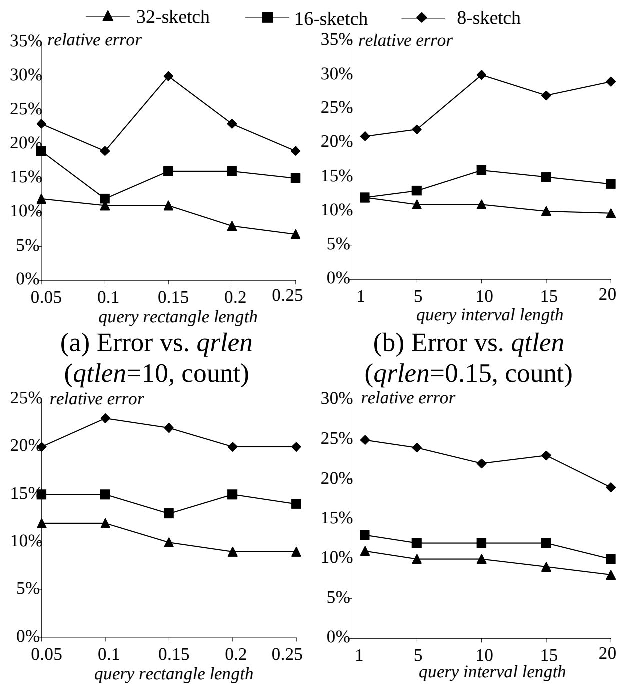
[4] As explained in [PTKZ02], query processing accesses at most two paths from the root to the leaf level of each B-tree, regardless the length of the query interval.



(a) Cost vs. *qrlen* (*qtlen*=10) (b) Cost vs. *qtlen* (*qrlen*=0.15)

Figure 5.2: Superiority of *sketch-pruning* (count)

As discussed in Section 2, a large number of sketches reduces the variance in the resulting estimate. To verify this, Figure 5.3a plots the count-workload error of indexes using 8-, 16-, and 32- sketches, as a function of *qrlen* (*qtlen*=10). As expected, the 32-sketch has the lowest error (below 10%), and its accuracy is most stable (it increases slowly with *qrlen*). When only 8 sketches are used, however, the error rate is much higher (up to 30%), and has serious fluctuation, indicating the prediction is not robust. The performance of 16-sketch is in between these two extremes, or specifically, its accuracy is reasonably high (average error around 15%) and stable (much less fluctuation than 8-sketch).

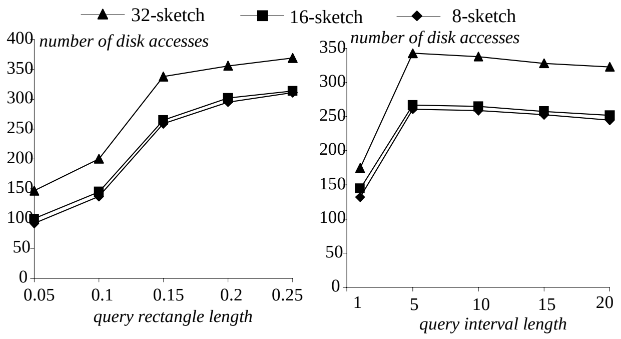


(a) Error vs. *qrlen* (*qtlen*=10, count) (b) Error vs. *qtlen* (*qrlen*=0.15, count)

(c) Error vs. *qrlen* (*qtlen*=10, sum) (d) Error vs. *qtlen* (*qrlen*=0.15, sum)

Figure 5.3: Accuracy of the approximate results

The same phenomena are confirmed in Figures 5.3b (where we fix *qrlen* to 0.15 and vary *qtlen*), and 5.3c and 5.3d (results for sum-workloads). Although a larger number of sketches improves the estimation accuracy, it also leads to higher space requirements (as shown in Figure 5.1), and processing costs. To demonstrate this, Figures 5.4a and 5.4b show the number of disk accesses for the settings of Figures 5.3a and 5.3b. All indexes have almost the same behavior, while the 32-sketch is clearly more expensive than the other two indexes. The interesting observation is that 8- and 16-sketches have almost the same overhead due to the similar heights of their B-trees. Since the diagrams for sum-workloads illustrate (almost) identical results, they are omitted to avoid redundancy.



(a) Cost vs. *qrlen* (*qtlen*=10) (b) Cost vs. *qtlen* (*qrlen*=0.15)

Figure 5.4: Costs of indexes with various signatures

Summary: The sketch index constitutes an effective method for approximate spatio-temporal (distinct) aggregate processing. Particularly, the best tradeoff (between space, query time, and estimation accuracy) is obtained by 16 sketches, which leads to size around 40% the database, fast response time (an order of magnitude faster than the relational method), and less than 15% average relative error.

5.2 Approximating spatio-temporal data

We proceed to study the efficiency of using sketches to approximate spatio-temporal data (proposed in Section 4.1). For this purpose, we generate data in the same way as in the last section, except that at each timestamp all airplanes report their locations to a central server (instead of their respective nearest bases). Specifically, the server maintains a table in the form <*time t*, *plane p*, *x*, *y*>, where (*x*,*y*) denotes the coordinates of *p* at time *t*. A count query (with parameters *qrlen* and *qtlen*) retrieves the number of distinct planes satisfying the spatial and temporal conditions. For comparison, we index the table using a 3D R*-tree on the columns *time*, *x*, and *y*. Given a query, this tree facilitates the retrieval of all qualifying tuples, after which a post-processing step is performed to obtain the number of distinct planes (in the sequel, we refer to this method as *3DR*). As mentioned earlier, our compression method introduces a regular *res*×*res* grid of the data space, where the resolution *res* is a parameter. We adopt 16 sketches because, as mentioned earlier, this number gives the best overall performance.

Figure 5.5 compares the sizes of the resulting sketch indexes (obtained with resolutions *res*=25, 50, 100) with the database size. In all cases, we achieve high compression rate (e.g., the rate is 25% for *res*=25). To evaluate the query efficiency, we first set the resolution to the median value 50, and use the sketch index to answer workloads with various *qrlen* (*qtlen*=10).

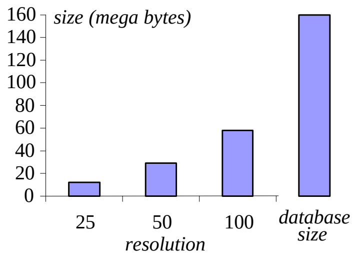
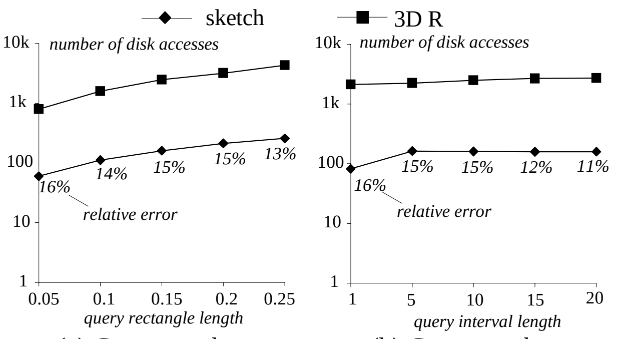


Figure 5.5: Size reduction

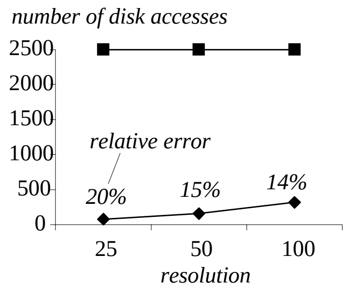
Figure 5.6a shows the query costs (together with the error in each case), and compare them with those of the *3DR* method. The sketch index is faster than *3DR* by an order of magnitude (note that the vertical axis is in logarithmic scale), while at the same time it achieve high accuracy (around 15% error). Figure 5.6b confirms these observations using workloads with different *qtlen*. Finally, we examine the effect of resolution *res* using a workload with *qrlen*=0.15 and *qtlen*=10. As shown in Figure 5.6c, larger *res* incurs higher query overhead, but improves the estimation accuracy.

<u>Summary</u>: The proposed sketch method can be used to efficiently approximate spatio-temporal data for aggregate processing. It consumes significantly smaller space, and answers a query almost in real-time with low error.



(a) Cost vs. *qrlen* (*qtlen*=10, *res*=25)

(b) Cost vs. *qtlen* (*qrlen*=0.15, *res*=25)



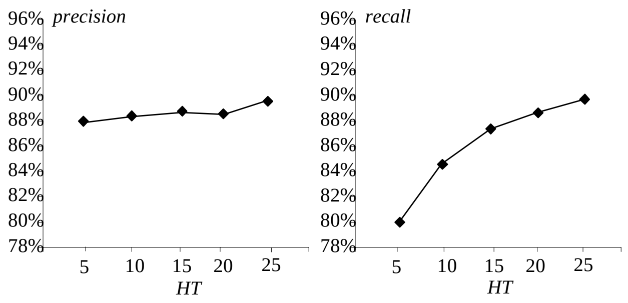
(c) Cost vs. *res* (*qrlen*=0.15, *qtlen*=10)

Figure 5.6: Query efficiency (costs and error)

5.3 Mining association rules

To evaluate the proposed algorithm for mining spatio-temporal association rules, we first artificially formulate 1000 association rules in the form (r_1,T,90%)⇒r_2 (with 90% confidence), such that (i) r_1 and r_2 are two regions randomly picked from 10k ones, (ii) each region appears in at most one rule, and (iii) *T* is the same for all rules. Then, at each of the following 100 timestamps, we assign 100k objects to the 10k regions following these rules. We execute our algorithms (using 16 sketches) to discover these rules, and measure (i) the *precision*, the number of "correct" rules divided by the total number of discovered rules, and (ii) *recall*, the percentage of the artificial rules successfully mined.

Figures 5.7a and 5.7b illustrate the precision and recall as a function of *T* respectively. Our algorithm has good precision (close to 90%) for all *T*, meaning that the majority of the rules discovered are correct. The recall, however, is relatively low for short *T*, but gradually increases (90% for *T*=25). This is expected because, as evaluated in the previous sections, the estimation error decreases as the query result becomes larger (i.e., the case for higher *T*).



(a) Precision vs. *HT*

(b) Recall vs. *HT*

Figure 5.7: Efficiency of the mining algorithm

<u>Summary</u>: The preliminary results justify the usefulness of our mining algorithm, whose efficiency improves as *T* increases.

6. Conclusions

While efficient aggregation is the objective of most spatio-temporal applications in practice, the existing solutions either incur prohibitive space consumption and query time, or are not able to return useful aggregate results due to the distinct counting problem. In this paper, we propose the sketch index that integrates traditional approximate counting techniques with spatio-temporal indexes. Sketch indexes use a highly optimized query algorithm resulting in both smaller database size and faster query time. Our experiments show that while a sketch index consumes only a fraction of the space required for a conventional database, it can process queries an order of magnitude faster with average relative error less than 15%.

While we chose to use FM sketches, our methodology can leverage any sketches allowing union operations. Comparing the efficiency of different sketches constitutes a direction for future work, as well as further investigation of more sophisticated algorithms for mining association rules. For example, heuristics similar to those used for searching sketch indexes may be applied to improve the brute-force implementation.

ACKNOWLEDGEMENTS

Yufei Tao and Dimitris Papadias were supported by grant HKUST 6197/02E from Hong Kong RGC. George Kollios, Jeffrey Considine and were Feifei Li supported by NSF CAREER IIS-0133825 and NSF IIS-0308213 grants.

References

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[CDD+01] Chaudhuri, S., Das, G., Datar, M., Motwani, R., Narasayya, V. Overcoming Limitations of Sampling for Aggregation Queries. *ICDE*, 2001.

[CLKB04] Jeffrey Considine, Feifei Li, George Kollios, John Byers. Approximate aggregation techniques for sensor databases. *ICDE*, 2004.

[CR94] Chen, C., Roussopoulos, N. Adaptive Selectivity Estimation Using Query Feedback. *SIGMOD*, 1994.

[FM85] Flajolet, P., Martin, G. Probabilistic Counting Algorithms for Data Base Applications. *JCSS*, 32(2): 182-209.

[G84] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[GAA03] Govindarajan, S., Agarwal, P., Arge, L. CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries. *ICDT*, 2003.

[GGR03] Ganguly, S., Garofalakis, M., Rastogi, R. Processing Set Expressions Over Continuous Update Streams. *SIGMOD*, 2003.

[HHW97] Hellerstein, J., Haas, P., Wang, H. Online Aggregation. *SIGMOD*, 1997.

[JL99] Jurgens, M., Lenz, H. PISA: Performance Models for Index Structures with and without Aggregated Data. *SSDBM*, 1999.

[LM01] Lazaridis, I., Mehrotra, S. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. *SIGMOD*, 2001.

[PGF02] Palmer, C., Gibbons, P., Faloutsos, C. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. *SIGKDD*, 2002.

[PKZT01] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. *SSTD*, 2001.

[PTKZ02] Papadias, D., Tao, Y., Kalnis, P., Zhang, J. Indexing Spatio-Temporal Data Warehouses. *ICDE*, 2002.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M.A. Indexing the Positions of Continuously Moving Objects. *SIGMOD*, 2000.

[SRF87] Sellis, T., Roussopoulos, N., Faloutsos, C. The R+-tree: A Dynamic Index for Multi-Dimensional Objects. *VLDB*, 1987.

[TGIK02] Thaper, N., Guha, S., Indyk, P., Koudas, N. Dynamic Multidimensional Histograms. *SIGMOD*, 2002.

[Tiger] www.census.gov/geo/www/tiger/

[TPS03] Tao, Y., Papadias, D., Sun, J. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *VLDB*, 2003.

[TPZ02] Tao, Y., Papadias, D., Zhang, J. Aggregate Processing of Planar Points. *EDBT*, 2002.

[TSP03] Tao, Y., Sun, J., Papadias, D. Analysis of Predictive Spatio-Temporal Queries. *TODS*, 28(4): 295-336, 2003.

[ZMT+01] Zhang, D., Markowetz, A., Tsotras, V., Gunopulos, D., Seeger, B. Efficient Computation of Temporal Aggregates with Range Predicates. *PODS*, 2001.

[ZTG02] Zhang, D., Tsotras, V., Gunopulos, D. Efficient Aggregation over Objects with Extent *PODS*, 2002.