Querying about the Past, the Present, and the Future in Spatio-Temporal Databases

Jimeng Sun[‡] Dimitris Papadias[†] Yufei Tao[§] Bin Liu[†]

[‡]*Department of Computer Science*
Carnegie Mellon University
Pittsburgh, PA, USA
jimeng@cs.cmu.edu

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

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

Abstract

Moving objects (e.g., vehicles in road networks) continuously generate large amounts of spatio-temporal information in the form of data streams. Efficient management of such streams is a challenging goal due to the highly dynamic nature of the data and the need for fast, on-line computations. In this paper we present a novel approach for approximate query processing about the present, past, or the future in spatio-temporal databases. In particular, we first propose an incrementally updateable, multi-dimensional histogram for present-time queries. Second, we develop a general architecture for maintaining and querying historical data. Third, we implement a stochastic approach for predicting the results of queries that refer to the future. Finally, we experimentally prove the effectiveness and efficiency of our techniques using a realistic simulation.

1. Introduction

Research on spatio-temporal access methods has mainly focused on two aspects: (i) storage and retrieval of historical information, (ii) future prediction. Several indexes [PJT00, KGT+01, TP01], usually based on multi-version or 3-dimensional variations of R-trees, have been proposed towards the first goal, aiming at minimization of the storage requirements and query cost. Methods for future prediction assume that, in addition to the current positions, the velocities of moving objects are known. The goal is to retrieve the objects that satisfy a spatial condition at a future timestamp (or interval) given their present motion vectors (e.g., "based on the current information, find the cars that will be in the city center 10 minutes from now"). The only practical index in this category is the TPR-tree [SJLL00] and its variations [SJ02, TPS03] (also based on R-trees). Other, mostly theoretical, predictive indexes have been proposed in [KGT99] (applicable only to 1D space) and [AAE00] (with good asymptotical performance, but inapplicable in practice due to the large hidden constants).

1.1 Motivation

Despite the large number of methods that focus explicitly on historical information retrieval or future prediction, currently there does not exist a single index that can achieve both goals. Even if such a "universal" structure existed (e.g., a multi-version TPR-tree keeping all previous history of each object), it would be inapplicable for several update-intensive applications, where it is simply infeasible to continuously update the index and at the same time process queries. For instance, an update (i.e., deletion and re-insertion) in a TPR-tree may need to access more than 100 nodes, which means that by the time it terminates its result may already be outdated (due to another update of the same object). Even for a small number of moving objects and a low update rate, the TPR-tree (or any other index) cannot "follow" the fast changes of the underlying data.

Another problem concerns the space requirements. The incoming data are usually in the form of data streams (e.g., through sensors embedded on the road network), which are potentially unbounded in size. Therefore, materializing all data is unrealistic. Furthermore, even if all the data were stored, the size of the tree would render query processing very expensive since any algorithm would have to access at least a complete path from the root to the leaf level. Finally, in several applications, such as traffic control systems, the main focus of query processing is retrieval of approximate summarized information about objects that satisfy some spatio-temporal predicate (e.g., " the number of cars in the city center 10 minutes from now"), as opposed to exact information about the qualifying objects (i.e., the car ids), which may be unavailable, or irrelevant.

1.2 Contributions

Motivated by the above observations, in this paper we propose a comprehensive method that (i) can approximately answer aggregate spatio-temporal queries about the past, the present and the future, (ii) can continuously follow the (typically, very frequent) changes in data distribution, (iii) it is efficient and adjustable to the

burden of the system, i.e., it does not strain the system, especially during periods of intensive workloads, and (iv) reduces the space consumption by summarizing the data.

Our first contribution is an *adaptive multi-dimensional histogram* (AMH), which is used for approximate processing of present-time queries. AMH utilizes idle CPU cycles to perform incremental maintenance. Its precision depends only on the available system resources, and does not deteriorate with time (meaning the AMH does not need to be periodically re-built).

The second contribution is a general architecture, called the *historical synopsis*, for maintaining and querying historical data. In this architecture, when a histogram bucket is updated, the previous version is kept in a main-memory index. If the available memory is exceeded, part of the index containing the least recent information migrates to the disk. As a result, queries referring to the present and recent past (which are more frequent) can be answered without any I/O.

Third we present a stochastic approach, based on *exponential smoothing*, that answers queries about the future using observations from the present and the recent past. Unlike previous approaches for spatio-temporal prediction, this technique does not assume knowledge of velocity vectors, which in most cases is unavailable.

Finally, we experimentally evaluate the proposed techniques under a real-life simulation, where updates and queries are interleaved with histogram maintenance operations. We study the effect of the system workload on the estimation accuracy, and verify the need for approximation methods that adapt to the available recourses.

The rest of the paper is organized as follows. Section 2 reviews previous work related to ours. Section 3 provides the problem definition and an overview of the proposed methods. Section 4 presents the concrete algorithms of AMH, while Section 5 describes the historical synopsis. Section 6 presents the stochastic approach for prediction, and Section 7 contains the experimental evaluation. Finally, Section 8 concludes the paper with directions for future work.

2. Related Work

Because the proposed techniques combine multiple goals, they relate to a number of different research topics. In particular, since the core of our method is AMH, in Section 2.1 we survey related work on multi-dimensional histograms. Section 2.2 presents methods for updating such histograms using query feedback. Section 2.3 discusses other multi-dimensional approximation techniques. Section 2.4 overviews spatio-temporal prediction techniques and their limitations in several applications. Finally, Section 2.5 presents methods for (exact) retrieval of spatio-temporal aggregation information.

2.1 Static multi-dimensional histograms

Histogram construction techniques split the data space into buckets, usually based on the assumption that data within a small region are (almost) uniform. The various methods differ on the partition policy. The *equi-depth* histogram of [MD88] first partitions along one dimension so that each bucket has the same number (i.e., *frequency*) of objects. Then, these buckets are partitioned again along another dimension. *Mhist* [PI97] splits on the most "critical" dimension according to a partitioning metric (e.g., variance). *Minskew* [APR99] partitions the space into a regular grid. Neighboring cells are then grouped into rectangular, non-overlapping buckets by a greedy algorithm that tries to minimize the spatial-skew (i.e., the variance of the cell density in each bucket). *Genhist* [GKTD00] allows overlapping buckets. As with *Minskew*, it starts with a regular grid and creates buckets for the cells with high density. Then it removes a percentage of data from these cells to make the area around them smoother. Finally it decreases the resolution and repeats the process recursively. The *SQ* [AN00] and *Euler* histograms [SAA02], in addition to locations, take into account objects extents, and are more accurate for non-point data.

All the above histograms apply to spatial or multi-dimensional databases, where the data are (almost) static. When the data distribution changes due to updates, the bucket structure may become outdated, and the histogram must be re-built. In several spatio-temporal applications, however, there is simply not enough time to scan the entire data (possibly several times) in order to build the histogram from scratch. Furthermore, the new histogram may already be outdated, due to the changes that occurred during its re-building.

2.2 Query-adaptive multi-dimensional histograms

Query adaptive histograms avoid re-building by using query feedback to refine the buckets. *STGrid* [AC99], based on the heuristic that "buckets with higher frequencies contribute more to the estimation error", splits buckets with high frequencies and merges buckets with low frequencies. Restructuring is performed at certain intervals (a parameter of the histogram) for entire rows or columns, selected by a *split* or a *merge threshold* (also a parameter). Its structure, however, cannot accurately capture arbitrary distributions. *STHoles* [BGC01] alleviates this problem by allowing nesting of buckets. If a query identifies large frequency variance within a bucket, a "hole", i.e., a new bucket, is created inside the original one. This "drilling" operation replaces the splitting mechanism of *STGrid*. *SASH* [LWV03] decomposes the multi-dimensional space into sub-spaces of lower dimensionality. For each subspace a separate histogram is built and maintained using query feedback mechanisms.

Query-adaptive histograms are based on the assumption that the frequency of queries is much higher than the frequency of updates. Thus, after the data distribution changes, there is a sufficient number of queries to "train" (i.e., refine) the histogram. Although the accuracy for these queries will be low, subsequent queries will be estimated using the refined histogram, which gradually becomes very accurate. Clearly, this assumption does not hold for update-intensive applications, where the number of queries for each instance of the data is, typically, very small.

2.3 Other multi-dimensional approximation methods

Several multi-dimensional methods embed the data into another domain and summarize the embedded data in order to provide a compact data representation. The DCT-based histogram [LKC99] maintains a large number of small buckets using discrete cosine transform, which cannot be incrementally maintained. The histogram of [MVW98] extracts the most descriptive wavelet coefficients. Although it can be incrementally maintained [MVW00], its performance degrades with the number of updates, so that eventually it has to be re-built. Another technique [TGIK02] compresses the data distribution into a *multi-dimensional sketch*; when the query arrives, the histogram is extracted from the *sketch*. The main drawback of this approach is that the extracting process is expensive and, therefore, not suitable for on-line queries.

[GMP02] incrementally maintains a small sample set (*backing sample*) of underlying data and histograms are then constructed/maintained by the *backing sample*. However, it requires scanning the entire data to regenerate the *backing sample* when the distribution changes, which is not possible in our case. Furthermore, some other applications of sampling-based techniques [GM98, WAA01] are also questionable, since it is not clear how to select and effectively maintain a representative multi-dimensional sample in the presence of very intensive updates. Finally, all previous methods capture a snapshot of the data and cannot be used for historical information retrieval.

2.4 Spatio-temporal prediction methods

Methods for spatio-temporal prediction estimate the number of objects that will satisfy some spatial condition during a future interval, based on the current location and velocity information. Choi and Chung [CC02] and Tao et al. [TSP03] present probabilistic models for uniform data, which are then applied to non-uniform distributions using some conventional multi-dimensional histogram. An experimental comparison of several techniques can be found in [HKT03].

All existing methods assume linear movement and that the velocities of all objects are known (the same assumptions that hold for the TPR-tree-based indexes [SJLL00, TPS03]). This restricts their applicability in several applications because: (i) movement is not always linear, (ii) even if it is linear, it is not usually known (e.g., mobile devices transmit only their location to the base station), and (iii) even if it is linear and known, it changes so fast that prediction using velocity information is meaningless (e.g., cars moving on road networks).

2.5 Spatio-temporal aggregation methods

Several methods (e.g., [PTKZ02, ZTG02]) focus on spatio-temporal aggregation. Although the target queries are similar to our historical queries, the context is different. In particular, they assume a "conventional" processing framework where every query invokes disk I/Os and returns an exact answer. Here we sacrifice accuracy for efficiency and attempt to answer the most frequent queries using only the main memory.

3. Problem Definition and System Overview

We assume a set of objects moving in the two-dimensional space and a central server collecting information about their locations over time. Update data received by the server contain the previous and the new location of individual objects, but not their velocity or id. Moving objects may issue updates in periodic intervals or based on the deviation from the previous transmitted position. Static objects do not send information to the server; until an object transmits a new location it is assumed to be in the last recorded position.

We adopt a 2D grid that partitions the data space into $w \times w$ (where w is a constant called the *resolution*) regular cells with width $1/w$ on each axis. Each cell c ($1 \leq c \leq w^2$) is associated with a *frequency* $\mathcal{F}_c$, which is the number of objects (at the present time) in its extent. The frequencies of all the cells are maintained up-to-date in memory, and serve as the data of the highest granularity. The resolution is determined by the number of moving objects in the system, as well as the desirable trade-off between accuracy and efficiency.

A spatio-temporal aggregation query $q(q_R,q_T)$ retrieves the number of objects that fall in a rectangle region q_R at a timestamp q_T. If (i) $q_T=0$ (i.e., the present time), q constitutes a *present timestamp* (PT) query; (ii) $q_T<0$, q is a *historical timestamp* (HT) query; and (iii) $q_T>0$, q is a *future timestamp* (FT) query. When the resolution of the grid is high, exhaustive examination of all the cells that are relevant to a query is expensive. AMH (adaptive multi-dimensional histogram) remedies this problem by grouping adjacent cells with similar frequencies into a small number of (rectangular) buckets (similar to other histograms such as *minskew* [APR99] and *genhist* [GKTD00]). Bucket information is updated as new tuples arrive, and bucket extents (i.e., grouping of cells) continuously adapt to the data distribution changes through incremental maintenance performed during the

idle CPU time (i.e., when there are no incoming data or queries). A binary partition tree accelerates bucket maintenance and processing of PT queries.

To support historical retrieval, buckets that are outdated (due to data or extent changes) are inserted in a main-memory index (T) optimized for HT queries. Each bucket is associated with a lifespan denoting the temporal interval during which the bucket information is valid. When the size of the index exceeds the available memory, part of the tree containing the *least recent* buckets is migrated to the disk, such that queries about the recent past can be answered using the memory-resident portion of the index. This migration occurs in blocks (rather than individual buckets) at a time, and incurs minimal I/O overhead. AMH together with index T constitute the historical synopsis, whose general structure is shown in Figure 3.1.

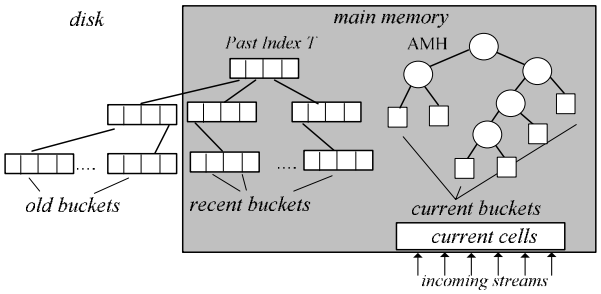


Figure 3.1: Historical synopsis

The historical synopsis is directly used for answering PT and HT queries. On the other hand, FT queries are processed by an exponential smoothing technique, which retrieves information about the present and the recent past, in order to predict the future. Figure 3.2 shows the abstract query processing mechanism and Table 3.1 lists the frequently-used symbols. In the rest of the paper we describe the various components of the architecture, starting with AMH in Section 4.

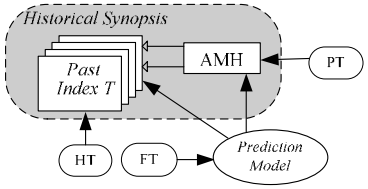


Figure 3.2: Query processing

Symbol	Description
w	resolution
$\mathcal{F}_c$	frequency of cell c
(B) n	(maximum) number of buckets
R_k	bucket extent or area
n_k	number of cells in bucket b_k
f_k	frequency (i.e., number of objects) in bucket b_k
g_k	average squared frequency of bucket b_k
v_k	frequency variance of bucket b_k

Table 3.1: Frequent symbols

4. Adaptive Multi-Dimensional Histogram

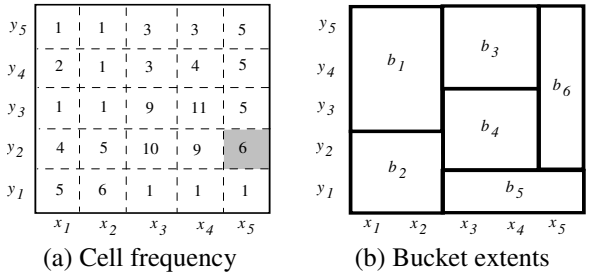
Given a regular $w \times w$ cell partition, AMH generates n rectangular buckets whose edges are aligned with the cell boundaries. For each bucket b_k ($1 \le k \le n$), we denote (i) as n_k the number of cells it covers, (ii) as f_k the average frequency of these cells (i.e., $f_k=(1/n_k)\Sigma_{(\forall \text{ cell in } bk)}\mathcal{F}_c$), and (iii) as v_k their variance (i.e., $v_k=(1/n_k)\Sigma_{(\forall \text{ cell in } bk)}(\mathcal{F}_c-f_k)^2$). AMH aims at minimizing the *weighted variance sum* (*WVS*) of all the buckets, or formally:

$$WVS=\sum_{i=1\sim n}(n_i \cdot v_i) \quad (4\text{-}1)$$

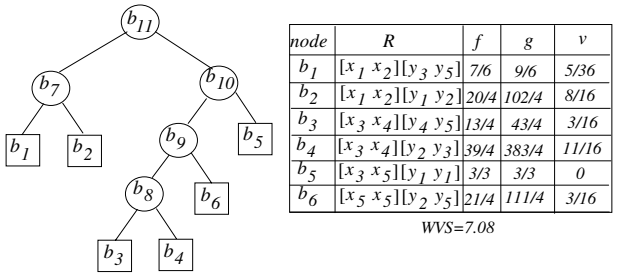
4.1 AMH structure

Each bucket stores: (i) its rectangular extent R (abusing the terminology slightly, we also use R to denote its area), (ii) the average frequency f of all the cells in R, and (iii) the average of "squared" frequency g of these cells: $g=(1/n_k)\sum_{(\forall \text{cell } c \text{ in } R)}(\mathcal{F}_c)^2$. Thus, the number n of cells covered by a bucket can be represented as $R \cdot w^2$, where w is the cell resolution. A bucket's variance v equals $v=g-f^2$, and as a result, *WVS* can be computed using R, f, g of all buckets.

AMH maintains a binary partition tree (BPT), where each leaf node corresponds to a bucket. An intermediate node is associated with a rectangular extent R that encloses the extents of its (two) children. Initially, BPT contains a single leaf node, and the histogram has one bucket covering the entire data space. New buckets (leaf nodes) are created through *bucket splits* (elaborated shortly), but the total number n of buckets never exceeds a system parameter B. As an example, Figure 4.1a shows the frequencies of 25 cells (i.e., w=5) at timestamp 1, Figure 4.1b demonstrates the extents of 6 buckets, and Figure 4.1c illustrates the corresponding BPT.



(a) Cell frequency (b) Bucket extents



node	R	f	g	v
b_1	$[x_1\ x_2][y_3\ y_5]$	7/6	9/6	5/36
b_2	$[x_1\ x_2][y_1\ y_2]$	20/4	102/4	8/16
b_3	$[x_3\ x_4][y_4\ y_5]$	13/4	43/4	3/16
b_4	$[x_3\ x_4][y_2\ y_3]$	39/4	383/4	11/16
b_5	$[x_3\ x_5][y_1\ y_1]$	3/3	3/3	0
b_6	$[x_5\ x_5][y_2\ y_5]$	21/4	111/4	3/16

WVS=7.08

(c) The BPT (d) Bucket information

Figure 4.1: AMH at time 1

Intermediate node b_7, for instance, covers buckets b_1, b_2, implying that they were created by splitting b_7. Figure 4.1d lists the detailed information of each bucket (v is not explicitly stored, but computed from f and g).

4.2 AMH information update

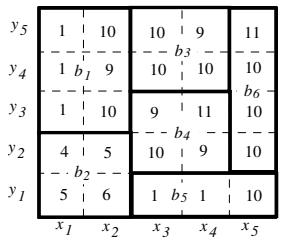
A location update contains the old and the new position of a moving object. When such an update is received, AMH executes an *info-update* process to modify the frequencies of the cells and buckets that cover the corresponding locations. Particularly, a cell can be identified using a hash function, while a bucket is located by following a single path of BPT, guided by the extents of the intermediate nodes. Assume, for example, that the frequency of cell (x_5,y_2) in Figure 4.1a changes from 6 to 10 at time 2, due to concurrent movement of several objects. After modifying the cell frequency, *info-update* obtains the frequency (squared frequency) difference 4 (64) between the new and old values. To complete the update, it identifies bucket b_6 covering the cell (following the path b_{11}, b_{10}, b_9), and updates its f and g as $f_{b_6}= (f_{b_6}\cdot n_{b_6}+4)/n_{b_6}$, $g_{b_6}= (g_{b_6}\cdot n_{b_6}+64)/n_{b_6}$, where n_{b_6} is the number of cells covered by b_6, and f_{b_6} (g_{b_6}) is their average (square) frequency. Assuming that BPT is fairly balanced, the algorithm (shown in Figure 4.2) has average cost log$B$, where B is the maximum number of buckets.

Algorithm Info-update (x, y, d)
/* (x,y) are the cell coordinates, d is the frequency difference */
1. find the cell c that contains (x, y) using a hash function
2. $\Delta f=d$; $\Delta g= (\mathcal{F}_c+d)^2 - \mathcal{F}_c^2$; $\mathcal{F}_c=\mathcal{F}_c+d$
3. descend BPT to find the bucket b that contains (x,y)
4. $f_b=(R_b\cdot w^2\cdot f_b+\Delta f)/(R_b\cdot w^2)$; $g_b=(R_b\cdot w^2\cdot g_b+\Delta g)/(R_b\cdot w^2)$
   // R_b is the extent area of b, w is the cell resolution

End Info-update

Figure 4.2: *info-update* algorithm

Updates may alter the frequency distribution, increasing the frequency variances and the approximation error. Figure 4.3a shows the updated cells at time 2, and Figure 4.3b illustrates the information of the corresponding buckets. Notice that v_{b_1}, v_{b_5} are significantly larger than those in Figure 4.1d, causing considerable increase of *WVS* (from 7.08 in Figure 4.1d to 173.58). To remedy this problem, AMH performs *bucket re-organization* when the CPU is idle (i.e., no data/query is pending), which involves bucket merging and splitting.



(a) Cell frequency

node	R	f	g	v
b_1	$[x_1\ x_2][y_3\ y_5]$	32/6	284/6	680/36
b_2	$[x_1\ x_2][y_1\ y_2]$	20/4	102/4	8/16
b_3	$[x_3\ x_4][y_4\ y_5]$	39/4	381/4	3/16
b_4	$[x_3\ x_4][y_2\ y_3]$	39/4	383/4	11/16
b_5	$[x_3\ x_5][y_1\ y_1]$	12/3	102/3	162/9
b_6	$[x_5\ x_5][y_2\ y_5]$	41/4	421/4	3/16

WVS=173.58

(b) Bucket information

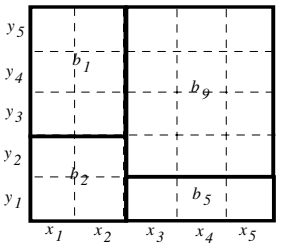
Figure 4.3: AMH at time 2

4.3 AMH bucket merge

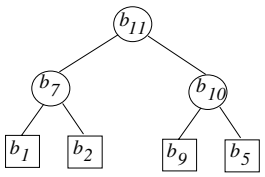
A merge is performed when the number of buckets has reached the maximum value B (assumed to be 6 in these examples), and aims at grouping cells of multiple buckets, whose frequencies have converged since the last re-organization, into a single bucket. Towards this, AMH selects the leaf node of BPT with the lowest *merge penalty* (*MPen*); *MPen* is defined as the increase of *WVS* (see equation 4-1) if the node is merged with its sibling.

Assume, for instance, the cell frequencies in Figure 4.3a, the bucket information in Figure 4.3b, and the tree in Figure 4.1c. Merging bucket b_3 with its sibling b_4 removes both nodes from BPT and makes b_8 a leaf (i.e., a new bucket in the histogram). The change incurs $n_8\cdot v_8-(n_3\cdot v_3+n_4\cdot v_4)$ increase in *WVS* (i.e., the merge penalty for b_3, b_4), where n_i is the number of cells covered by node b_i, and v_i is its frequency variance. A bucket's sibling can sometimes be an intermediate node, e.g., the sibling of bucket b_6 is b_8, in which case *MPen* should be computed as $n_9\cdot v_9-(n_3\cdot v_3+n_4\cdot v_4+n_6\cdot v_6)$. Notice that, merging b_6 with b_8 would reduce the bucket number by 2 (i.e., removing 3 leaf nodes b_3, b_4, b_6, spawning b_9).

The leaf node with the lowest merge penalty can be identified using a single post-order traversal of BPT's intermediate nodes (i.e., each intermediate node is processed after its children). Consider, for example, Figure 4.1c, where node b_7 is the first node processed. We compute the following information (from its children b_1, b_2): (i) the average frequency of all cells in its extent $f_{b_7} = (f_{b_1}\cdot R_{b_1}+f_{b_2}\cdot R_{b_2})/ R_{b_7}$, (ii) the average squared frequency $g_{b_7} = (g_{b_1}\cdot R_{b_1}+ g_{b_2}\cdot R_{b_2})/R_{b_7}$, (iii) the merge penalty of its children $MPen_{b_7} = n_7\cdot v_7-(n_1\cdot v_1+n_2\cdot v_2)$, where $n_i=R_{b_i}\cdot w^2$, $v_i=g_{b_i}-f_{b_i}^2$. Next, similar computations are performed for b_8, b_9, b_{10}, b_{11} (in this order). Note that i) the leaf node b_3 (b_4, b_6) is updated from the corresponding cells; ii) the computation for b_9 (b_{10}) is based on the already-computed information of intermediate node b_8 (b_9). Similarly, the information of b_{11} is computed by b_7 and b_{10}. Finally, the algorithm merges the children of the node that incurs the smallest penalty (in this case b_9). Figures 4.4a and 4.4b illustrate the resulting bucket extents and BPT, respectively. Figure 4.5 shows the pseudo-code of the *bucket-merge* algorithm. Its running time is B, since it visits all nodes of BPT.



(a) Buckets after merging



(b) BPT after merging

Figure 4.4: Example of bucket merge (cont. Figure 4.1)

Algorithm Bucket-Merge
1. *minMPen*=∞; *newleaf*=NULL
2. let *b*=the first intermediate node in the post-order traversal
3. do
4. if (*b* has at least one leaf node)
5. compute f_b, g_b, $MPen_b$ from its children
6. if ($MPen_b$<*minMPen*)
7. *minMPen*=$MPen_b$; *newleaf*=*b;*
8. *b*=next intermediate node in the post-order traversal
9. until (*b*=root)
10. merge the children of *newleaf*

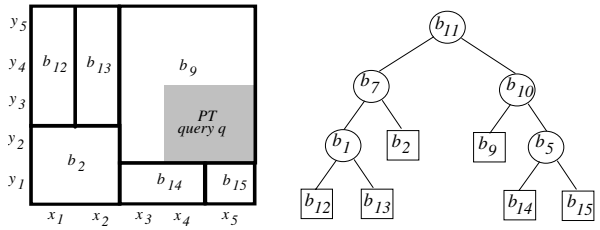
End Bucket-Merge

Figure 4.5: *bucket-merge* algorithm

4.4 AMH bucket split

A bucket split creates refined partitions in regions where buckets have large variance. It improves AMH's approximation accuracy by (i) reducing the bucket frequency variance, and (ii) decreasing the bucket extent. Towards this, the algorithm splits the bucket with the highest *split benefit* (*SBen*), i.e., the maximum *decrease* of *WVS* achieved by splitting this bucket at the "best position". Consider, for example, bucket b_1 in Figure 4.4a. Since a split guarantees that the resulting buckets cover an integer number of cells, there are three possible ways to split b_1 (one on the x-, and two on the y-axes, respectively). The largest reduction of *WVS* is achieved by splitting b_1 on the x-dimension, resulting in buckets b_{12}, b_{13} (Figure 4.6a) and split benefit $n_1 \cdot v_1 - (n_{12} \cdot v_{12} + n_{13} \cdot v_{13})$. The split benefits of the other buckets are also computed and the one with the largest *SBen* (in this case b_1) is split (at its best split position).

The *bucket-split* algorithm repeats the split process until (i) new data/queries arrive, (ii) the number of buckets reaches the maximum threshold *B* or (iii) there does not exist any bucket with positive split benefit, implying that the current bucket number is sufficient to model the cell frequency distribution. Continuing the example, *bucket-split* computes *SBen* for the new buckets (b_{12}, b_{13}) and splits b_5 (whose benefit is currently the largest) into b_{14}, b_{15}. Since the bucket number has reached *B* (=6), the algorithm terminates. Figures 4.6a, 4.6b illustrate the final bucket extents and BPT, respectively.



(a) Final Buckets (b) BPT after splitting

Figure 4.6: After splitting b_1, b_5 (cont. Figure 4.4)

Figure 4.7 summarizes the *bucket-split* algorithm. Each bucket re-organization involves at most one bucket merging (i.e., if the number of buckets equals *B*), and a small number of bucket splits. After its termination, the process starts again if the CPU is still idle. Compared to traditional histograms requiring expensive total re-building, AMH deploys repetitive, simple, interruptible, partial re-organizations, and, therefore, it is ideal for dynamic spatio-temporal environments, where updates/queries may occur in (unpredictable) bursts.

Algorithm Bucket-Split
1. for every bucket
2. compute its split benefit and best split position
3. sort all split benefits in descending order into a list L_{sb}
4. while (bucket number n <*B* and no new data/query and the first bucket in L_{sb} has positive split benefit)
5. remove the first bucket *b* from L_{sb}
6. split it into b_1, b_2
7. obtain split benefits and best split positions for b_1, b_2
8. insert b_1, b_2 into the appropriate positions in L_{sb}
9. return

End Bucket-Split

Figure 4.7: *bucket-split* algorithm

4.5 AMH PT query processing

A present-time query is answered by examining all buckets whose extents intersect the query region q_R. Such buckets are located by traversing BPT and following the nodes that overlap q_R. For example, the query *q* in Figure 4.6a intersects only bucket b_9, which can be reached by following the intermediate nodes b_{11}, b_{10}. For each intersecting bucket b_k, we compute the overlap area R_{intr} between its extent and q_R, and obtain a partial result $f_k \cdot R_{intr} \cdot w^2$ (note that $R_{intr} \cdot w^2$ equals the number of cells in b_k that are covered by q_R). Then, the final result is computed as the sum of the partial results of all intersecting buckets, without accessing the underlying cells.

5. Historical Synopsis

In this section, we discuss HT query processing using the historical synopsis that consists of (i) a dynamic histogram maintaining the currently valid buckets (e.g., AMH), and (ii) a past index *T* storing the obsolete buckets.

5.1 General architecture

A bucket in AMH *dies* if (i) the frequency in any of its cells is updated, or (ii) its extent changes due to a split or merge. In case (i), a new bucket with the same extent, but different frequency statistics (i.e., *f* and *g*) is created. In case (ii) new bucket(s) are created due to merge or split. Each bucket contains a *lifespan* [l_s, l_e), where l_s (l_e) is the time of creation (death). All the buckets in AMH are *alive*, and their l_e equals "now". Dead buckets are inserted into index *T* and their cell content is discarded.

In Figure 4.1, for example, all the buckets o_1,...,o_6 are alive and have lifespans [1,now). Then, in Figures 4.3 buckets b_1, b_3, b_5, b_6 die at time 2 because some cells in their extents change frequencies at this timestamp. This

produces four dead buckets with lifespans [1,2) (which are inserted into T), and creates four live ones with lifespans [2,now). At the next timestamp 3, buckets b_1, b_3, b_4, b_5, b_6 die (yielding four buckets with lifespans [2,3], and one b_4 with lifespan [1,3)) since b_3, b_4, b_6 are merged (Figure 4.4) and b_1, b_5 are split (Figure 4.6). Buckets b_9, b_{12}, b_{13}, b_{14}, b_{15} that result from these operations are alive with lifespans [3, now). As shown in Figure 5.1, after these operations, T contains 9 dead buckets and AMH 6 live ones.

	node	R	f	g	v	lifespan	
dead stored in T	b_1	$[x_1\ x_2][y_3\ y_5]$	7/6	9/6	5/36	[1,2)	*
	b_3	$[x_3\ x_4][y_4\ y_5]$	13/4	43/4	3/16	[1,2)	
	b_5	$[x_3\ x_5][y_1\ y_1]$	3/3	3/3	0	[1,2)	
	b_6	$[x_5\ x_5][y_2\ y_5]$	21/4	111/4	3/16	[1,2)	
	b_3	$[x_3\ x_4][y_4\ y_5]$	39/4	381/4	3/16	[2,3)	
	b_4	$[x_3\ x_4][y_2\ y_3]$	39/4	383/4	11/16	[1,3)	*
	b_6	$[x_5\ x_5][y_2\ y_5]$	41/4	421/4	3/16	[2,3)	
alive stored in AMS	b_1	$[x_1\ x_2][y_3\ y_5]$	32/6	284/6	680/36	[2,3)	
	b_5	$[x_3\ x_5][y_1\ y_1]$	12/3	102/3	162/9	[2,3)	
	b_2	$[x_1\ x_2][y_1\ y_2]$	20/4	102/4	8/16	[1,now)	*
	b_9	$[x_3\ x_5][y_3\ y_5]$	119/12	1185/12	59/12	[3,now)	
	b_{12}	$[x_1\ x_1][y_3\ y_5]$	3/3	3/3	0	[3,now)	
	b_{13}	$[x_2\ x_2][y_3\ y_5]$	29/3	281/3	2/3	[3,now)	
	b_{14}	$[x_3\ x_4][y_1\ y_1]$	2/2	2/2	0	[3,now)	
	b_{15}	$[x_5\ x_5][y_1\ y_1]$	10/1	100/1	0	[3,now)	

Figure 5.1: Buckets at time 3 (cf. Figures 4.1, 4.3, 4.6)

The size of T grows continuously with time and eventually exceeds the available memory, in which case part of it *migrates* to the disk. The migration is performed in blocks, meaning that we simply move pages of equal sizes from the memory to the disk. Further, the disk accesses are sequential except for, possibly, the first page transferred. Accordingly, pointers to these pages are modified to the corresponding disk addresses (from their original main-memory addresses). Each page of T that is currently in memory is assigned a *migration rank*, such that pages are migrated in ascending order of their ranks. A separate main-memory structure (e.g., priority queue) can be maintained to select the page with the lowest rank efficiently. In our implementation, the migration rank of a page is set to the latest ending timestamp l_e of all its entries' lifespans. In general, pages with low ranks should contain old buckets, and be less likely accessed by queries. Finally, pages that are already on the disk may be eventually (physically) erased or moved to tertiary storage, when they become completely obsolete.

5.2 Implementation using a packed B-tree

The past index T should support the following operations efficiently: (i) insertion of a dead bucket, and (ii) selection of buckets intersecting a HT query. Our first implementation adopts a *packed B-tree*, which indexes the ending time l_e of the buckets' lifespans. Specifically, each leaf entry stores all information of a bucket b (i.e., R_b, f_b, g_b, and its life span), while an intermediate entry stores a lifespan $[l_s, l_e)$, which encloses those of the leaf entries in its child node. All the nodes in the B-tree are full, except for the right-most one at each level, which does not have a minimum utilization requirement. Particularly, we refer to the right-most leaf node as the *active leaf*, for which a special pointer is maintained. Since buckets are inserted into T in ascending order of their l_e, each insertion simply appends the new bucket into the active leaf, and terminates if the leaf incurs no overflow. Otherwise, a new active leaf is created with the most recently inserted bucket, and this change propagates to the parent levels. The amortized insertion cost is constant. Figure 5.2 illustrates the corresponding packed B-tree for the dead buckets in Figure 5.1, assuming a capacity of 3 entries per node.

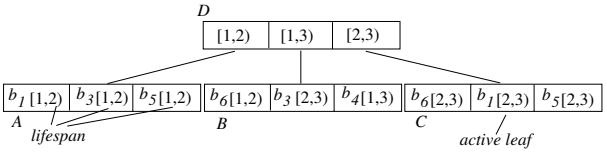


Figure 5.2: The packed B-tree

In some cases, HT may have to access AMH, if there are some buckets that were created at the query timestamp, but still remain alive. Consider, for example, the query with region q_R=$[x_2,x_3][y_2,y_3]$ and timestamp q_T =1 on the buckets of Figure 5.1 (the corresponding AMH is shown in Figure 4.6). The query must visit all the buckets whose extents (lifespans) intersect q_R at time 1, namely, those marked with "*" in Figure 5.1. The final result is the sum of the partial results of all such buckets. The partial result from bucket b_2 (which is still alive in AMH), is obtained by a PT query as discussed in Section 4.5. To locate the intersecting buckets (i.e., b_1 and b_4) in the B-tree, the algorithm accesses the nodes whose lifespans contain q_T (=1), i.e., nodes D, A, B in Figure 5.2. At the leaf level the partial results of qualifying buckets are obtained in the same way as PT queries (i.e., by computing the size of the intersection area with q_R). The packed B-tree implementation contains a simple and very efficient insertion algorithm, but processes HT queries using only temporal pruning. In the next section, we describe an alternative structure that takes advantage of spatial conditions to accelerate query processing.

5.3 Implementation using a 3D R-tree

Our second implementation is based on a main-memory adaptation of the 3D R*-tree [BKSS90]. In this structure, each intermediate entry contains a 3D box which encloses the extents and lifespans of all the buckets in its sub-tree. Given a HT q (which can also be regarded as a 2D rectangle defined by q_R at time q_T), we only need to visit those nodes whose 3D boxes intersect that of q. Compared to the packed B-tree, the 3D R-tree implementation achieves lower query cost since it utilizes both spatial and temporal conditions to prune the search space. On the other hand, it incurs higher update overhead

due to the complex insertion operations of the R-tree (which are required to achieve spatio-temporal locality).

Both indexes use the same migration mechanism discussed in Section 5.1, i.e., when the main memory capacity is reached, the least recent blocks of the index are transferred to the disk. Thus, their update cost difference refers to CPU overhead. In summary, the B-tree method is preferable for very high streaming rates, while the R-tree for applications with heavy query workload. The tradeoff between update and query efficiency is explored in the experimental evaluation.

6. Prediction Model

As discussed in Section 2.4, spatio-temporal prediction based on current locations and velocities has limited applicability in the majority of real applications. Consider, for instance, a traffic supervision scenario, where cars moving on a road network periodically transmit their information. The assumption that the velocity will remain constant between the current and the query time q_T is not realistic, since the cars that reach the end of the road segment on which they travel, must update their velocity (i.e., they must turn or stop). As shown in the experimental evaluation, for typical traffic simulation settings, up to 95% of the cars may update their velocities per timestamp, implying that velocity-based prediction is meaningless even for the next timestamp.

The motivation of our model is that, although the individual object velocities may change abruptly, the overall data distribution varies gradually (and slowly) with time, due to the continuity of movement. Furthermore, since we keep past and present data, we can use this information to identify the trend of movement and estimate the result of a future query. Consider, for instance, that the current time is 0, and we want to process the query $q(q_R,1)$, that predicts the number of objects in q_R at the next timestamp. The output of q can be represented as a function of the result of a PT and a set of recent HT queries with the same region q_R.

Exponential smoothing, a well-known forecasting method in time series analysis [G85], is based on the same reasoning. According to this method, the estimated result S'_1 of the query $(q_R,1)$ can be modeled as a time series such that:

$$S'_1 = \alpha S_o + \alpha(1-\alpha)S_{-1} + \alpha(1-\alpha)^2 S_{-2} + \ldots + \alpha(1-\alpha)^n S_{-n}$$

where (i) S_0 is the actual result of the PT query $(q_R,0)$ (at the current time), (i) S_{-i} is the actual result of the HT query $(q_R,-i)$ at the i^{th} previous timestamp, (iii) n determines the length of previous history that will be used for prediction, and (iv) a is the smoothing parameter in the range (0,1). The formula is based on the idea that recent timestamps are more important for prediction than older ones. The relative weight is adjusted by the value of a: as it approaches 1, the significance of the most recent timestamps increases.

The model is applicable in our case because the changes of the query result are smooth, or more specifically, the series has a locally constant mean which shows some "drift" over time. If we want to predict the result at a future timestamp i>1, (i.e., no actual values for S_1 to S_{i-1} are available), we have to use a step-by-step technique, i.e., first compute S'_1, then use this value to estimate S'_2 and so on. In general, the value of S'_i can be represented by the following recurrence:

$$S'_i = \alpha S_o + (1-\alpha)S'_{i-1}$$

where S'_{i-1} is the predicted value for the query at future timestamp i-1. Figure 6.1 illustrates the pseudo-code for the prediction algorithm based on the above discussion. In our implementation we fix a to 0.25 and n to 6. Although, these parameters can be set on-the-fly for each individual query (i.e., the values that give the best estimation for the current and recent timestamps, since the actual results are known), we chose to fix global values for all queries in order to avoid the overhead of tuning. In particular, it has been suggested [H86] that a = 0.2~0.3 gives accurate results for a variety of problems, while n = 6 represents a good trade-off between accuracy and query cost (the longer we go into the past the higher the cost).

Algorithm Prediction (*qr*,*qt*)
/* *qr*: query region; *qt*: prediction timestamp;
n: the least recent timestamp taken into account for prediction,
α : smoothing parameter */
1. S_0 =PT(*qr*,0) // *compute result of corresponding PT query*
2. for *i*=1 to *n*
3. S_{-i}= HT(*qr*,-*i*) // *result of HT at* i^{th} *previous timestamp*
4. $t = -n$; $S_{current} = S_{-n}$;
5. while ($t \le qt$)
6. if ($t \le 0$) // *present or past*
7. $S_{next} = \alpha S_t + (1-\alpha) S_{current}$
8. else // *future*
9. $S_{next} = \alpha S_0 + (1-\alpha) S_{current}$
10. $S_{current} = S_{next}$
11. $t = t+1$
12. return $S_{current}$ // *predicted value at qt*
End Prediction

Figure 6.1: Algorithm for prediction

7. Experimental Evaluation

This section evaluates the proposed methods, using a Pentium IV 1.8GHz CPU and 256 MBytes of main memory. Section 7.1 describes the experimental settings, Section 7.2 tunes the size of AMH and Section 7.3 evaluates its ability to follow the data distribution effectively. Section 7.4 compares AMH with the conventional, total rebuilding approach. Section 7.5 evaluates the approximation error and compares exponential smoothing with velocity-based prediction. Finally, Section 7.6 studies the effect of update intensity on the performance of the historical synopsis and the approximation error.

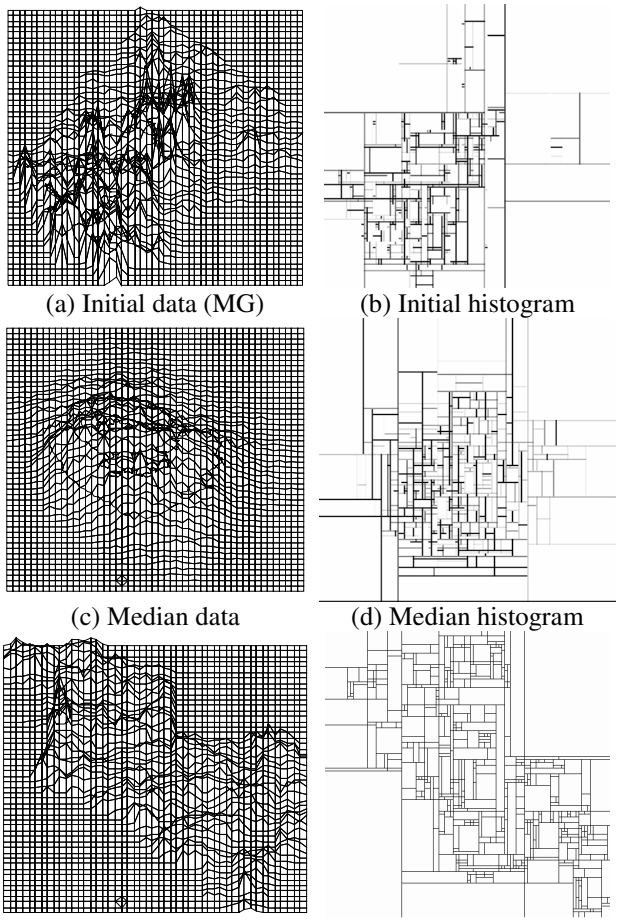
7.1 Settings

The first dataset, denoted as *spatial*, simulates a scenario involving 50k mobile objects. The initial positions of the objects are sampled from the real dataset MG, and their destinations are randomly selected from another real dataset LB (both available from www.census.gov/geo/www/tiger/). Each object moves from the initial to the destination point following a straight line and reports a fixed number of location changes to the server. After it reaches its destination, it selects a new random destination from the initial dataset and this process is repeated for a total of 5 "round trips". Although the endpoints of each trip are different, the overall distribution after the completion of a trip is either MG or LB.

The second dataset, denoted as *road*, is created by the spatio-temporal generator of [B02]. The input of the generator is the road map of Oldenburg (a city in Germany), containing 6105 nodes and 7035 edges, and the output is a set of point objects moving on this network. Each point is represented by its locations at successive timestamps. The datasets are created by setting the parameters (see [B02] for their detailed semantics) of the generator as follows. There are 6 classes of objects (e.g., cars, pedestrians), so that each class moves with a particular speed range. The total number of objects (in all 6 classes) at any timestamp is 500,000 (the generator decides the number of objects per class according to some function beyond users' control). At each timestamp 25k objects disappear (i.e., they reach their destinations), while 25k new ones appear at random nodes. Given these settings, on average, 95% of the objects issue updates per timestamp (i.e., they reach the end of the road segment that they are on and they turn or stop). The total number of updates in both *road* and *spatial* datasets is 2.5M.

The server maintains a regular partition of the space in 100×100 cells (as discussed in Sections 3 and 4). When a location update is received from an object, the server (i) decreases (by 1) the frequency of the cell corresponding to the old position, and (ii) increases the frequency of the cell at the new location. The first column of Figure 7.1 shows the density of objects per cell at the initial, median and ending timestamp in a single trip, for the *spatial* dataset. The second column illustrates the corresponding bucket structure, which continuously follows the data distribution. Figure 7.2a shows the road network of Oldenburg and 7.2b the density of the cells at the initial timestamp. Although the velocity updates for *road* are much more frequent than the *spatial* dataset, the overall distribution does not change significantly over time because movement is constrained to the network.

Each query has two parameters: the side length q_L and the query timestamp q_T. In all cases, the query extent is a square with side length q_L, uniformly distributed in space. For HT (FT), queries q_T is uniformly distributed in the past (future) 10 timestamps. An FT query is processed by issuing 6 HT and 1 PT query with the same extent. The *error rate* is measured as: $|act-app|/|act|$, where *act* (*app*) equals the actual (approximate) answer.

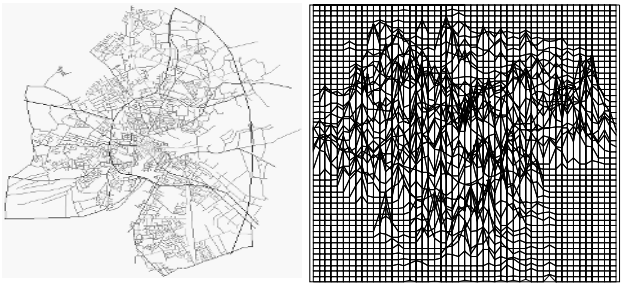


(a) Initial data (MG) (b) Initial histogram

(c) Median data (d) Median histogram

(e) Final data (LB) (f) Final histogram

Figure 7.1: *Spatial* dataset



(a) Road Network (b) Data on road

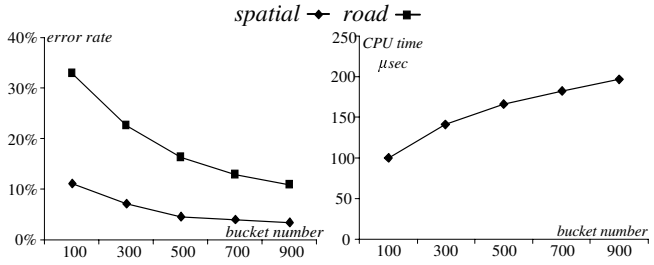
Figure 7.2: *Road* dataset

7.2 Setting the number of buckets

Unlike other multi-dimensional histograms (e.g., *genhist*) that require tuning for several parameters, AMH involves a single parameter B (maximum number of buckets). In order to tune B, we issue 25k PT queries, distributed in the whole history with length q_L=6% of the spatial universe. We allow AMH to re-organize every 500 incoming tuples. This re-organization is limited to five

merge operations, each followed by one or more splits (depending on whether intermediate BPT nodes are merged).

As expected (and shown in Figure 7.3a), the average error rate decreases with B. The error is higher for *road* because the local uniformity assumption (within each bucket) does not accurately capture the real object distribution. The same problem exists for all multi-dimensional histograms (see Section 2) that decompose the space based on local uniformity. To the best of our knowledge, there does not exist any histogram specifically targeting network data.



(a) Error rate vs. B (b) Re-organization cost vs. B

Figure 7.3: Tuning B (q_L=6%, q_T=0)

In order to evaluate the overhead, we use the same settings and measure the total CPU-time spent on re-organization operations. Figure 7.3b shows the average CPU cost per location update as a function of B for the *spatial* dataset (the CPU cost for *road* is about the same and omitted). The overhead increases linearly with the maximum number of buckets, since both merge and split operations examine all buckets. Based on these experiments, we set B=500, since it provides good accuracy (average error less than 5% and 20%, for *spatial* and *road*, respectively) with reasonable overhead.

7.3 Robustness with time

The goal of this experiment is to evaluate the robustness of AMH with time. We fix B=500, apply the same settings as in Figure 7.3, and issue 25k PT queries, uniformly distributed in history (q_L=6%). Figure 7.4 shows the error rate as a function of the total number of updates for the *spatial* dataset. The periodic behaviour of the error is because the initial and final distributions are more skewed than the intermediate ones (see Figure 7.1).

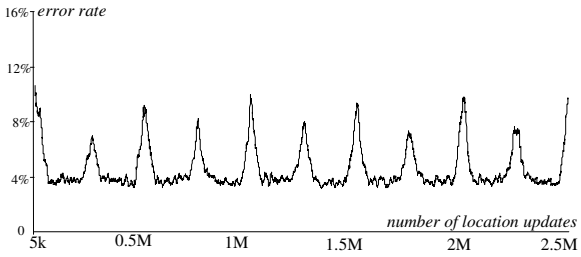


Figure 7.4: Error vs. number of updates (*spatial*)

Figure 7.5 repeats the experiment for the *road* dataset, which does not show the periodic effect of *spatial*, because the data distribution is constrained by the underlying road network. In both cases the overall error rate remains stable as time evolves, indicating that AMH captures the dynamic data distribution very well through successive re-organizations.

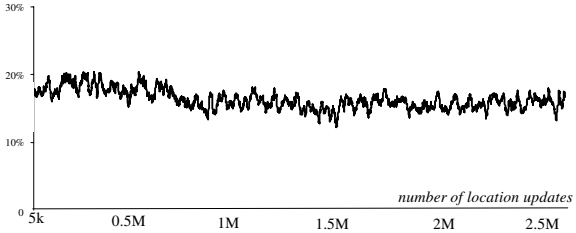
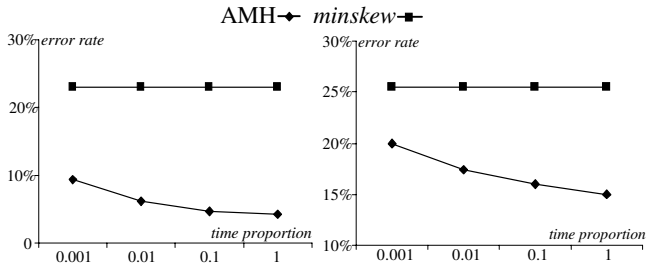


Figure 7.5: Error vs. number of updates (*road*)

7.4 Comparison with conventional histograms

Now, we compare AMH with *minskew* [APR99] (a common benchmark in the histogram literature [GKTD00, WAA01]), constructed using the same 100×100 regular partition as AMH. Since *minskew* is a static histogram, we re-build it every 50k location updates, measure the cost (about 0.5 seconds) for each re-building and assign a percentage tp of this cost to re-organization operations in AMH. For instance, if tp=100%, we use the same amount of time for both AMH re-organizations and *minskew* re-building. The difference is that the re-organization operations of AMH are uniformly distributed among the 50k location updates.

Figure 7.6a (b) illustrates the average error rate of *spatial* and *road* datasets over 25k uniformly distributed (in history) PT queries as a function of tp. AMH is much more accurate than *minskew*, even when consuming 1/1000 of the CPU resources, because it continuously adapts to the data distribution using cheap operations. On the other hand, *minskew* is accurate only during the short period after the re-building and its bucket structure soon becomes outdated.



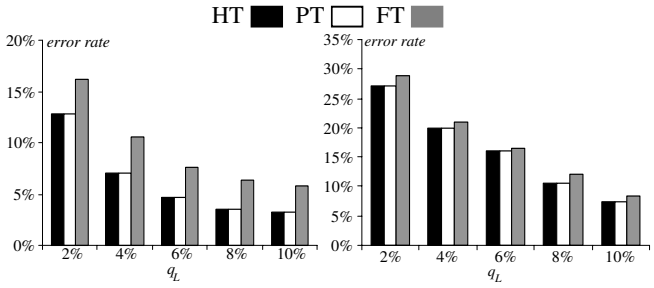
(a) *spatial* (b) *road*

Figure 7.6: Error rate vs. tp (q_L = 6%, q_T=0, B=500)

7.5 Evaluation of query and prediction error

In order to evaluate the error of different query types, we issue 3 query sets, each containing 25k queries of the same type (HT, PT, or FT). Figure 7.7 shows the error rate as the function of the query length q_L (from 2% to 10% of the spatial universe), respectively. The error of FT queries (average of predictions for the next 1-10

timestamps) is higher because it also includes the inaccuracy of prediction. The results for the *road* dataset are less accurate, because (as discussed in Figure 7.3) the local uniformity assumption within each bucket does not capture well the actual data distribution.

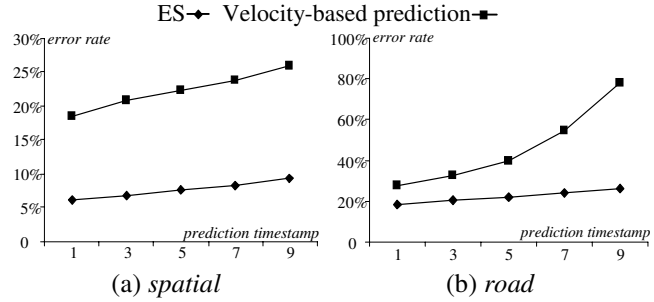


(a) *spatial* (b) *road*

Figure 7.7: Query error comparison (B=500)

Next, we compare our prediction method, ES for exponential smoothing, with the state-of-the-art model [TSP03] that uses only information about current location and velocity. In this model, non-uniform data are handled by a 4D *minskew* histogram (2D for spatial, 2D for velocity). The velocity of an object at timestamp i is determined by its location at timestamps i and i+1. The histogram is constructed using 100×100 and 6×6 regular partitions on the spatial and velocity dimensions, respectively. The number of buckets is set to 2000 (more buckets are needed due to the 4D space) and rebuilding occurs every timestamp (so that the error is due to the model and not the histogram deterioration). In other words, both the space consumption and the CPU overhead are much higher than AMH.

Figure 7.8 shows the error rate for 25k uniformly distributed FT queries as a function of q_T (i.e., future prediction time), fixing q_L to 6% of the spatial universe. Although in *spatial* the movement is linear, velocity-based prediction is less accurate than ES (Figure 7.8a). Compared to the results reported in [TSP03], the error of the method is higher, because it now includes the updates that occur between the current and the query time (in [TSP03] it is assumed that there are no such updates). For the road dataset (Figure 7.8b), where updates are much more intense, the quality of prediction deteriorates fast with q_T, confirming our observations (in Section 6) that velocity-based prediction is meaningless in such cases.



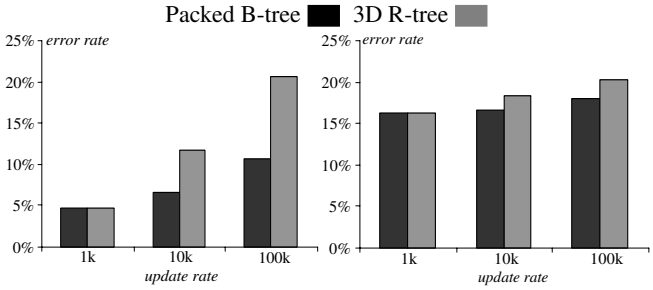
(a) *spatial* (b) *road*

Figure 7.8: Error rate vs. q_T (q_L = 6%)

7.6 The effect of update intensity

Having demonstrated the accuracy of the proposed methods, we now test the historical synopsis architecture in real-time environments involving intensive updates. In this scenario we assume that updates have higher priority and, therefore, re-organization is delayed during periods of high workload. In order to evaluate the behaviour of the two implementations (packed B-tree and 3D R-tree), we issue 25k HT, PT and FT queries uniformly distributed in history. Figure 7.9 shows the error as a function of the update rate, which varies from 1k to 100k updates per second. For 1k updates the accuracy of both implementations is the same, implying that they can both follow the data distribution effectively. In the other cases, however, the 3D R-tree incurs higher error, because it consumes a significant amount of time for insertion operations (of the dead buckets in the tree). On the other hand, the efficient insertion algorithms of the packed B-tree, allow the system to devote more CPU time to histogram re-organization, leading to lower error. Note that the error rate of *spatial* increases faster than that of the *road* dataset, because the data distribution varies significantly with time, implying that if the buckets do not get re-organized, they will soon become outdated.

However, if a large percentage of the workload consists of queries, then the 3D R-tree synopsis is preferable, since it utilizes the spatial condition to effectively prune the search space. Figure 7.10 shows the average cost of all query types for the two implementations. In both cases, PT queries are processed by AMH and have the same performance. HT (and, consequently, FT) queries incur almost half the cost in R-trees. For typical settings, we found that if the query/update ratio in the workload exceeds 3/7, the 3D R-tree is better (i.e., it allows for more re-organization operations).



(a) *spatial* (b) *road*

Figure 7.9: Error rate vs. update rate (q_L = 6%, B=500)

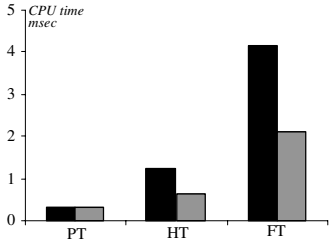


Figure 7.10: Query cost comparison (q_L = 6%, B =500)

8. Conclusions

Existing spatio-temporal access methods have several disadvantages that severely limit their applicability: (i) they focus explicitly on either historical information retrieval, or future prediction; (ii) they are very expensive to update and query on-line, since both types of operations incur a large number of disk accesses; (iii) their prediction assumptions are unrealistic for most practical scenarios. Motivated by these problems, we present a comprehensive approach for processing queries that refer to any time in history. Instead of keeping detailed information about individual objects, the proposed architecture maintains an incremental multi-dimensional histogram, which answers present-time queries. Outdated buckets are stored in a main-memory index, in order to answer queries about the recent past without any I/O operations. The "oldest" parts of the index migrate to the disk in blocks, so that the total I/O cost of updates is minimized. Finally, future queries are answered by a stochastic method that uses the recent history to predict the future, without any assumptions about velocity. Extensive experiments confirm the effectiveness of our techniques under realistic settings.

Acknowledgements

This work was supported by grants HKUST 6081/01E and HKUST 6197/02E from Hong Kong RGC.

References

[AAE00] Agarwal, P., Arge, L., Erickson, J. Indexing Moving Points. *PODS*, 2000.

[AC99] Aboulnaga, A., Chaudhuri, S. Self-tuning Histograms: Building Histograms Without Looking at Data. *SIGMOD*, 1999.

[AN00] Aboulnaga, A., Naughton, J. Accurate Estimation of the Cost of Spatial Selections. *ICDE*, 2000.

[APR99] Acharya, S., Poosala, V., Ramaswamy, S. Selectivity Estimation in Spatial Databases. *SIGMOD*, 1999.

[B02] Brinkhoff, T. A Framework for Generating Network-Based Moving Objects. *GeoInformatica* 6(2), 153-180, 2002.

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

[BGC01] Bruno, N., Chaudhuri, S., Gravano, L. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD*, 2001.

[CC02] Choi, Y., Chung, C. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. *SIGMOD*, 2002.

[G85] Gardner, E. Exponential Smoothing: The State of the Art. *Journal of Forecasting* (4): 1-28, 1985.

[GKTD00] Gunopulos, D., Kollios, G., Tsotras, V., Domeniconi, C. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. *SIGMOD*, 2000.

[GM98] Gibbons, P., Matias, Y. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. *SIGMOD*, 1998.

[GMP02] Gibbons, P., Matias, Y., Poosala, V. Fast Incremental Maintenance of Approximate Histograms. *TODS* 27(3):261-298, 2002.

[H86] Hunter, J. The Exponentially Weighted Moving Average, *Journal of Quality Technology,* 18(4), 203-207, 1986.

[HKT03] Hadjieleftheriou, M., Kollios, G., Tsotras, V. Performance Evaluation of Spatio-temporal Selectivity Estimation Techniques. *SSDBM*, 2003.

[HKTG02] Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D. Efficient Indexing of Spatiotemporal Objects. *EDBT*, 2002.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. *PODS*, 1999.

[LKC99] Lee, J., Kim, D., Chung, C.. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. *SIGMOD*, 1999.

[LWV03] Lim, L., Wang, M., Vitter, J. SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads. *VLDB*, 2003.

[MD88] Muralikrishna, M., DeWitt, D. Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. *SIGMOD*, 1988.

[MVW00] Matias,Y.,Vitter,J,Wang,M., Dynamic Maintenance of Wavelet-Based Histograms. *VLDB*, 2000.

[MVW98] Matias, Y., Vitter, J., Wang, M. Wavelet-Based Histograms for Selectivity Estimation. *SIGMOD*, 1998.

[PI97] Poosala, V., Ioannidis, Y. Selectivity Estimation Without the Attribute Value Independence Assumption. *VLDB*, 1997.

[PJT00] Pfoser, D., Jensen, C., Theodoridis, Y. Novel Approaches in Query Processing for Moving Object Trajectories. *VLDB*, 2000.

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

[SAA02] Sun, C., Agrawal, D., Abbadi, A. Exploring Spatial Datasets with Histograms. *ICDE*, 2002.

[SJ02] Saltenis, S., Jensen, C. Indexing of Moving Objects for Location-Based Services. *ICDE*, 2002.

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

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

[TP01] Tao, Y., Papadias, D. The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *VLDB*, 2001.

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

[TSP03] Tao, Y., Sun, J. Papadias, D. Selectivity Estimation for Predictive Spatio-Temporal Queries. *ICDE*, 2003.

[WAA01] Wu, Y., Agrawal, D., Abbadi, A. Using the Golden Rule of Sampling for Query Estimation. *SIGMOD*, 2001.

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