Approximate Temporal Aggregation

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Christos Faloutsos
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
christos@cs.cmu.edu

Abstract

Temporal aggregate queries retrieve summarized information about records with time-evolving attributes. Existing approaches have at least one of the following shortcomings: (i) they incur large space requirements, (ii) they have high processing cost and (iii) they are based on complex structures, which are not available in commercial systems. In this paper we solve these problems by approximation techniques with bounded error. We propose two methods: the first one is based on multi-version B-trees and has logarithmic worst-case query cost, while the second technique uses off-the-shelf B- and R-trees, and achieves the same performance in the expected case. We experimentally demonstrate that the proposed methods consume an order of magnitude less space than their competitors and are significantly faster, even for cases that the permissible error bound is very small.

1. Introduction

Temporal databases have received considerable attention due to the large number of applications that require efficient management of data with time-evolving attributes. In the related systems, records are typically represented as two-dimensional intervals in the so-called *key-time* space. Consider, for example, a telecom company that preserves the following information about phone calls made by its customers: (i) the starting/ending time of each call, and (ii) its cost (in dollars). Figure 1.1 illustrates the interval representation of 6 calls, where the key of each interval (i.e., its projection on the vertical axis) denotes its cost, while the horizontal projection corresponds to its duration or, following the common terminology in the literature, its *lifespan*. For example, the lifespan of *f* is the open interval [1,5) (we represent this using different colors for its end points). A data interval is *alive* during its lifespan, and *dead* outside of it.

While the majority of research (see [ST99] for a survey) in temporal databases aims at retrieving information about individual objects that satisfy certain (temporal and non-temporal) predicates, the motivation of this work is that many applications require only aggregate results and can often accept approximate answers with small bounded error. Given a key range *qk* and an interval *qt*, a *temporal count* query retrieves the total number of data intervals that are alive during *qt* with keys in range *qk*. For example, the shaded rectangle in Figure 1.1 represents the query "return the number of phone calls in period *qt*=[6,8], with costs in *qk*=[1.75,3]". Equivalently, the goal is to count the number of intervals intersecting the query rectangle (i.e., in our example the result is 2). An alternative is the *temporal sum query*, which, assuming that each data interval is associated with a *weight*, retrieves the sum of the weights of the qualifying records. For instance, if the database also stores the number of persons involved in each call (i.e., it is possible to have conference calls with more than two users), then a temporal sum query returns the total number of persons in all calls that qualify *qk* and *qt*.

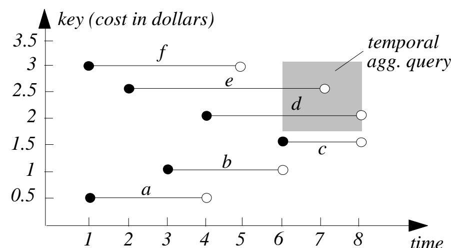


Figure 1.1: Interval representation of temporal data

- **Motivation**

Existing temporal aggregation techniques focus on exact query processing. The best existing structure, the MVSB-tree [ZMT+01], consumes $O(N/B \cdot \log_B N/B)$ space and answers any query with $O(\log_B N/B)$ node accesses, where N is the number of intervals in the database and B is the number of intervals that fit in one disk page. As demonstrated in our experiments, in spite of its good query cost, the size of this structure is much larger than that of the database (due to its sub-optimal space complexity), seriously hampering its usefulness for (typically very voluminous) practical datasets.

Furthermore, the MVSB-tree (as well as most other methods for this problem) is currently inapplicable in practice since it is based on some complex, specialized indexes that are not implemented in any commercial database product. A practical solution should leverage only tools that are "off-the-shelf" from the market and at the same time provide excellent query performance.

- **Contribution**

This paper presents the first work on approximate temporal aggregate processing. Specifically, for count queries, the goal is to provide answers guaranteed to deviate from the actual ones within threshold $\varepsilon \cdot N$, where ε is an arbitrary positive constant smaller than 1 (i.e., the maximum error is specified as a percentage of the dataset cardinality). We address this problem from both theoretical and practical perspectives: (i) using the multi-version B-tree, we show that any query can be answered in $O(\log_B N/B)$ cost and linear space $O(N/B)$; (ii) for practical scenarios, we reduce the problem to *constrained nearest neighbor search* [FSAA01], which can be answered by an R-tree (available in latest products from Informix and Oracle) probabilistically in $O(\log_B N/B)$ time and $O(N/B)$ space. The methodology also leads to the same bounds for approximate sum queries, where the maximum approximation error is defined as $\varepsilon \cdot \sum_{i=1}^{N} w_i$ and w_i ($1 \le i \le N$) is the weight of the i-th data interval.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem of approximate temporal aggregation and presents our methods. Section 4 contains an extensive experimental evaluation that confirms the applicability of our approaches, while Section 5 concludes the paper with directions for future work.

2. Related work

Section 2.1 first describes the multi-version B-tree, since it is the basic structure of previous methods and constitutes a part of the proposed solutions. Then, Section 2.2 discusses existing approaches on multi-dimensional aggregate processing, covering both temporal and non-temporal databases. Finally, Section 2.3 briefly reviews algorithms for constrained nearest neighbor search using R-trees.

2.1 The multi-version B-tree (MVB-tree)

The MVB-tree [BGO+96, VV97] aims at processing *timestamp key range* queries in temporal databases. Such a query retrieves all data intervals that are alive at a timestamp *qt*, and whose keys fall in a range *qk*. For instance, the query with *qt*=6 and *qk*=[1.75,3] will return intervals *d* and *e* in Figure 1.1.

The MVB-tree can be regarded as a space-efficient scheme for storing multiple (logical) B-trees. Each entry has the form <*key*, t_{st}, t_{ed}>, where t_{st} denotes the starting time of a data interval, and t_{ed} denotes the ending time (i.e., $[t_{st}, t_{ed})$ is the entry's lifespan). For leaf entries, *key* is the key of an interval (e.g., the cost of a call in Figure 1.1), while for an intermediate entry *e*, *key* equals the minimum key of the leaf entries in its subtree that are alive in the lifespan of *e*. The inclusion of a data interval in the tree involves two separate operations at its starting and ending timestamps respectively (all the operations are performed chronologically). Specifically, if the current processing time is *t*, intervals starting at this time are inserted with their t_{st} set to *t* and t_{ed} to "*", indicating that their ending time is temporarily unknown (such entries are said to be *alive*). Similarly, intervals ending at this time have their t_{ed} changed (from *) to *t* (they *die* at *t*). Figure 2.1 illustrates an example of an MVB-tree with both alive and dead entries.

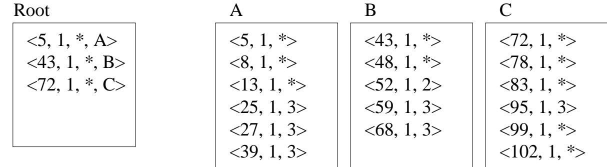


Figure 2.1: A MVB-tree example

For each timestamp *t* and each node, except for the roots, it is required that either none, or at least $B \cdot P_{version}$ entries are alive at *t*, where $P_{version}$ is a tree parameter and *B* the disk page size (for the following examples $P_{version}$=1/3 and *B*=6). This *weak version condition* ensures that entries alive at the same timestamp are mostly grouped together. A *weak version underflow* occurs if this condition is violated (e.g., due to deletion at the current processing time). A *block overflow* occurs when an entry is inserted into a full node, in which case a *version split* is performed. To be specific, all the alive entries of the node are copied to a new node, with their t_{st} modified to the current time. The value of t_{ed} of these entries in the original node is also set to the insertion time[1]. The insertion of <28,4,*> into node *A* at timestamp 4 (in the tree of Figure 2.1) will cause node *A* to overflow. As shown in Figure 2.2, a new node *D* is created to store the alive entries of *A*, and *A* dies meaning that it will not be modified any more in the future. A new entry <5,4,*,*D*> (pointing to the new node) is inserted into the root node. When the root generates a version split, the new node of the split becomes the root of another logical tree.

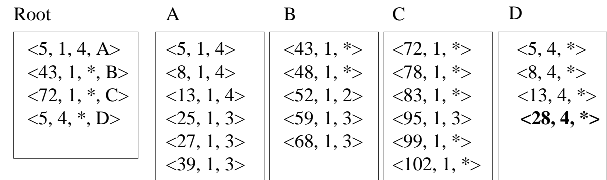


Figure 2.2: Version split in the MVB-tree

In some cases, the new node may be almost full after a version split so that a small number of insertions would cause its overflow again. On the other hand, if it contains too few entries, a small number of deletions will cause its

---

[1] In practice this step can be avoided since the deletion time is implied by the entry in the parent node.

underflow. To avoid these problems, after a version split the number of entries in the new node must be in the range $[B{\cdot}P_{svu}, B{\cdot}P_{svo}]$ (P_{svu} and P_{svo} are tree parameters). A *strong version overflow (underflow)* occurs when the number of entries exceeds $B{\cdot}P_{svo}$ (becomes lower than $B{\cdot}P_{svu}$). A strong version overflow is handled by a *key split,* which is a version-independent split according to the keys of the entries in the node, and is processed in the same way as the B-tree. The strong version underflow is similar to the weak version underflow; the only difference is that the former happens after a version split, while the latter occurs when the weak version condition is violated after a deletion. In both cases, a merge is attempted with the copy of a sibling node, using only its alive entries. If the merged node strong version overflows, a key split is performed. In [VV97], the merging process is improved to reduce the tree size.

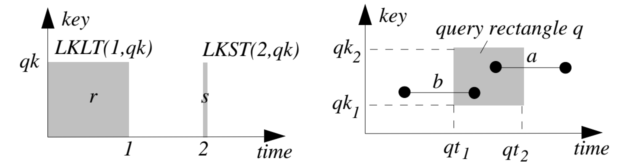
As shown in [BGO+96], given N data intervals, the MVB-tree consumes $O(N/B)$ space, and answers timestamp range queries with $O(\log_B N/B + K/B)$ I/Os, where K is the number of intervals retrieved, i.e., both space consumption and query cost are optimal. Furthermore, the MVB-tree can also optimally process the *timestamp search* query $TS(qt,qk)$, which retrieves the interval, alive at qt, with the largest key $\leq q_k$. For instance, in Figure 1.1, $TS(6,3)$ returns interval e. The following theorem summarizes the performance bounds for MVB-trees, which we utilize in Section 3.2.

Theorem 2.1: Given N data intervals, a MVB-tree consumes $O(N/B)$ space and answers any timestamp search query in $O(\log_B N/B)$ time. ■

2.2 Aggregate processing techniques

Zhang et al. [ZMT+01] address exact retrieval of temporal aggregation. According to their terminology, given a value qk and a timestamp qt, a data interval satisfies the *less-key-less-time condition* $LKLT(qt,qk)$ if its key (starting timestamp) is no larger than qk (qt). As shown in Figure 2.3a (qt=1), such intervals intersect the rectangular region r, whose main diagonal starts at the origin of the axes and ends at point (qt,qk). Similarly, an interval satisfies the *less-key-single-time condition* $LKST(qt,qk)$, if it is alive at timestamp qt and its key is no larger than qk. In Figure 2.3a (qt=2), the qualifying intervals intersect the line segment s defined by points (2,0) and $(2,qk)$. The corresponding *count LKLT query* returns the number of intervals satisfying the *LKLT* condition. The *sum LKLT query* returns the sum of their weights. Without ambiguity, in the sequel we use the notation $LKLT(qt,qk)$ to denote both the *LKLT* condition, and the result of a *LKLT* count or sum query. The semantics of $LKST(qt,qk)$ are similar.

A temporal count/sum query with key range $[qk_1, qk_2]$ and interval $[qt_1,qt_2]$ can be reduced to 4 *LKLT* and 2 *LKST* queries, or equivalently, the result equals $LKLT(qt_1,qk_1) + LKLT(qt_2,qk_2) - LKLT(qt_2,qk_1) - LKLT(qt_1,qk_2) + LKST(qt_1,qk_2) - LKST(qt_1,qk_1)$. Figure 2.3b illustrates the two types of intervals that qualify a counting query q (shaded region): (i) those (e.g., a) whose starting points fall in q, and (ii) those (e.g., b) that cross the left edge of q. The number of intervals of type (i) is given by $LKLT(qt_1,qk_1) + LKLT(qt_2,qk_2) - LKLT(qt_2,qk_1) - LKLT(qt_1,qk_2)$, while that of type (ii) by $LKST(qt_1,qk_2) - LKST(qt_1,qk_1)$. Based on this reduction, Zhang et al. [ZMT+01] propose the MVSB-tree, which is the multi-version counterpart[2] of the SB-tree [YW01]. For N data intervals, the MVSB-tree answers a temporal count/sum query optimally in $O(\log_B N/B)$ time, but consumes sub-optimal $O(N/B{\cdot}\log_B N/B)$ space.



(a) *LKLT* and *LKST* (b) 2 qualifying interval types

Figure 2.3: Reduction of temporal aggregate queries

The aP-tree [TPZ02b] aims at the aggregate processing of planar points. Although the method could also be applied, with appropriate transformations, for temporal aggregation, it also requires $O(N/B{\cdot}\log_B N/B)$ space. Another aggregation structure for multi-dimensional points, the CRB-tree [GAA03], consumes linear space, however, under the very restrictive assumption, that most of the tree is stored in sequential pages (which can only hold for static data). Furthermore, its extension for temporal aggregation is unclear. In [ZTG02], Zhang et al. develop two versions of the ECDF-B-tree for aggregate processing on rectangular objects (also applicable to intervals) with different space-query time tradeoffs. Specifically, the first version consumes $O(N/B{\cdot}\log_B N/B)$ space and answers a query in $O(B{\cdot}\log_B^2 N/B)$ time, while the corresponding complexities of the second version are $O(N{\cdot}\log_B N/B)$ (for space) and $O(\log_B^2 N/B)$ (for query cost). These bounds are worse than those of the MVSB-tree (due to the higher applicability of the ECDF-B-tree).

The above techniques rely on specialized index structures not available in any commercial product. The most "practical" method in the literature is the aggregate R-tree[3] (aR-tee) [PKZT01], which augments the traditional R-tree with aggregate information in the intermediate

[2] As discussed in [BGO+96], the algorithms introduced in Section 2.1 can be applied to obtain the multi-version counterpart of any "ephemeral" structure.

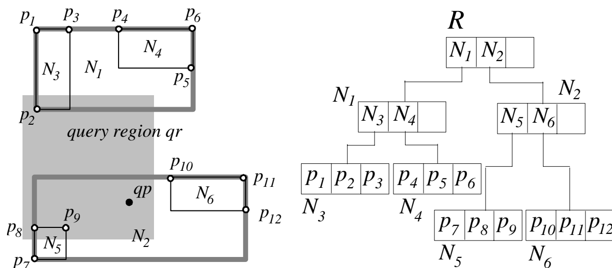
[3] Strictly speaking the aR-tree is not available in current DBMS either. Here we categorize it as "practical" because the necessary changes (with respect to the R-tree) are small. Our solutions, however, are not based on aR-trees.

entries. The main idea is that if the MBR of an intermediate entry is totally contained in the query region, its sub-tree is not visited; instead, the aggregate number stored with the entry is retrieved directly. As shown in [PKZT01], aR-trees achieve significant speedup compared to conventional R-tree for large query windows. The aR-tree can be used for temporal aggregation by treating each interval as an MBR with zero extent on the y axis.

Aggregate processing has also been considered in other applications, e.g., Papadias et al [PTKZ02] deal with spatio-temporal aggregation, while Zhang et al [ZGTS03] consider data streams. These approaches are specific to the targeted problems and cannot be applied in our case. Finally, approximate query answering in non-temporal databases 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 temporal (historical) data (the only histograms with a temporal aspect focus on spatio-temporal prediction [TSP03]).

2.3 Constrained nearest neighbor search

Given a set of multi-dimensional points, a query region *qr* and a point *qp*, a *constrained nearest neighbor query* retrieves the data point that is closest to *qp* among all points that fall in *qr*. Figure 2.4 illustrates 12 points and the MBRs of the corresponding R-tree. To answer the query with *qr* and *qp* shown in Figure 2.4a, the algorithm of [FSAA01] first retrieves the root of the R-tree, and (because the MBRs of both N_1 and N_2 intersect *qr*) visits the subtrees of the root entries in ascending order of their minimum distances to the query point. In this example, N_2 has zero distance to *qr* and its contents N_5, N_6 are fetched. Since the MBR of N_6 does not intersect *qr*, only N_5 is retrieved and point p_9 becomes the current nearest neighbor (NN). Then, the algorithm backtracks to the root and terminates after discovering that the distance from *qp* to N_1 is longer than that between *qp* and p_9, which becomes the final result. A conventional (unconstrained) nearest neighbor query would return p_{10} as the result for *qp*.



(a) Points and node extents (b) Corresponding R-tree

Figure 2.4: A constrained NN query and its processing

The performance of NN queries has been very well studied. The cost models of [B00, KPF01] indicate that the expected query cost depends only on the (i) *fractal dimension* [FK94] of the dataset, and (ii) the height of the R-tree. Specifically, they show that the cost is bounded by $O((B+1)^f/B \cdot \log_B N/B)$, where B is the disk page size, and f the fractal dimension. Since for 2D points $f \leq 2$, we have $O((B+1)^f/B \cdot \log_B N/B) = O((B+1) \cdot \log_B N/B) = O(\log_B N/B)$, leading to Theorem 2.2.

Theorem 2.2: Given N 2D data points, an R-tree answers a (constrained or not) nearest neighbor query in expected $O(\log_B N/B)$ time. ■

In Section 3.3 we illustrate the connection between nearest neighbor search and temporal aggregation, and show how it leads to practical solutions with good performance in the expected case.

3. Approximate Temporal Aggregation

Section 3.1 formally defines approximate temporal aggregate queries and illustrates how they can be reduced to approximate *LKST* queries (following the terminology of [ZMT+01]). Sections 3.2 and 3.3 solve *LKST* count queries using techniques based on MVB-trees and combination of R- and B-trees, respectively. Section 3.4 extends the solutions to temporal sum.

3.1 Problem definition and basic reductions

We consider the popular *transaction time database* model [ST99], where records can only be appended to the database chronologically. When a new data interval begins/ends (i.e., in the context of Figure 1.1, a call starts/finishes), the database sets its starting/ending timestamp to the current time. Intervals already written to the database are never removed. Further, we assume the typical *word-wise machine*, where each value occupies the whole memory word[4]. Our goal is to design access methods for approximate processing of temporal count and sum queries.

Problem 3.1: Given N data intervals with arbitrary lengths[5] and weights $w_1, w_2, \ldots, w_N$, an *approximate temporal count* (*sum*) query returns a value, which deviates from the precise result by less than $\varepsilon \cdot N$ ($\varepsilon \cdot \sum_{i=1}^{N} w_i$), where ε is a constant (referred to as the *approximation ratio* in the sequel) in [0,1]. ■

---

[4] In the alternative *bit-wise* machine [GAA03], any integer v is represented by exactly $\log_2 v$ bits, so that multiple integers may be compressed into a single word. Structures implemented in bit-wise machines usually consume less space, at the expense of significant implementation overhead.

[5] The problem is substantially simpler if the intervals are of the same length, or their lengths are smaller than a constant. In both cases the problem can be reduced to point aggregation by enlarging the query region accordingly.

Notice that the error bound is related to the maximum possible query result. Specifically, the "largest" count query retrieves all intervals (i.e., the maximum result is N), and similarly, the highest value returned by a sum query is $\sum_{i=1}^{N} w_i$. Consider, for instance, that N=10^4 and ε=0.001; then, the output of any temporal count query may deviate by less than 10 from the actual result. On the other hand, if N becomes 10^5, the maximum error is less than 100 for the same ε. Motivated by the reduction of [ZMT+01] (see Section 2.2), we consider the approximate versions of *LKLT* and *LKST* queries defined as follows:

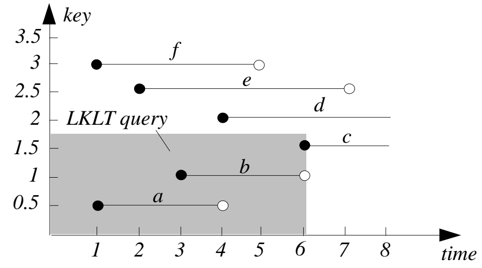
Problem 3.2: Given N data intervals with arbitrary lengths, an *approximate LKLT*(qt,qk) count (sum) query returns a value that deviates from the actual result, i.e., the number of intervals (or the sum of their weights) satisfying condition *LKLT*(qt,qk), by no more than $\varepsilon\cdot N$ ($\varepsilon\cdot\sum_{i=1}^{N} w_i$), where ε is the approximation ratio in the range [0,1]. The *approximate LKST* query can be defined in the same way. ■

The approximate version of the reduction (from temporal count/sum to *LKLT* and *LKST*) is more complex than its exact counterpart. As shown in Section 3.2, our methods for approximate *LKLT* and *LKST* queries always return values smaller than the actual result. Given this fact, we prove the following lemma.

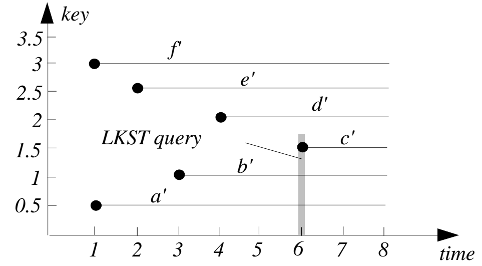
Lemma 3.1: An approximate count/sum query with ratio ε, can be reduced to 4 approximate *LKLT* and 2 *LKST* queries, all with approximation ratio ε/3.

Proof: Let the count/sum query have key range q_K=[qk_1,qk_2] and interval q_T=[qt_1,qt_2]. Then, as mentioned in Section 2.2, the actual result act = *LKLT*(qt_1,qk_1) + *LKLT*(qt_2,qk_2) − *LKLT*(qt_2,qk_1) − *LKLT*(qt_1,qk_2) + *LKST*(qt_1,qk_2) − *LKST*(qt_1,qk_1). Let the actual values of the 6 terms in this equation be v_1, v_2, …, v_6, respectively (i.e., $act = v_1 + v_2 - v_3 - v_4 + v_5 + v_6$). Accordingly, let v_1', v_2', …, v_6' denote the approximate results returned by our method with approximation ratio ε/3. The approximate result for the original query is computed as $est = v_1' + v_2' - v_3' - v_4' + v_5' + v_6'$. Given that $v_i' \le v_i \le v_i' + \varepsilon/3$, for $1 \le i \le 6$, it is easy to obtain that $-\varepsilon \le est - act \le \varepsilon$. ■

Recall that *LKLT* queries are used by [ZMT+01] to obtain the intervals (of type (i) in Figure 2.3b) whose starting points fall in the query region. It turns out that each *LKLT* can be further reduced to a *LKST* query by a simple transformation. Consider query *LKLT*(6,1.75) in Figure 3.1a, returning the number (3) of intervals starting in the shaded region. Figure 3.1b demonstrates the transformed query *LKST*(6,1.75) and data, where the ending time of each transformed interval is fixed to the last timestamp (i.e., 8) of the history, and its key and starting time are the same as those of the original interval. Observe that *LKST*(6,1.75) (on the transformed data in Figure 3.1b) has the same result as *LKLT*(6,1.75) (in Figure 3.1a). Note that the transformed intervals[6] are not actually materialized to the hard disk (in fact, as elaborated shortly, our technique only requires storing a fraction of the database).



(a) *LKLT* query



(b) The transformed data and *LKST* query

Figure 3.1: Reduction from *LKLT* to *LKST*

The following lemma summarizes the above discussion.

Lemma 3.2: An approximate count/sum query with approximation ratio ε can be reduced to 6 approximate *LKST* queries with approximation ratio ε/3.

Proof: This lemma results from Lemma 3.1 and the reduction from *LKLT* to *LKST*. ■

It is worth mentioning that the *LKST* query, in addition to being the core of temporal count/sum queries, is also an important stand-alone operation in practice. Focusing first on *LKST* count, in the next section we present a solution for its approximate processing based on the MVB-tree. In Section 3.3, we present an alternative method that adopts only B- and R-trees (hence it is readily applicable in commercial systems). The same methodology is applied to sum queries in Section 3.4.

3.2 Approximate *LKST* count using MVB-trees

We define the *rank* of an interval e at timestamp t (t is within the lifespan of e) as the output of *LKST*(t,$e.key$), namely, the number of intervals (i.e., including e itself) alive at t with keys≤$e.key$. Obviously, the *rank* of an interval may change at different timestamps. In Figure 3.2a, for example, the *rank* of f equals 2, 3, 4 during intervals [1,2), [2,3), and [3,5), respectively.

[6] A similar transformation (but for a different problem) is applied in [TPZ02b].

Now let us consider query $q=LKST(6,3)$ in Figure 3.2a, whose actual result is 3, since intervals *e*, *d*, *c* are alive at timestamp 6 and have keys ≤ 3. Our solution is based on the observation that the result equals the largest *rank* among those of the qualifying intervals (in this case the *rank* of *e*). In order to obtain this rank efficiently, we "clip" every data interval into several smaller ones, each corresponding to the period when its *rank* remains fixed. Figure 3.2b illustrates the *clipped* intervals for the data of Figure 3.2a. For example, *f* is partitioned into 3 intervals with lifespans [1,2), [2,3), [3,5), and *ranks* 2, 3, 4 respectively, while *d* generates a single interval because its *rank* remains 2 throughout its lifespan. The clipped intervals are indexed by a MVB-tree, where each leaf entry contains in addition to *key*, t_{st}, t_{ed}, also the *rank* of the interval during $[t_{st}, t_{ed})$. The $LKST(6,3)$ query can now be answered by the *timestamp search* query $TS(6,3)$ (see Section 2.1), which finds the qualifying interval with the largest key (i.e., the clipped interval of *e* with lifespan [3,7)). The *rank* (3) of this interval constitutes the (actual) result.

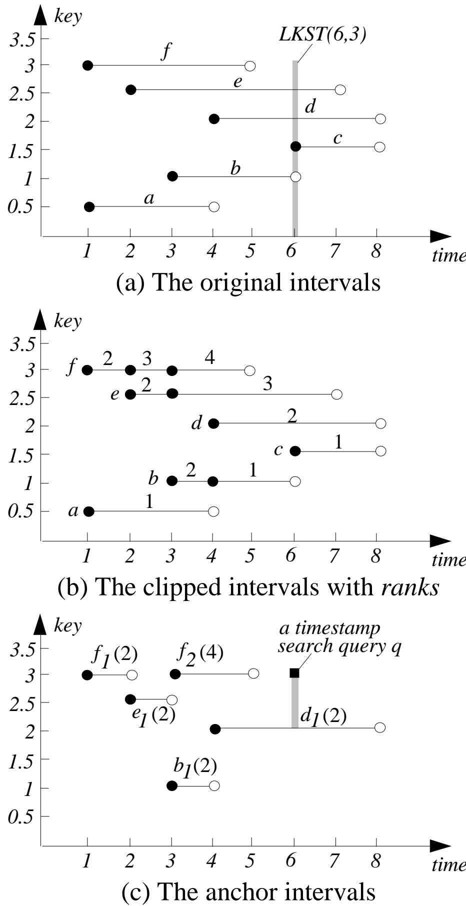


Figure 3.2: Generating anchor intervals based on *ranks*

The problem of this method is that in the worst case, each interval will be clipped into $O(N)$ smaller ones, incurring $O(N^2)$ space consumption (which is higher than the MVSB-tree). Fortunately, the space consumption can be significantly reduced for approximate processing. Assume, for example, that we want the absolute error to be less than 2 (i.e., $\varepsilon=2/6=1/3$). This implies that we only need to store even *ranks* because even if the actual result is an odd number, we can retrieve the next (lower) *rank* (which must be even)[7]. Therefore, the difference between the approximate and actual answers is at most 1. This is illustrated in Figure 3.2c that shows the intervals with even *ranks*, as well as, the timestamp search query $TS(6,3)$, which now retrieves interval d_1, with *rank* 2 as the approximate result (recall that the actual result is 3).

In the general case, where ε is an arbitrary number in [0,1], we need to keep only the clipped intervals whose *rank* is a multiple of $\lceil \varepsilon \cdot N \rceil$, so that any approximate output deviates at most $\lceil \varepsilon \cdot N \rceil - 1$ from the actual result. In the sequel, we refer to these intervals as the *anchor intervals*. As shown in the following lemma, the number of anchor intervals is $O(N)$, i.e., we reduce the space consumption from quadratic to linear.

Lemma 3.3: Given N data intervals, the total number of anchor intervals, whose *ranks* are multiples of $\lceil \varepsilon \cdot N \rceil$, is $O(N)$.

Proof: First observe that at any timestamp, there can be at most $\lfloor 1/\varepsilon \rfloor = O(1)$ anchor intervals (i.e., since $O(N)$ intervals are alive at each timestamp, and the *ranks* of consecutive anchor intervals differ by $\lceil \varepsilon \cdot N \rceil$). On the other hand, a new anchor interval can be created only at timestamps when an interval starts/ends, and the number of such timestamps is at most $2N$. Therefore, the total number of anchor intervals is at most $2N \cdot \lfloor 1/\varepsilon \rfloor = O(N)$. ■

It is worth mentioning that, although the number of anchor intervals (i.e., after discarding those whose ranks are not multiples of $\lceil \varepsilon \cdot N \rceil$) produced by a particular data interval may still be $O(N)$, Lemma 3.3 indicates that there can be only $O(1)$ such (data) intervals. As mentioned earlier, we index the $O(N)$ anchor intervals using an MVB-tree, which, as stated in Theorem 2.1, consumes $O(N/B)$ space, and answers a time search query (and hence the original *LKST* query) in $O(\log_B N/B)$ I/Os.

Next we discuss the algorithm for efficiently generating the anchor intervals. To facilitate illustration, we refer to the starting/ending timestamps of all the intervals collectively as *event timestamps*, which are first sorted and processed in ascending order. In practice, however, this sorting step can be avoided as data intervals start/end chronologically. Processing a starting (ending) event timestamp t involves inserting (removing) the corresponding data interval into (from) a B-tree, using its

[7] Applying the same methodology, we can easily obtain approximate results that are always larger.

key as the index key. In other words, the B-tree always maintains the set of $O(N)$ data intervals alive at t. On the other hand, we store the set of anchor intervals alive at t in an *anchor list* L_{anchor}, using $O(1)$ space. At the end of each timestamp (i.e., the next event timestamp is larger than t), we empty L_{anchor}, and select, from the B-tree (as described in detail shortly), the set of $O(1)$ intervals, whose *ranks* at t are multiples of $\lceil \varepsilon \cdot N \rceil$, and place them in L_{anchor}.

It remains to clarify how to select, from a B-tree, the record with a specific *rank* r, i.e., the record whose key is the r-th smallest among all records in the tree. Towards this, we augment each intermediate entry of the B-tree with an additional value, specifying the number of data records in its subtree. Figure 3.3 shows an example with 10 records (e.g., the aggregate 5 in parenthesis of the first entry of G denotes that there are 5 entries in its subtree). Then, a record with any *rank* can be found by accessing $O(h)$ nodes, where h is the height of the tree. For instance, to find the record with *rank* r=8, we first retrieve the root G, and since the aggregate in the first entry is 5<r, we follow the second entry and reach F. Now we search for the record with *rank* r=8−5=3 in the subtree of F; thus, we follow the first entry of F (since its aggregate 3≥r) and reach C, where we simply select the 3rd record (i.e., 8).

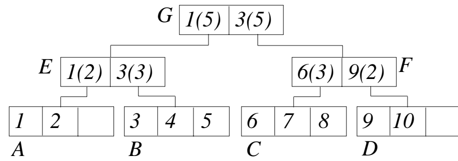


Figure 3.3: A B-tree with aggregate information

We are now ready to analyze the execution time of the algorithm for producing the anchor intervals. Processing each timestamp t involves (i) inserting/removing K intervals (where K is the number of intervals starting or ending at t) into the B-tree, (ii) performing $O(1)$ queries to find the anchor intervals at t, and (iii) updating the anchor list. Since the number of intervals alive at the same timestamp is $O(N)$, step (i) requires $O(K \cdot \log_B N/B)$ time, step (ii) $O(\log_B N/B)$ time, and step (i) $O(1)$ time. Thus, the total overhead of the algorithm is bounded by inserting/removing N intervals, namely, $O(N \cdot \log_B N/B)$. It is worth mentioning that our algorithm is “append-only” (since we process the intervals in ascending order of the event timestamps), and thus is suitable for incremental maintenance in transaction time databases. The following theorem summarizes the above discussion.

Theorem 3.1: Given N data intervals, any approximate *LKST* query can be solved using a MVB-tree in $O(\log_B N/B)$ time and $O(N/B)$ space. The MVB-tree can be updated in amortized $O(\log_B N/B)$ time per interval. ■

Since, by Lemma 3.2, a count query is reduced to 6 *LKST* queries, Theorem 3.1 also describes the bounds for approximate count queries. Note that although approximate results of *LKST* queries are always lower than the actual values, the final output of an approximate count (or sum) query may be actually larger than the precise answer (but within the bounds), because the partial results of some *LKST* queries are subtracted.

If the *LKST* query is issued as a stand-alone operation, Theorem 3.1 still holds even if we further decrease the error upper bound as follows. Specifically, given *LKST*(qt,qk), we require that the approximate result deviates from the actual value by less than $\varepsilon \cdot N_t$ (instead of $\varepsilon \cdot N$), where N_t is the number of data intervals alive at time qt. This can be achieved using the same algorithm for generating the anchor intervals, except that the intervals maintained at time qt are those whose ranks are multiples of $\lceil \varepsilon \cdot N_t \rceil$. Further, note that this upper bound is indeed the lowest that can be guaranteed by our methodology, given that we are allowed to use only linear space.

3.3 Approximate *LKST* count using B-, R-trees

To enable the application of R-trees, we convert approximate *LKST* queries to constrained nearest neighbor search on a set of 2D points. Towards this, we first obtain, as discussed in the previous section, a set of $O(N)$ anchor intervals (i.e., the clipped intervals whose ranks are multiples of $\lceil \varepsilon \cdot N \rceil$). Then, each interval with key k and lifespan $[t_s,t_e)$ generates a set of *anchor points* with the same key k as follows: (i) the starting point of the interval is the first anchor point, and (ii) at every timestamp $t \in [t_s,t_e)$ when another anchor interval starts or ends, a new anchor point is created. Figure 3.4a illustrates the anchor intervals of Figure 3.2b. Every interval, except for f_2 and d_1, produces a single anchor point (its starting point). On the other hand, f_2 generates an additional point at the timestamp when d_1 starts and b_1 ends. Similarly, d_1 produces an additional point at the time when f_2 ends. Figure 3.4b shows the set of anchor points. Each point is tagged with the *rank* of the corresponding anchor interval.

As with anchor interval generation, anchor points can also be obtained using plane-sweep, by maintaining the anchor list storing the $O(1)$ anchor intervals alive at the sweeping timestamp. The difference is that, at each event timestamp (i.e., when an anchor interval starts/ends), a new anchor point is created for each interval in the anchor list. Figure 3.5 illustrates the pseudocode for this process. As shown in Lemma 3.4, the number of anchor points is at the order of the dataset size.

Lemma 3.4: Given N data intervals, the total number of anchor points is $O(N)$.

Proof: As stated in the proof of Lemma 3.3, there are $O(N)$ event timestamps at which we need to generate anchor points. The number of such anchor points at each event timestamp is of the same order $O(1)$ as that of the anchor intervals at that time, which completes the proof. ■

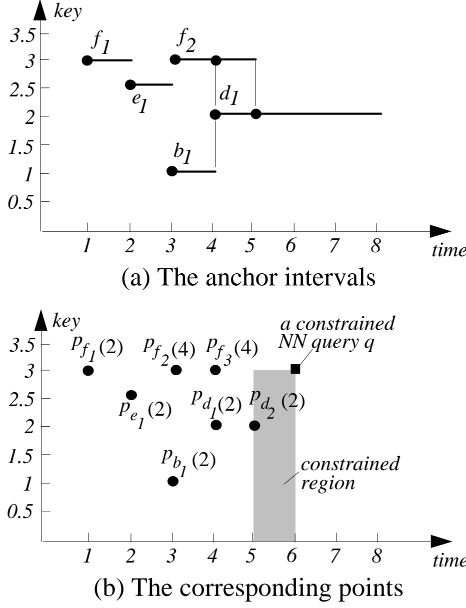


Figure 3.4: Reduction from *LKST* to constrained NN

Algorithm Generate_Anchor_Point (ε, N)
// ε is the approximation ratio, and N is the number of intervals
1. sort the starting/ending time of all intervals into L_{time}
2. initialize a B-tree *BT* (augmented as in Figure 3.3)
3. initialize an empty anchor list L_{anchor}
4. t_{last} = the first timestamp in L_{time}
5. for each timestamp t in L_{time}
6. if $t \neq t_{last}$
7. empty L_{anchor}; $t_{last} = t$
8. retrieve from *BT* the intervals whose ranks are multiples of $\varepsilon \cdot N$, and put them in L_{anchor}
9. for each interval e in L_{anchor} generate anchor point $(t, e.key)$
10. if t is a starting (ending) timestamp of interval e
11. insert (remove) e in (from) *BT*

end Generate_Anchor_Point

Figure 3.5: Algorithm for anchor point generation

We index all the $O(N)$ event timestamps and anchor points in a B- and R-tree, respectively. Given a query *LKST*(qt,qk), we first find, using the B-tree in $O(\log_B N/B)$ time, the latest event time t_a before or at qt (we say t_a is the *anchor time* of qt). For example, for the query *LKST*(6,3) in Figure 3.2c, t_a=5, i.e., the ending time of anchor interval f_2. Then, we retrieve the nearest anchor point of query point (qt,qk) in the constrained region, whose key projection is $[-\infty, qk]$ and whose time projection is $[t_a, qt]$. In Figure 3.4b the constrained region is the shaded rectangle, where the nearest anchor point retrieved is p_{d2} (in this example, there is only one point in the constrained region, while this number is $O(1/\varepsilon)$ in general). The tagged *rank* (i.e., 2) of the retrieved anchor point is the result of the approximate *LKST* query, as justified in the following lemma.

Lemma 3.5: Given query *LKST*(qt,qk), the *rank* of the nearest anchor point of (qt,qk) in the constrained region (with key/time projection $[-\infty, qk]/[t_a, qk]$), deviates from the actual result of *LKST*(qt,qk) by less than $\varepsilon \cdot N$.

Proof: The correctness of the lemma follows the observation that the anchor point returned by the constrained nearest neighbor query corresponds to the anchor interval retrieved by the timestamp search query (qt,qk). ■

According to Theorem 2.2, the R-tree answers a constrained nearest neighbor query in expected $O(\log_B N/B)$ time, which is also the overall query overhead, as stated in Theorem 3.2. Similar to the MVB-approach, this theorem still holds if the upper bound of the absolute error (of the approximate result) is lowered to $\varepsilon \cdot N_t$, i.e., the number of data intervals alive at the time query time qt.

Theorem 3.2: Given N data intervals, any approximate *LKST* query can be solved using a B- and R-tree in expected $O(\log_B N/B)$ time using $O(N/B)$ space. Both trees can be updated in amortized cost $O(\log_B N/B)$ per interval. ■

3.4 Approximate temporal sum processing

The methodology can be easily extended to approximate sum processing with the same query and space overhead. Given a set of N data intervals with weights $w_1, w_2, ..., w_N$, it suffices to discuss the approximate *LKST* sum query, since, by Lemma 3.2, a temporal sum can be reduced to 6 such queries. Let $W=\sum_{i=1}^{N} w_i$ be the sum of the weights of all data intervals, i.e., W is the largest (actual) result of any sum query. Figure 3.6a shows the weights of the 6 intervals in Figure 3.2a.

For an interval e we define its *w-rank* (at an arbitrary timestamp t during its lifespan) as the sum of the weights of all intervals (including e) alive at t with keys smaller than or equal to $e.key$. As with the rank defined in Section 3.2, the *w-rank* of an interval may also vary with time, and thus every interval is partitioned so that each *clipped interval* represents the period during which its *w-rank* remains fixed. Figure 3.6b illustrates the resulting clipped intervals with their corresponding *w-ranks*.

Similar to the approximate count problem, we convert a *LKST* sum query to a timestamp search on a set of chosen anchor intervals. Specifically, given the approximation ratio ε, we keep, at each timestamp t, the *smallest* set of clipped intervals (among those alive at t) satisfying the condition: for any clipped interval a with *w-rank* $\geq \varepsilon \cdot W$, there exists one anchor interval b such that $0 \leq b.w\text{-}rank - a.w\text{-}rank \leq \varepsilon \cdot W$. Assuming ε=1/3 (i.e., maximum error below $\varepsilon \cdot W$=1/3×35=11.7), Figure 3.6c illustrates the anchor intervals together with their *w-ranks*. At timestamp 4, for example, there are 4 alive clipped

intervals (from b, d, e, c) with *w-ranks* 2, 12, 22, 31 respectively, out of which only d_1 and f_2 (with ranks 12, 31) are anchor intervals. The algorithm for computing the anchor intervals is similar to the one for temporal count processing. We omit the details and simply summarize in theorem 3.3.

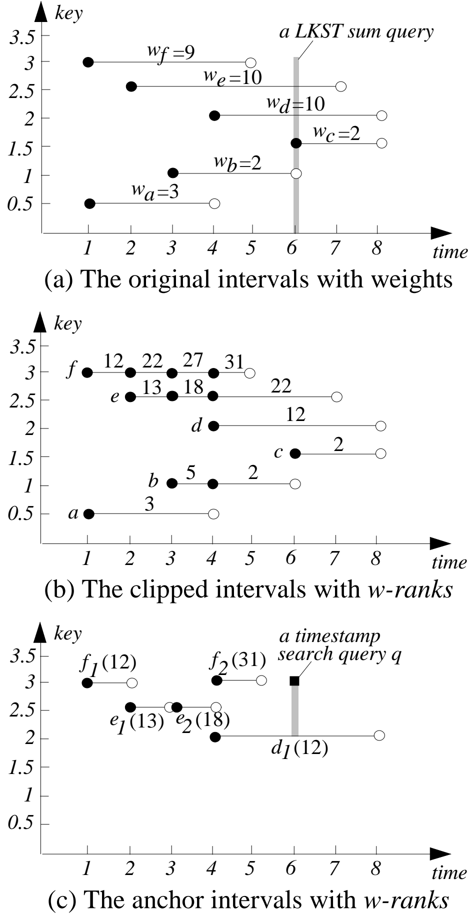


Figure 3.6: Anchor interval generation for *LKST* sum

Theorem 3.3: Given N data intervals, any approximate *LKST* sum query can be answered using a MVB-tree in $O(\log_B N/B)$ time and $O(N/B)$ space. The MVB-tree can be updated in amortized cost $O(\log_B N/B)$ per interval. ■

Similar to Theorem 4.1, the above theorem still holds even if we reduce the maximum absolute error from $\varepsilon \cdot W$ to $\varepsilon \cdot W_t$, where W_t is the sum of the weights of the intervals alive at the query time. Furthermore, we can also solve the problem with only a B- and an R-tree, by converting a query to a constrained nearest neighbor search on a set of anchor points computed using an algorithm similar to the one of Figure 3.5.

Theorem 3.4: Given N data intervals, any approximate *LKST* sum query can be solved using a B- and R-tree in expected $O(\log_B N/B)$ time using $O(N/B)$ space. Both trees can be updated in amortized $O(\log_B N/B)$ time per interval.

3.5 Discussion

To summarize, for each type of aggregate queries (i.e., count or sum), our theoretical solution consists of two separate MVB-trees, which manage the anchor intervals generated for answering *LKST* and *LKLT* queries respectively (recall that, as shown in Figure 3.1b and Lemma 3.2, *LKLT* can be reduced to *LKST*). The practical solution maintains a B- and an R-tree indexing the anchor intervals for *LKST* processing, and another two trees for *LKLT*. As demonstrated in the experiments, the overall space consumption of both solutions is much smaller than the original database size. Here we qualitatively illustrate this for count queries (the extension to sum is straightforward), adopting the following parametric modeling [TPZ02a]. The database consists of T+1 timestamps such that (i) M data intervals (whose keys follow arbitrary distribution) start at timestamp 0, and (ii) at each of the subsequent T timestamps, A percent of the intervals terminate their lifespans, while the same number of intervals start at this timestamp with distinct key values (A is the *dataset agility*[8]). Clearly the total number of intervals throughout the history equals $N = M+M \cdot A \cdot T \approx M \cdot A \cdot T$ (assuming large T). On the other hand, as proven in the previous sections, the number of anchor intervals/points at each timestamp is at most $1/\varepsilon$. As a result, the total number of anchor intervals/points is bounded by $1/\varepsilon \cdot (T+1)$, which is smaller than N if $M \cdot A > 1/\varepsilon$ (in fact, with a more complex analysis, we can prove a tighter condition $M \cdot A > 1/(2\varepsilon)$). This intuitive condition essentially says that the number of new intervals starting at a timestamp is larger than the number of anchor intervals/points we keep, which is indeed the case for "agile" datasets (see the settings of our experiments).

4. Experiments

This section experimentally demonstrates the efficiency of our methods. We generate temporal datasets in the same way as [TPZ02a], simulating the phone call history of a telecom company. Specifically, at timestamp 0 the keys of M intervals (i.e., calls) are generated in [0,100] following certain distribution *DIST*. In practice, each timestamp corresponds to the minimum billing period, e.g., 6 seconds, and its key denotes its cost. Then, at each subsequent timestamp t ($1 \le t \le 1000$, i.e., the history consists of 1000 timestamps), A% (i.e., the dataset agility) of the M intervals are randomly selected to produce key changes as follows: each such interval terminates its lifespan at t (i.e., the call ends), updates its key by some offset uniformly generated in [-1,1], and creates another

[8] In general, A may vary at different timestamps, or separate agilities may be defined for insertions and deletions (i.e., the number of alive intervals at each timestamp may change). Here we fix A for simplicity.

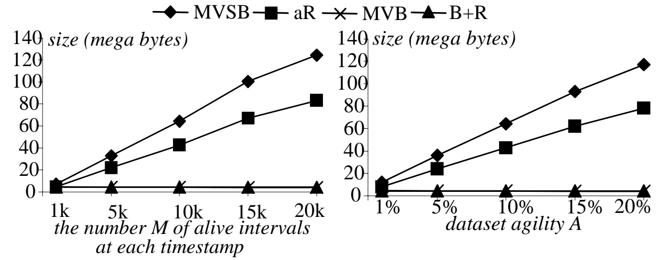
interval (with the new key) starting at t. Note that, in this way the number of alive intervals (calls) at each timestamp remains fixed (=M). We vary M from 1k to 10k, and the dataset agility A from 1% to 20%, so that the total number of data records ranges from 11k to 2 million.

The key (time) range of each temporal count/sum query is uniformly generated in [0,100] ([0,1000]), and each workload consists of 200 queries with the same parameters $|qk|$, $|qt|$, ε, which denote the lengths of the key range, time range, and approximation ratio, respectively. In the sequel, we represent $|qk|$ ($|qt|$) as the percentages over the key (time) axis respectively, e.g., 10% for $|qk|$ ($|qt|$) corresponds to key (time) range of length 10 (100).

We compare (i) the MVSB-tree [ZMT+01], (ii) the aR-tree (see Section 2.2), (iii) our MVB-tree-based solution (proposed in Section 3.2), and (iv) the combined B- and R-trees (proposed in Section 3.3, denoted as B+R in the sequel). In particular, methods (i) and (iii) are "theoretical" solutions (since they utilize access methods currently unavailable in commercial database products), while (ii) and (iv) are readily applicable in practice. The performance of each method is measured as the average number of node accesses (per query) in answering a query workload. We have conducted experiments of both count/sum queries using different key distributions *DIST*={uniform, Gaussian, Zipf}, obtaining almost identical results. Due the space constraint, in the sequel we demonstrate the diagrams for count queries and Zipf key distribution (specifically, skewed towards 0 with base coefficient 0.8, simulating the fact that most calls in practice tend to have small costs).

4.1 Size comparison

The first experiment evaluates the space consumption of the alternative solutions. In Figure 4.1a, we fix the agility A to 10%, but vary M (i.e., the number of intervals alive at a timestamp) from 1k to 20k. The approximation ratio ε of the proposed structures is set to 0.5%[9]. The space of the aR-tree corresponds to the database size, since this structure stores each interval exactly once. The size of MVSB is around 1.5 times larger than the database size (recall that its space complexity is $O(N/B\log_B N/B)$). The proposed methods, however, are significantly smaller (e.g., for M=20k, they consume about 1/20 of the database size) because they only keep a constant (i.e., at most $1/\varepsilon$=200) number of anchor intervals/points at each timestamp t, regardless of the total number of intervals alive at t. Observe that, in contrast to the competitors, the space consumption of MVB and B+R actually decreases for larger M. This happens because the maximum absolute error allowed (with the same ε) grows for higher M, resulting in even fewer anchor intervals/points maintained at each timestamp. Figure 4.1b examines the structural sizes as a function of dataset agility A, fixing M to 10k, confirming similar observations.



(a) Size vs M (A=10%) (a) Size vs A (M=10k)

Figure 4.1: Size comparison (ε=0.5%)

Figure 4.2 examines the space consumption of the proposed methods for various approximation ratios, using the median values of M (=10k) and A (=10%). For comparison, we also illustrate the sizes of MVSB and aR-trees. As expected, the approximate solutions require more space for higher precision, e.g., for the lowest tested value of ε=0.1%, both structures require half the data base size.

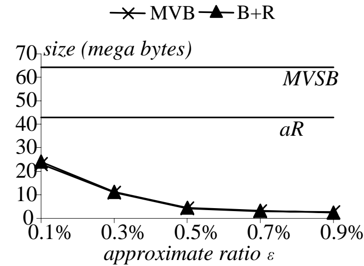


Figure 4.2: Size vs approximation ratio (M=10k, A=10%)

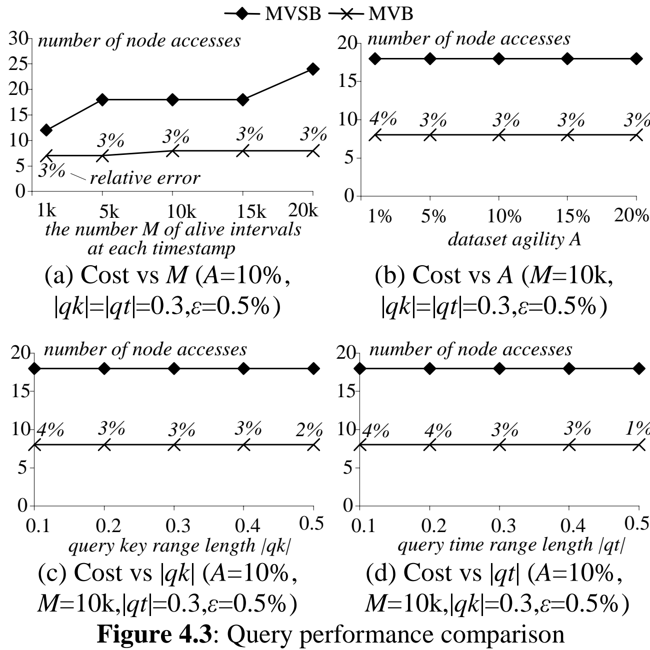
4.2 Query cost evaluation of theoretical solutions

In this section, we compare the query performance of the MVSB-tree and the proposed MVB-tree technique, since they both require specialized access methods not available in the existing products. In Figure 4.3a, we test the query cost (averaged over all queries in a workload) as a function of M (i.e., the number of intervals alive at the same timestamp), fixing agility A to 10%, and the query parameters $|qk|$, $|qt|$ to 0.3. For MVB-trees we use ε=0.5% (the same value used for the size experiments in Figure 4.1) and we also illustrate the corresponding relative error of the approximate results. Specifically, let act_i and apr_i denote the actual and approximate results of the i-th ($1 \le i \le 200$) query in the workload respectively; the relative error is defined as $(1/200)\cdot\sum_{i=1}^{200}|act_i-apr_i|/act_i$.

The query cost of the MVB is considerably lower than that of MVSB, and remains (almost) constant as M increases (while the cost of MVSB degrades continuously). Recall that the performance of MVB and MVSB is determined by the heights of the logical trees

[9] We quantify the relative error of different approximation ratios in Section 4.2.

(i.e., B- and SB-trees) in the respective multi-version structures. The height is lower for MVB since its logical tree only needs to index the anchor intervals, the number of which is smaller than the number of original data intervals, indexed by a logical tree in MVSB. As mentioned earlier, the number of anchor intervals kept at each timestamp does not increase with *M*, which explains the constant cost of MVB (the "step-wise" growth of MVSB cost corresponds to height increases). The accuracy of our approximate results is very high (maximum relative error 4%) since the absolute error is bounded using a very small ε. Furthermore, the relative error does not vary with *M* because larger *M* increases both the query result and the absolute error.



(a) Cost vs *M* (*A*=10%, |*qk*|=|*qt*|=0.3,ε=0.5%)
(b) Cost vs *A* (*M*=10k, |*qk*|=|*qt*|=0.3,ε=0.5%)
(c) Cost vs |*qk*| (*A*=10%, *M*=10k,|*qt*|=0.3,ε=0.5%)
(d) Cost vs |*qt*| (*A*=10%, *M*=10k,|*qk*|=0.3,ε=0.5%)

Figure 4.3: Query performance comparison

Figure 4.3b evaluates the query performance by varying the agility from 1% to 10%. The overhead of both structures remains the same, because the heights of their logical trees depend only on the number of intervals alive at a single timestamp (which is not relevant to the dataset agility). Figures 4.3c and 4.3d plot the query cost as a function of query parameters |*qk*| and |*qt*|, respectively. As expected, the costs of both structures are not affected by the query size, i.e., their costs do not depend on how many intervals qualify the query predicates (while larger queries lead to even smaller relative error). *In all cases, the MVB-tree outperforms the MVSB-tree by a factor of 2, although its size is 20 times smaller (as shown in Figure 4.1a) and the relative error is at most 4%.*

The next experiment explores the behavior of MVB with various approximation ratios ε (from 0.1% to 0.5%). As shown in Figure 4.4, the query cost is higher for small ε (which offers better precision, i.e., less than 1% relative error) due to the increase in the height of the logical tree (in particular, note that the costs for ε=0.1%, 0.3% are double those for higher ε values). Nevertheless, MVSB is still more expensive in all cases.

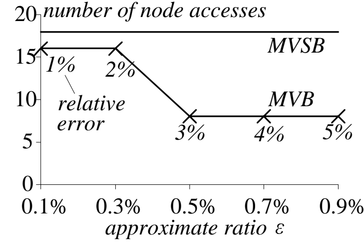
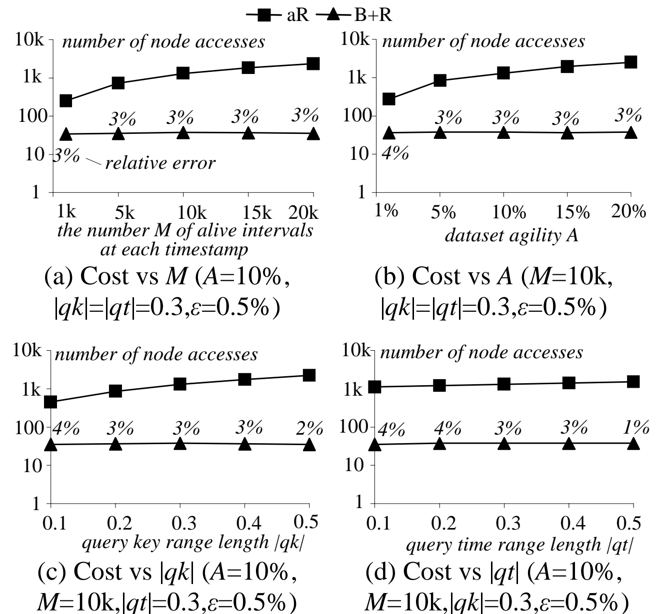


Figure 4.4: Cost vs. ε (*A*=10%, *M*=10k, |*qk*|=|*qt*|=0.3)

4.3 Query cost evaluation of practical solutions

This section compares the costs of the aR-tree and the proposed B+R technique. Figure 4.5 repeats the experiments of Figure 4.3. It is clear that the B+R tree outperforms the aR-tree, usually by more than an order of magnitude (note that the y-axis is in logarithmic scale). In particular, its performance is independent of all parameters, due to the fact that the cost of a constrained nearest neighbor search only relies on the fractal dimension of the dataset (which is the same in all cases), as discussed in Section 2.3. On the other hand, the overhead of the aR-tree is prohibitive, and increases with the number of qualifying intervals, which is consistent with the existing understanding of this structure [PKZT01]. In summary, *the B+R technique is about an order of magnitude faster, and consumes an order of magnitude less space than the aR-tree, while its maximum error for these settings is only 4%.*



(a) Cost vs *M* (*A*=10%, |*qk*|=|*qt*|=0.3,ε=0.5%)
(b) Cost vs *A* (*M*=10k, |*qk*|=|*qt*|=0.3,ε=0.5%)
(c) Cost vs |*qk*| (*A*=10%, *M*=10k,|*qt*|=0.3,ε=0.5%)
(d) Cost vs |*qt*| (*A*=10%, *M*=10k,|*qk*|=0.3,ε=0.5%)

Figure 4.5: Query performance comparison

Similar to Figure 4.4, Figure 4.6 evaluates the effect of the approximation ratio ε on the query performance. The cost of R+B increases only slightly, even for the smallest ε, and is always considerably lower than its competitor.

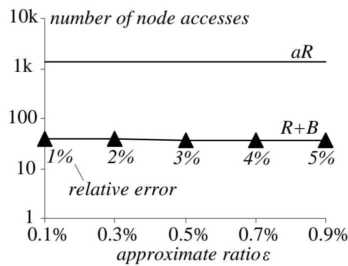


Figure 4.6: Cost vs. ε (A=10%, M=10k, $|qk|$=$|qt|$=0.3)

5. Conclusion

This paper presents novel techniques for approximate aggregate processing in temporal databases. In particular, we propose two sets of solutions. The first one utilizes a specialized structure (i.e., the MVB-tree) and achieves linear space (with respect to the database size) and logarithmic query time in the worst case. The second solution leverages off-the-shelf B- and R-trees, and provides the same performance in the expected case. As confirmed through extensive experimentation, our techniques are economical in terms of space consumption and outperform significantly (usually by more than an order of magnitude) the exact methods, while at the same time offering high approximation accuracy.

In this work we address the count and sum aggregate functions, leaving the other aggregate functions open. In particular, it would be interesting to investigate the *average* processing (i.e., the quotient of count and sum), where the main difficulty is that bounding the absolute error of both count and sum does not necessarily limit the error of the quotient. Another promising direction for future work is to extend this technique to spatio-temporal aggregation [PTKZ02], where the database contains the object (e.g., vehicle) locations at any timestamp in history, and the goal is to find (approximately) the number of objects in a specific query region during a certain time interval. The potential solution requires applying the proposed method in higher dimensionality which, however, may require more sophisticated reductions.

Acknowledgements

This work was supported by grant HKUST 6197/02E from Hong Kong RGC.

References

[B00] Bohm, C. A Cost Model for Query Processing in High Dimensional Data Spaces. TODS, 25(2): 129-178, 2000.

[BGO+96] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P. An Asymptotically Optimal Multiversion B-tree. The VLDB Journal, 5: 246-275, 1996.

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

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

[FK94] Faloutsos, C., Kamel, I. Beyond Uniformity and Independence, Analysis of R-trees Using the Concept of Fractal Dimension. PODS, 1994.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agarwal, D., Abbadi, A. Constrained Nearest Neighbor Queries. SSTD, 2001.

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

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

[KPF01] Korn, F., Pagel, B., Faloutsos, C. On the Dimensionality Curse and the Self-Similarity Blessing. TKDE, 13(1): 96-111, 2001.

[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.

[ST99] Salzberg, B., Tsotras, V. A Comparison of Access Methods for Temporal Data. ACM Computing Survey, 31(2): 158-221, 1999.

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

[TPZ02a] Tao, Y., Papadias, D., Zhang, J. Cost Models for Overlapping and Multiversion Structures. TODS, 27(3): 299-342, 2002.

[TPZ02b] Tao, Y., Papadias, D., Zhang, J. Aggregate Processing of Planar Points. EDBT, 2002.

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

[VV97] Varman, P., Verma, R. Optimal Storage and Access to Multiversion Data. TKDE, 9(3): 391-409, 1997.

[YW01] Yang, J., Widom, J. Incremental Computation and Maintenance of Temporal Aggregates. ICDE, 2001.

[ZGTS03] Zhang, D., Gunopulos, D., Tsotras, V., Seeger, B. Temporal and Spatio-Temporal Aggregations over Data Streams using Multiple Time Granularities. Information Systems, 28(1-2): 61-84: 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.