

# Efficient Temporal Counting with Bounded Error

Yufei Tao      Xiaokui Xiao

Department of Computer Science and Engineering  
Chinese University of Hong Kong  
New Territories, Hong Kong  
{taoyf, xkxiao}@cse.cuhk.edu.hk

## Abstract

This paper studies aggregate search in transaction time databases. Specifically, each object in such a database can be modeled as a horizontal segment, whose y-projection is its search key, and its x-projection represents the period when the key was valid in history. Given a query timestamp  $q_t$  and a key range  $\vec{q}_k$ , a count-query retrieves the number of objects that are alive at  $q_t$ , and their keys fall in  $\vec{q}_k$ . We provide a method that accurately answers such queries, with error less than  $\frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t)$ , where  $N_{alive}(q_t)$  is the number of objects alive at time  $q_t$ , and  $\varepsilon$  is any constant in  $(0, 1]$ . Denoting the disk page size as  $B$ , and  $n = N/B$ , our technique requires  $O(n)$  space, processes any query in  $O(\log_B n)$  time, and supports each update in  $O(\log_B n)$  amortized I/Os. As demonstrated by extensive experiments, the proposed solutions guarantee query results with extremely high precision (median relative error below 5%), while consuming only a fraction of the space occupied by the existing approaches that promise precise results.

**To appear in VLDB Journal.**

**Keywords:** Temporal Database, Aggregate Search, Approximate Query Processing

## 1 Introduction

A traditional database discards the old values of a tuple once it has been updated. Such a “now-only” approach is not sufficient in many applications where it is necessary to query about the past versions of data. These applications have triggered a large amount of research on *temporal databases* (see [30] for an excellent survey), which preserve all the historical values of a set of entities. In addition to its conventional attributes, a temporal tuple is also associated with a time interval, indicating the period during which the tuple correctly describes the underlying entity. To illustrate the idea, let us visit two modern applications where a temporal database plays an imperative role.

**Application 1 (Banking).** Maintenance of historical versions is crucial in a bank database that stores account balances. Figure 1 demonstrates some tuples in such a database, where the current time equals 8. Each tuple is represented as a horizontal segment, whose projection along the x-dimension indicates its valid duration (with the time granularity being a day), and its projection on the y-dimension captures the balance of an account in that duration. For instance, segment  $o_1$  corresponds to an account whose balance was 3.5k dollars in the period of  $[1, 5)$ . Note that the interval is semi-closed, meaning that the balance ceased to be 3.5k on Day 5 (in Figure 1, this is reflected by a white dot at the end of  $o_1$ ). Segments  $o_4$  and  $o_5$  do not carry white dots, because they represent the balances of two accounts that are valid *now*.

Every account deposit or withdrawal generates a new tuple in the database. For instance, segments  $o_6$  and  $o_5$  may describe the balances of the same account in different periods. In particular, this account had a balance of 1k dollars in  $[1, 6)$ . Then, on the 6th day, 500 dollars were deposited into the account, whose balance has remained 1.5k until the current time.

Although  $o_1, o_2, o_3, o_6,$  and  $o_7$  are “obsolete” (since they capture balances that are no longer valid), they cannot be expunged, because of the need to support queries that explore a past “snapshot” of the database. For example, a query may be to “retrieve the accounts whose balances are at most 3.5k dollars on Day 6”. The result includes  $o_3, o_4$  and  $o_5$  ( $o_6$  does not satisfy the predicate because its valid period does not include time 6). These objects correspond to the segments intersecting the vertical column at time 6.

**Application 2 (Sensor Monitoring).** Consider a meteorology system that monitors the temperatures at numerous regions in a state. Each temperature reading is taken by a sensor, and periodically (e.g., every 30 minutes) transmitted to a central server. It is easy to imagine that every tuple in the server’s database can also be visualized as a horizontal segment, whose y-projection is the measured temperature of a sensor, and its x-projection denotes the interval between two updates. For instance, several consecutive reports from the same sensor may be recorded as (89F, [9pm, 9:30am)), (90F, [9:30am, 10am)), and so on (in practice, the readings of nearby sensors may be averaged into a single value, to indicate the temperature of a large district). All the

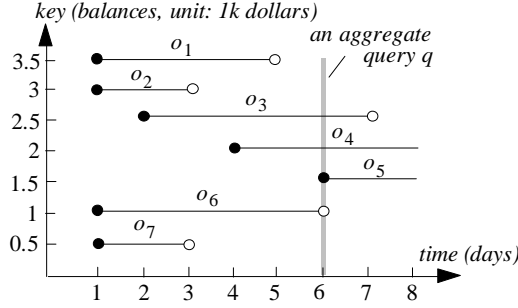


Figure 1: Object representations in a temporal database

historical measurements must be retained, for supporting queries like “find the districts whose temperatures are above 95F at 2pm on January 1, 2006”.

**Motivation.** In many applications, a user is interested in only *statistical* information, rather than the IDs of the qualifying objects. A typical example is an online analytical processing (OLAP) system, which provides fast aggregate results about the underlying database, according to the user-specified query conditions. For instance, in Application 1, an OLAP query may be to “find the *number* of accounts whose balances are at most 3.5k dollars on Day 6”. The system returns a single answer 3, as opposed to the concrete IDs as explained earlier. A similar query in Application 2 may demand the number of districts whose temperatures fall in a certain range on a given day. As reviewed in Section 2, these *counting* queries have been widely studied in temporal and non-temporal domains, due to their importance both as a stand-alone operator, and as the building-brick of many other data mining methods, such as association rule mining [1], decision tree construction [2], etc.

Evidently, a count query can be trivially supported by first enumerating all the objects that satisfy the query predicate, and then counting the number of retrieved objects. However, since our objective is to return merely a single integer, we should expect a solution that significantly outperforms that trivial approach (observe that the trivial approach finds considerable redundant information). Motivated by this, in this paper we explore methods that are specialized for extracting statistical data.

### 1.1 Problem Definition

Let  $DB$  be a temporal dataset with cardinality  $N$  where each object is defined as follows:

**Definition 1.** A **temporal object**  $o$  is represented by a constant **search key**, denoted as  $o.key$ , and a semi-closed interval  $[o.t_-, o.t_+)$  called the **lifespan** of  $o$ .

For simplicity, we abbreviate “search key” simply as “key”, which, obviously, should be distinguished with the “primary key” of a relation. Furthermore, without loss of generality, all timestamps are assumed to be

positive. We consider the *transaction time model* [30], where every change to the database is stamped with the *current time CT*, which is a monotonically increasing value. The model permits two types of updates: “insertion” and “logical deletion”. Specifically, an *insertion* adds a new object  $o$  to  $DB$  with a lifespan  $[o.t_+, *)$ , where  $o.t_+$  equals  $CT$ , and the asterisk indicates that the lifespan of  $o$  will continuously evolve as  $CT$  increases. A *logical deletion*, on the other hand, terminates the lifespan of an existing object  $o$ , by replacing its “\*” with the current  $CT$ , which becomes the final  $o.t_+$ . We say that  $o$  is *born* at time  $o.t_+$ , and *dies* at time  $o.t_+$ , i.e., it is *alive* during its lifespan. Logical deletions are applied only to live objects.

Therefore, a temporal dataset is uniquely decided by a sequence of updates, which we call a *temporal stream*. For example, the data in Figure 1 is determined by the following stream:  $\langle \text{INS}, 3.5, 1 \rangle$ ,  $\langle \text{INS}, 3, 1 \rangle$ ,  $\langle \text{INS}, 1, 1 \rangle$ ,  $\langle \text{INS}, 0.5, 1 \rangle$ ,  $\langle \text{INS}, 2.5, 2 \rangle$ ,  $\langle \text{LD}, 3, 3 \rangle$ ,  $\langle \text{LD}, 0.5, 3 \rangle$ ,  $\langle \text{INS}, 2, 4 \rangle$ , ... Each update has the form  $\langle \text{type}, \text{key}, \text{time} \rangle$ , where “type” is INS (insertion) or LD (logical deletion), “key” is the key of the object being inserted/logically deleted, and “time” always equals the current  $CT$ . The first 4 updates of the stream add  $o_1, o_2, o_6, o_7$  into  $DB$ , all of which have lifespans  $[1, *)$  at the end of time 1. The 5th update inserts  $o_3$  with lifespan  $[2, *)$ , whereas the 6th ends the lifespan of  $o_2$  at time 3. The effect of the next two updates can be understood in the same manner. As a result, at the end of time 4, there are four live objects:  $o_1, o_3, o_4$  and  $o_6$ , which have lifespans  $[1, *)$ ,  $[2, *)$ ,  $[4, *)$ , and  $[1, *)$ , respectively.

We are interested in timestamp range counting formulated as follows:

**Definition 2.** Given a timestamp  $q_t$  and a range  $\vec{q}_k = [q_{k+}, q_{k-}]$ , a **timestamp range count query**  $q$  returns

$$|QS(q_t, \vec{q}_k)| \tag{1}$$

where  $QS(q_t, \vec{q}_k)$  is the set of objects  $o \in DB$  alive at time  $q_t$ , whose  $o.key(q_t)$  at time  $q_t$  falls in  $\vec{q}_k$ .

As reviewed in Section 2, the above query type has been studied previously. The fastest solution [38], which settles a query in  $O(\log_B n)$  I/Os, requires  $O(n \log_B n)$  space<sup>1</sup>. Here,  $B$  denotes the disk page size, and  $n$  equals  $N/B$ . The  $O(n \log_B n)$  space requirement limits the applicability of the solution in practice, because a temporal database is typically extremely large, and its size continuously grows with time due to its appending nature (imagine a database which manages bank accounts for 10 years).

In practice, a user can often tolerate small *bounded* error, provided that finding the approximate answer requires less storage and computation resources. Motivated by this, we define an approximate version of the queries in Definition 2:

---

<sup>1</sup>If no insertion and deletion are allowed on  $DB$ , there exists a solution, based on the CRB-tree [15], which can bring the space consumption down to  $O(n)$ , without increasing the query time complexity. Nevertheless, if updates must be supported, as is the goal of this paper, the query cost of that solution becomes  $O(\log_B^2 n)$ . Furthermore, CRB-trees work only on a machine that permits “bit compression”, a complex process that would be hard to implement in practice. We discuss the details in Section 2.1.

**Definition 3.** Given a timestamp  $q_t$  and a range  $\vec{q}_k = [q_{k+}, q_{k-}]$ , an **approximate timestamp range count query**  $q$  returns a value that deviates from Formula 1 by less than

$$\frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t) \quad (2)$$

where  $N_{alive}(q_t)$  is the number of objects  $o \in DB$  alive at  $q_t$ , and  $\varepsilon$  is a system constant in the range of  $(0, 1]$  termed the **approximation ratio**.

Note that the error bound  $\frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t)$  depends on the “snapshot” of  $DB$  at the query time  $q_t$  (analogous definitions in the form of “approximate quantile” are popular in non-temporal environments [3, 16, 25]). This is a significant improvement over the work of [32], where the authors deal with a similar problem, but formulate the error bound as  $\varepsilon \cdot N$ . This bound can be extremely loose, since it is proportional to the total number of objects in history (i.e., including those not alive in  $q_t$ ).

This paper aims at solving the following problem:

**Problem 1.** Preprocess  $DB$  into a space-economical structure such that any query of Definition 3 can be answered in a small number of I/Os.

## 1.2 Contributions and Paper Organization

We settle Problem 1 by building a data structure on  $DB$  that consumes linear space  $O(n)$ , answers any query in  $O(\log_B n)$  I/Os, and can be maintained in amortized  $O(\log_B n)$  I/Os per update<sup>2</sup>. As a side product, our solution achieves the same performance, even if the maximum error of a timestamp range count query must be controlled under  $\varepsilon \cdot N_{alive}(q_t)$ , as long as the dataset satisfies a weak condition (see Section 6). Note that all the complexities are independent of  $\varepsilon$ .

The above theoretical results are made possible with a novel methodology for tackling temporal aggregation. Specifically, we aim at replacing  $DB$  with a set of “anchor segments”, which can be used to provide satisfactory answers for queries on  $DB$  (as a result,  $DB$  does not need to be materialized at all). This set has  $O(N)$  size in theory, but in practice, is *significantly smaller* than  $DB$ . We prove experimentally that our technique provides extremely accurate results of timestamp range counting (with median relative error below 5%), while consuming only a fraction of the space required by the previous methods that promise precise answers.

Our findings directly lead to an approach for approximately processing the “sum” counterpart of the queries in Definition 3. In that case, each object is associated with an integer weight, and a query  $q$  returns the sum

<sup>2</sup>If  $DB$  does not need to be updated, our technique has better complexities:  $O(\min\{n, n \cdot (\frac{1}{\varepsilon})^2 / \overline{N_{alive}}\})$  space, and  $O(\min\{\log_B n, \log_B \frac{n}{\varepsilon} - \log_B \overline{N_{alive}}\})$  query time, where  $\overline{N_{alive}}$  is around the average number of objects alive at a timestamp.

of the weights of all objects in  $QS(q_t, \vec{q}_k)$ , where function  $QS(\cdot)$  is explained in Definition 2. Specifically, we only need to duplicate an object as many times as its weight, and then apply the proposed count-solutions on the resulting dataset. To guarantee maximum error less than  $\frac{1}{\varepsilon} + \varepsilon \cdot W_{alive}(q_t)$ , where  $W_{alive}(q_t)$  is the sum of the weights of all objects alive at  $q_t$ , any timestamp range sum query can be answered in  $O(\log_B W)$  I/Os, using a structure that occupies  $O(W/B)$  space, and incurs amortized  $O(\log_B n)$  I/Os per update. Here,  $W$  is the sum of the weights of all the original objects in  $DB$ . If each object’s weight is confined to no more than a constant (i.e.,  $W = O(N)$ ), both the space and query complexities are identical to those for count retrieval.

The rest of the paper is organized as follows. Section 2 surveys the previous approaches related to temporal aggregation. Section 3 simplifies timestamp range counting by reducing it to “half-ranged” retrieval. Section 4 illustrates the rationale of anchor segments with a relatively simple method for settling Problem 1. Then, Section 5 improves the method by reducing its space consumption significantly. Section 6 further enhances the performance of our solution with a “zoning transformation”, and Section 7 clarifies how to support data updates. Section 8 contains an extensive experimental evaluation that demonstrates the superiority of the proposed technique over alternative approaches. Finally, Section 9 concludes the paper with directions for future work.

## 2 Related Work

In Section 2.1, we summarize the existing approaches that tackle various forms of temporal aggregate search. Then, Section 2.2 reviews the methods of spatial aggregation, clarifies their relevance to our problem, and discusses other aggregation techniques in non-temporal scenarios. Finally, Section 2.3 provides an introduction to the multi-version B-tree, which is an important temporal access method, and the foundation of our solutions.

### 2.1 Temporal Aggregation

Based on an interesting taxonomy in [26], the previous research in this area can be classified into two categories: *sequenced* and *non-sequenced*. Specifically, the former aims at producing the aggregate results at all timestamps in history, whereas the objective of the latter is to perform aggregation during only a specific period. In the sequel, we will discuss the two categories in turn (see [9] for richer semantics of “sequenced” and “non-sequenced”).

**Sequenced Retrieval.** Conceptually, a sequenced query first divides a dataset into as many partitions as the number of timestamps in history, such that each partition consists of all the objects alive at the same timestamp. The partitioning is analogous to the conventional “group-by” operation, except that an object, depending on the length of its lifespan, may appear in multiple partitions (as opposed to only one group

in a group-by). Then, the query returns an aggregate value for the objects in each partition, according to an aggregate function, e.g., COUNT, SUM, AVERAGE, etc. Optionally, a user may also specify a filtering condition to eliminate the non-qualifying objects before the partitioning. For example, a sequenced request may be to “return the number of accounts whose balances are larger than 100k dollars on every single day in the past”, where “larger than 100k dollars” is a condition, and a partition contains all the accounts alive on a specific day.

All the existing algorithms for sequenced computation are based on a two-step framework. The first phase scans the entire database to construct an index structure, which encodes useful information about all the partitions, and is traversed in the second step to report the aggregate results. Assuming the whole dataset to fit in memory, Kline and Snodgrass [22] propose a *k-ordered aggregate tree* as the intermediate index, which guarantees the overall computation cost of  $O(N^2)$ , where  $N$  is the cardinality of the dataset. Also focusing on memory-resident datasets, Kim et al. [21] present a faster method that replaces the *k-order aggregate tree* with a *point-based aggregation tree*, and reduces the query cost to  $O(N \log_2 N)$ . Leveraging a new structure called *the balanced tree*, Moon et al. [27] provide another solution that achieves the same complexity, and develop several I/O efficient algorithms by combining the solution with a *meta-array* approach [27].

The aforementioned methods target a machine with only one CPU and disk, whereas accelerating sequenced queries with parallel computing has been explored in [14, 27, 36]. Recently, Bohlen et al. [8] define new semantics of sequenced queries by extending the multidimensional join operator in [10].

**Non-Sequenced Retrieval.** A query of this type returns the aggregate result about the objects alive at a particular timestamp (c.f., Definitions 2 and 3), or during a continuous time interval (e.g., from January 1 to February 1, 2006). Although an algorithm in the sequenced category can obviously be applied to solve a non-sequenced query, the application is costly, because it also fetches a large amount of useless information. A (much) more efficient approach is to pre-compute an index on the given dataset, which, given a query, can guide the processing directly to the time period of interest, thus avoiding accessing the irrelevant data.

Yang and Widom [35] propose the *segment B-tree* (SB-tree), which is the external-memory counterpart of an adapted interval tree [7]. Specifically, an SB-tree manages a set of  $N$  one-dimensional intervals using  $O(n)$  space, and allows finding the number of intervals containing any query value (in the same one-dimensional domain) in  $O(\log_B n)$  I/Os. For example, we can create an SB-tree on the lifespans of the objects in Figure 1, and use it to efficiently “find how many accounts were active on January 1, 2006”. Note that the query is a special instance of Problem 1 where  $q_t = 01/01/2006$ , and  $\vec{q}_k$  covers the entire key (i.e., attribute “balance”) domain, namely,  $\vec{q}_k = (-\infty, \infty)$ .

Since an SB-tree does not incorporate any information about the key dimension, it entails expensive cost in processing a general query  $q$  of Definition 2 whose  $\vec{q}_k$  is a subrange of  $(-\infty, \infty)$ . In this case, we must first

retrieve all the objects alive at time  $q_t$ , and then filter them according to  $\vec{q}_k$ . To remedy the drawback, Zhang et al. [38] propose the MVSB-tree (later adapted in [37] to data streams), which extends the SB-tree using a “multi-version” technique discussed in Section 2.3. An MVSB-tree on  $N$  objects consumes  $O(n \log_B n)$  space, can be updated in  $O(\log_B n)$  I/Os, and permits any query of Definition 2 to be processed in  $O(\log_B n)$  I/Os. It is the state-of-the-art solution for *precise* temporal aggregation.

As mentioned in Section 1.1, for very large databases, spending  $O(n \log_B n)$  space cost on indexing may not be realistic. Thus, it is natural to explore approaches that return approximate answers (with bounded error), but require space linear to  $n$ . Some efforts towards this goal have been made in [32], where the authors show how to provide an answer to any timestamp range count query with error less than  $\varepsilon \cdot N$ , and  $\varepsilon$  is a pre-defined constant in the range of  $(0, 1]$ . Their method takes up  $O(\frac{n}{\varepsilon})$  space, and handles any query in  $O(\log_B \frac{n}{\varepsilon})$  I/Os. Since the error bound  $\varepsilon \cdot N$  is proportional to the total number of objects in the dataset (rather than those alive at the query timestamp), an approximate result may deviate too much from the real answer, and is thus of limited use in practice.

Finally, it is worth mentioning that, although the above analysis concentrates on counting, the non-sequenced methods mentioned earlier can also tackle other aggregation functions (e.g., SUM and AVERAGE), with some straightforward modifications.

## 2.2 Non-Temporal Aggregation

Also closely related to the topic of this paper is aggregate retrieval on rectangles (including points). In the sequel, we will explain why, and present the relevant previous results. Then, we will discuss other approximate query processing techniques.

**Spatial Aggregation.** As demonstrated in Figure 1, each temporal object can be modeled as a horizontal segment in the key-time space, and similarly, a timestamp range count query as a vertical segment. Since axis-parallel segments are merely degenerated rectangles, timestamp range count search is a special instance of the following “rectangle intersection counting” problem: Given a set  $S$  of  $N$  data rectangles and a query rectangle  $q$ , find the number of rectangles in  $S$  that intersect  $q$ . Zhang et al. [40] propose two versions of the ECDF-B-tree that solve this problem with different tradeoffs between space consumption and query cost (actually, ECDF-B-trees settle a more general problem termed “dominance-sum”; see [40] for details). Specifically, the first version requires  $O(n \log_B n)$  space and  $O(B \log_B^2 n)$  query time, while the second one demands  $O(N \log_B n)$  space but reduces the query overhead to  $O(\log_B^2 n)$  I/Os. All complexities also apply to queries of Definition 2.

The rectangle intersection counting problem becomes “point enclosure counting” when all the data rectangles degenerate into points. Interestingly, as mentioned in [15], there is a subtle reduction from temporal

aggregation (Definition 2) to point enclosure counting. In particular, as long as the latter problem can be settled, the former can also be solved with the *same* space and query asymptotical performance (the reverse is true, too: point enclosure counting can also be reduced to temporal aggregation [33]). In [33], a structure called the *aggregation point tree* (aP-tree) is developed to support point enclosure counting. Specifically, given  $N$  data points, we can create an aP-tree that occupies  $O(n \log n)$  space, and answers any point enclosure count query in  $O(\log_B n)$  I/Os. In other words, the aP-tree has the same performance as the MVSb-tree (reviewed in Section 2.1) in timestamp range aggregation.

Govindarajan et al. [15] show that, on a machine that permits “bit compression”, point enclosure counting can be optimally solved in  $O(\log_B n)$  I/Os by a *compressed range B-tree* (CRB-tree) that consumes  $O(n)$  space. Bit compression [11] requires that every integer  $x$  should be stored in exactly  $\lceil \log_2 x \rceil$  bits; as a result, multiple integers can be packed into a single machine word to save space. For example,  $x = 10$  must occupy only 4 bits, so that four integers ‘10’ can be stored using a single word with 16 bits, as opposed to using 8 words in a practical machine (32 bits per integer). Although bit compression is technically possible, it complicates implementation considerably, and, to our knowledge, is not supported by any commercial database product. Furthermore, the excellent efficiency of the CRB-tree is possible only if the structure can be preserved in a large number of sequential pages. This is necessary because the CRB-tree contains numerous secondary structures implemented as arrays, which is essential to ensure the  $O(\log_B n)$  query time (benefiting from the fact that accessing any element in an array needs  $O(1)$  I/Os). When the vacant pages are fragmented (as is true, if updates are frequent), an array must be replaced with a linked list, in which case the query performance can become significantly worse. Finally, the CRB-tree is by default static, that is, it does not support insertions and deletions. To remedy this defect, Govindarajan et al. [15] apply the “logarithmic method” [4] to “dynamize” the CRB-tree, which, however, compromises the query efficiency. In particular, the resulting structure incurs  $O(\log_B^2 n)$  I/Os per query and update, namely, slower than the MVSb-tree.

Besides the above methods that have low worst-case space and query overhead, there also exist several “engineering approaches” [23, 28, 39], which deploy heuristics to accelerate aggregate retrieval on spatial data. Specifically, these heuristics augment an R-tree [6, 17] with various statistics in the intermediate entries, and lead to an alternative structure called the *aggregate R-tree* (aR-tree). Such a structure consumes  $O(n)$  space, and has reasonably good query performance on “practical” data distributions, although, for an adversely designed dataset, the query cost can be  $O(n)$ .

**Other Aggregation Techniques.** The error bound of Formula 2 is reminiscent of the precision guarantee of an  $\varepsilon$ -approximate quantile summary. Given  $N$  one-dimensional values, we can build such a summary (whose size is by far smaller than  $O(N)$ ), which enables reporting the number of data values smaller than or equal to any query value accurately, with error at most  $\varepsilon \cdot N$ . Several algorithms [3, 16, 25] have been proposed to maintain the summary, when the underlying values are updated according to different models

(e.g., insertion only [16] or “sliding window” [3, 24, 25]). In our context, these algorithms may be applied to support queries of Definition 3 whose query time  $q_t$  equals now (i.e., aggregating the currently alive objects). In this paper, however, we aim at answering queries concerning any timestamp in the past.

The point enclosure counting problem mentioned earlier also exists in data warehouses, where all the attributes of the workspace have discrete domains. In this scenario, every data point can be thought of as a cell in a multidimensional array. In preprocessing [12, 13, 18, 29], a value is computed and materialized for every cell in the array, regardless of whether it contains a data point; then, every query can be answered in a small number of lookups on the materialized information. This technique is not applicable to our problem (even with the reduction from temporal aggregation to point enclosure counting) because objects’ keys are distributed in a continuous domain. Furthermore, the multidimensional materialization requires prohibitive space consumption, i.e., proportional to the size of the array, rather than to the dataset cardinality.

Finally, approximate aggregate retrieval is also related to selectivity estimation, which has been tackled using numerous methodologies (typically, histograms [31] and sampling [19]). These approaches can obtain accurate estimates for most queries, but may suffer from considerable error in the worst case.

### 2.3 The Multi-Version B-Tree

The *multi-version B-tree* (MVB-tree) [5, 34] is an access method for transaction time databases that can process *timestamp range queries* optimally. Given a timestamp  $q_t$  and a key range  $\vec{q}_k$ , such a query returns the set  $QS(q_t, \vec{q}_k)$  in Definition 2 (i.e., all the objects alive at time  $q_t$  whose keys are contained in  $\vec{q}_k$ ). An MVB-tree manages a temporal dataset of  $N$  objects using  $O(n)$  space, and solves any timestamp range query in  $O(\log_B n + m/B)$  I/Os, where  $m$  is the number of retrieved objects. For a temporal stream explained in Section 1.1, an MVB-tree can be incrementally maintained in  $O(\log_B n)$  I/Os per update.

Conceptually, for every historical timestamp  $t$ , an MVB-tree keeps a conventional B-tree, which indexes the keys of all the objects alive at  $t$ , and has a node fanout  $O(B)$ . Given a timestamp range query, the algorithm first identifies the B-tree responsible for the query time  $q_t$ , and then, retrieves the qualifying objects from this tree in the same way as executing a one-dimensional range query in a normal B-tree. The identification (of the responsible B-tree) takes  $O(\log_B n)$  time, and the subsequent processing (in the identified tree) requires  $O((\log_B N_{alive}(q_t)/B) + m/B)$  I/Os, where  $N_{alive}(q_t)$  is the number of objects alive at time  $q_t$ .

Evidently, it is intractable to physically retain the B-trees at all timestamps (in particular, this is simply impossible, if the time dimension has a real domain). However, the B-trees at two consecutive timestamps usually share a large common portion (corresponding to the objects alive at both timestamps), which only needs to be stored once. This is the rationale behind a sophisticated “multi-version technique” that embeds a perhaps infinite number of B-trees in  $O(n)$  space (see [5, 34] for the details). In fact, the technique is general,

since it can be applied to embed a large number (one per timestamp) of other structures (e.g., SB-trees [35], R-trees [17]) in a space economical fashion. The embedding, however, is not always “linear”; for example, as mentioned in Section 2.1, although the SB-tree [35] has a linear space complexity, the MVSB-tree [38] requires  $O(n \log_B n)$  space.

The solutions developed in this paper will utilize an MVB-tree to perform “timestamp floor search”. Formally, let  $DB$  be a set of  $N$  temporal objects. Given a timestamp  $q_t$  and a value  $q_v$ , a *timestamp floor query*, denoted as  $TF(q_t, q_v)$ , retrieves the object whose key is the largest, among all the objects in  $DB$  that (i) are alive at time  $q_t$ , and (ii) their keys are at most  $q_v$ . For example, if  $DB$  consists of the 7 objects in Figure 1, then  $TF(6, 2.3)$  finds  $o_4$ . To understand why, notice that only objects  $o_4, o_5$  satisfy conditions (i) and (ii); between them,  $o_4$  has a larger key. Indexing  $DB$  with an MVB-tree (occupying  $O(n)$  space), we can process any timestamp floor query in  $O(\log_B n)$  I/Os [5, 34].

### 3 Reduction to Half-Ranged Search

We first introduce a special version of approximate timestamp range count search:

**Definition 4.** Let  $q$  be an approximate timestamp range count query formulated in Definition 3. We say that  $q$  is **half-ranged** if  $q_{k+}$  equals  $-\infty$ , that is,  $\vec{q}_k = (-\infty, q_{k-}]$ .

The next theorem shows that a normal query with approximation ratio  $\varepsilon$  can be converted to two half-ranged queries  $q$  whose error is less than  $\frac{1}{2\varepsilon} + \frac{\varepsilon}{2} \cdot N_{alive}(q_t)$ .

**Theorem 1.** Denote  $\delta$  as an infinitely small positive value. Given any query  $q$  of Definition 3 with approximation ratio  $\varepsilon$ , we construct two half-ranged queries  $q_1$  and  $q_2$ , by setting  $\vec{q}_{1k} = (-\infty, q_{k+} - \delta]$ ,  $\vec{q}_{2k} = (-\infty, q_{k-}]$ , and  $q_{1t} = q_{2t} = q_t$ . Let  $v_1$  and  $v_2$  be the approximate results of  $q_1$  and  $q_2$  respectively, with error less than  $\frac{1}{2\varepsilon} + \frac{\varepsilon}{2} \cdot N_{alive}(q_t)$ . Then,  $v_2 - v_1$  is a correct answer for the original query  $q$ .

*Proof.* The construction of  $q_1$  and  $q_2$  ensures  $QS(q_t, \vec{q}_k) = QS(q_{2t}, \vec{q}_{2k}) - QS(q_{1t}, \vec{q}_{1k})$ . To prove the lemma, we aim at establishing

$$\left| v_2 - v_1 - |QS(q_t, \vec{q}_k)| \right| \leq \varepsilon + \frac{1}{\varepsilon} \cdot N_{alive}(q_t) \quad (3)$$

By the definition of  $v_1$ :

$$\left| v_1 - |QS(q_{1t}, \vec{q}_{1k})| \right| < \frac{1}{2\varepsilon} + \frac{\varepsilon}{2} \cdot N_{alive}(q_{1t})$$

where  $QS(q_{1t}, \vec{q}_{1k})$  is the set of objects  $o$  alive at  $q_{1t}$  whose  $o.key$  falls in  $\vec{q}_{1k}$ , and  $N_{alive}(q_{1t})$  is the number of objects alive at  $q_{1t}$ . The equation leads to

$$v_1 > |QS(q_{1t}, \vec{q}_{1k})| - \frac{1}{2\varepsilon} - \frac{\varepsilon}{2} \cdot N_{alive}(q_t). \quad (4)$$

Symbol	Meaning	Defined in (Section)
$DB$	The original temporal database	1.1
$B$	The disk page size	1.1
$N$ and $n$	The cardinality of $DB$ , and $N/B$ , respectively	1.1
$o.key$ and $[o.t_+, o.t_-]$	The key and lifespan of an object $o$ , respectively	1.1
$o.rank(t)$	The rank of an object $o$ at time $t$	4
$q_t$	The query timestamp	1.1
$\vec{q}_k = [q_{k+}, q_{k-}]$	The key range of a query ( $q_{k+} = \infty$ if $q$ is half-ranged)	1.1
$QS(q_t, \vec{q}_k)$	The set of objects $o \in DB$ alive at $q_t$ satisfying $o.key \in \vec{q}_k$	1.1
$TF(t, k)$	A timestamp floor query at time $t$ with query key $k$	2.3
$N_{alive}(t)$	The number of objects in $DB$ alive at time $t$	1.1
$N_{alive}$	The average number of objects alive at an anchor timestamp	5.4
$\varepsilon$ and $\varepsilon_{hr}$	The approximation ratio, and its half, respectively	1.1 and 3
$S_{anchor}$	The set of anchor segments alive at the sweeping line	5.1
$seg.key$ and $[seg.t_+, seg.t_-]$	The key and lifespan of an anchor segment $seg$ , respectively	5.2
$seg.rank$	The (constant) associated rank of an anchor segment $seg$	5.2

Table 1: Frequently used symbols

Similarly, for  $v_2$ , it holds that

$$\left| v_2 - |QS(q_{2t}, \vec{q}_{2k})| \right| < \frac{1}{2\varepsilon} + \frac{\varepsilon}{2} \cdot N_{alive}(q_{2t})$$

which results in

$$v_2 < |QS(q_{2t}, \vec{q}_{2k})| + \frac{1}{2\varepsilon} + \frac{\varepsilon}{2} \cdot N_{alive}(q_t). \quad (5)$$

From Inequalities 4 and 5, we know

$$\begin{aligned} v_2 - v_1 &< |QS(q_t, \vec{q}_{2k})| - |QS(q_t, \vec{q}_{1k})| + \frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t) \\ &= |QS(q_t, \vec{q}_k)| + \frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t) \end{aligned}$$

Hence, we know  $v_2 - v_1 - |QS(q_t, \vec{q}_k)| \leq \frac{1}{\varepsilon} + \varepsilon \cdot N_{alive}(q_t)$ . By similar reasoning, we can also obtain  $v_2 - v_1 - |QS(q_t, \vec{q}_k)| \geq -\frac{1}{\varepsilon} - \varepsilon \cdot N_{alive}(q_t)$ . Therefore,  $v_2 - v_1$  is a correct result of  $q$ .  $\square$

As an example, consider an approximate timestamp range count query  $q$  with approximation ratio 0.2,  $q_t =$  December 31, 2005, and  $\vec{q}_k = [\$100k, \$500k]$ . We can correctly solve  $q$ , as long as there exists a way to answer half-ranged queries at time  $q_t$  with maximum error less than  $\frac{1}{0.4} + 0.1 \cdot N_{alive}(q_t)$ . Specifically, we can construct two such queries  $q_1$  and  $q_2$  with  $q_{1k} = (-\infty, \$100k - \delta]$ ,  $q_{2k} = (-\infty, \$500k]$ , and  $q_{1t} = q_{2t} =$  December 31, 2005, where  $\delta$  is as defined in Theorem 1. If  $v_1$  and  $v_2$  are respectively the approximate answers for  $q_1$  and  $q_2$  satisfying the precision requirement mentioned earlier, then  $v_2 - v_1$  is a correct result for  $q$ .

Clearly, the reduction in Theorem 1 requires  $O(1)$  time to construct the result of  $q$  from those of  $q_1$  and  $q_2$ . Therefore, towards settling Problem 1, it suffices to solve only:

**Problem 2.** *Preprocess  $DB$  into a space-economical structure such that any half-ranged query  $q$  can be answered in a small number of I/Os, with maximum error lower than  $\frac{1}{4\varepsilon_{hr}} + \varepsilon_{hr} \cdot N_{alive}(q_t)$ , where  $\varepsilon_{hr} = \varepsilon/2$ , and  $\varepsilon$  is the approximation ratio in Definition 3.*

We will tackle the above problem in three steps. First, assuming that  $DB$  is static, Sections 4 and 5 propose several methods that settle:

**Problem 3.** *Same as Problem 2, except that the maximum error of  $q$  must be lower than  $\varepsilon_{hr} \cdot N_{alive}(q_t)$ .*

These methods, in the worst case, may take up more space than the dataset itself. Hence, in Section 6 which also focuses on a static  $DB$ , we extend those solutions with a “zoning technique” to solve Problem 2, while limiting the space strictly to  $O(n)$ , and the query cost to  $O(\log_B n)$  I/Os. Finally, Section 7 extends our solutions to support data updates. Table 1 lists the symbols to be used frequently (some of them have not appeared so far, but will be introduced later).

## 4 The First Method

Our methodology converts timestamp range counting to timestamp floor search (explained in Section 2.3) on a set of selected “anchor segments”. In the sequel, we will illustrate the idea by presenting a method that solves Problem 3 using  $O(\frac{n}{\varepsilon_{hr}})$  space. For simplicity, the discussion in this section assumes that, if two objects are alive at the same timestamp, they must have different keys. This assumption will be removed later in Section 5.

**Definition 5.** *The rank of an object  $o$  at a timestamp  $t$  equals one more than the number of objects that are alive at time  $t$ , and their keys are smaller than  $o.key$ .*

We denote the rank of  $o$  at time  $t$  as  $o.rank(t)$ . To explain this concept, let us inspect the dataset in Figure 1. At timestamp 1, object  $o_1$  has rank 4, since 3 objects are alive at time 1, and possess keys lower than  $o_1.key = 3.5$ . At time 2, the rank of  $o_1$  becomes 5, due to the appearance of  $o_3$ , whose key is smaller than  $o_1.key$ . Similarly,  $o_1.rank(t)$  equals 3 and 4 when  $t$  distributes in  $[3, 4)$  and  $[4, 5)$ , respectively, while  $o_1.rank(t)$  is undefined at any  $t \notin [1, 5)$ .

Consider the half-ranged query  $q$  in Figure 1 (i.e.,  $q_t = 6$  and  $\vec{q}_k = (-\infty, 3.5]$ ), whose real answer equals 3 (since  $QS(q_t, \vec{q}_k) = \{o_3, o_4, o_5\}$ ). Note that the value 3 is also the rank of  $o_3$  at  $q_t = 6$ . In turn,  $o_3$  is the object with the largest key below  $q_{k+1} = 3.5$  among all the objects alive at time 6; namely,  $o_3$  is the result of  $TF(6, 3.5)$  — a timestamp floor query at time  $q_t$  with key  $q_{k+1}$ . In general, for any half-ranged query  $q$

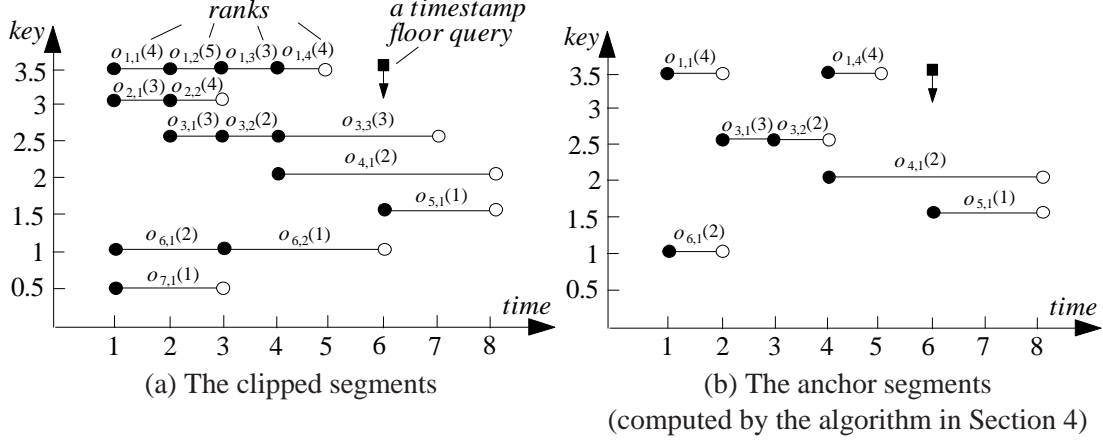


Figure 2: Reduction from a half-range query to timestamp floor search

with parameters  $q_t$  and  $(-\infty, q_{k-}]$ , its result always equals the rank at time  $q_t$  of the object retrieved by a timestamp floor query  $TF(q_t, q_{k-})$ . As a special case, if  $TF(q_t, q_{k-})$  returns  $\emptyset$ , the answer for  $q$  is 0.

Motivated by this, we clip each object into several segments such that, during the lifespan of each *clipped segment*, the rank of the object remains fixed; in the sequel, we refer to the rank as the *associated rank* of the clipped segment. For instance,  $o_1$  in Figure 1 generates 4 clipped segments  $o_{1,1}, o_{1,2}, \dots, o_{1,4}$  shown in Figure 2a, during which the rank of  $o_1$  equals 4, 5, 3, and 4, respectively. As another example,  $o_4$  spawns only a single clipped segment  $o_{4,1}$ , since the rank of  $o_4$  remains 2 during its entire lifespan. After obtaining the clipped segments (of all objects), we index them using an MVB-tree, and store with each segment its associated rank. Given a half-ranged query  $q$ , we perform timestamp floor search  $TF(q_t, q_{k-})$  using the MVB-tree, and return the rank associated with the retrieved clipped segment as the result for  $q$  (e.g.,  $TF(6, 3.5)$ , as shown in Figure 2a, retrieves  $o_{3,3}$ ).

In the worst case, the total number of clipped segments is quadratic to the dataset cardinality  $N$ , rendering space consumption  $O(N^2/B)$  of the MVB-tree (although the processing time of a half-ranged query has satisfactory complexity:  $O(\log_B N^2/B) = O(\log_B n)$ ). Fortunately, it is sufficient to keep only a subset of clipped segments, called the *anchor segments*, for approximate counting.

**Definition 6.** The set  $S$  of **anchor segments** can be regarded as a set of temporal objects (see Definition 1) with the following properties. (i) Each of them has an **associated rank**. (ii) Given any half-ranged query  $q$  with range  $\vec{q}_k = (-\infty, q_{k-}]$ , timestamp  $q_t$ , and approximation ratio  $\varepsilon_{hr}$ , we construct a timestamp floor query  $TS(q_t, q_{k-})$ . Let  $r$  be the result of  $q$  on the original dataset, and  $r'$  the rank of the anchor segment retrieved by performing  $TS(q_t, q_{k-})$  on  $S$ . Then, it always holds that  $0 < r - r' \leq \lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil - 1$ .

To illustrate the concept, assume that the approximation ratio  $\varepsilon_{hr}$  of half-ranged search equals 0.5 (strictly speaking, 0.5 is not an appropriate value for  $\varepsilon_{hr}$  since it implies a useless  $\varepsilon$  of 1; however, setting  $\varepsilon_{hr}$  to

0.5 allows us to avoid complex examples without loss of generality). We only need to store the clipped segments in Figure 2b, to provide correct approximate results. For example, consider the half-ranged query  $q$  in Figure 1 again. As before, it is transformed to floor search  $TF(6, 3.5)$ , which now retrieves  $o_{4,1}$  with rank 2; hence, we return 2 as the result for  $q$ . Remember that the actual answer of  $q$  is 3, so our result has error 1, which is acceptable since it does not exceed  $\varepsilon_{hr} \cdot N_{alive}(q_t) = 0.5 \times 3 = 1.5$ .

The segments in Figure 2b have a common property: their associated ranks are a multiple of  $\lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil$  at some timestamp  $t$ . For example, at  $t = 2$ ,  $\lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil = \lceil 0.5 \times 5 \rceil = 3$ ; therefore,  $o_{3,1}$  is the only anchor segment at this timestamp. At  $t = 4$ , on the other hand,  $\lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil = \lceil 0.5 \times 4 \rceil = 2$ ; so we take  $o_{4,1}$  and  $o_{1,4}$  as anchor segments. The property has the following consequence. Given a half-ranged query  $q$ , let us perform the corresponding timestamp floor search  $TS(q_t, q_{k-1})$  on (i) the full set of clipped segments, and (ii) only the set of anchor segments, respectively. If we denote  $r$  and  $r'$  as the ranks of the segments retrieved from (i) and (ii), respectively, it always holds that  $0 < r - r' \leq \lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil - 1 < \varepsilon_{hr} \cdot N_{alive}(t)$ . Therefore, we can directly return  $r'$  as an approximate answer for  $q$ , which has error less than  $\varepsilon_{hr} \cdot N_{alive}(t)$ . By Definition 6, the segments in Figure 2b constitute a set of anchor segments.

It is not hard to verify that the number of anchor segments is  $O(\frac{N}{\varepsilon_{hr}})$ . Observe that, at any timestamp  $t$ , there can be at most  $\lfloor 1/\varepsilon_{hr} \rfloor$  anchor segments (since  $O(N_{alive}(t))$  intervals are alive at time  $t$ , and the ranks of anchor segments with consecutive keys differ by  $\lceil \varepsilon_{hr} \cdot N_{alive}(t) \rceil$ ). On the other hand, a new anchor segment can be created only at a timestamp when an object is born or dies. The number of such timestamps is  $O(2N)$ ; therefore, there are  $O(2N \cdot 1/\varepsilon_{hr}) = O(\frac{N}{\varepsilon_{hr}})$  anchor segments. Thus, an MVB-tree indexing these anchor segments consumes  $O(\frac{n}{\varepsilon_{hr}})$  space, and answers a timestamp floor query (and hence, the original half-ranged count query) in  $O(\log_B \frac{n}{\varepsilon_{hr}})$  I/Os.

## 5 An Improved Method

Next, we will show that the number of anchor segments can be decreased to  $O(\min\{\frac{N}{\varepsilon_{hr}}, (\frac{1}{\varepsilon_{hr}})^2 \cdot N/\overline{N_{alive}}\})$ , where  $\overline{N_{alive}}$  is a positive integer, corresponding to the average number of live objects at an ‘‘anchor timestamp’’ (to be defined in Section 5.1). Accordingly, a query of Problem 3 can be answered by performing fewer I/Os using a structure occupying less space. To facilitate discussion, Sections 5.1 and 5.2 keep the assumption that no two objects have the same key at an identical timestamp. Section 5.3 extends our results to the general scenario without this assumption. Finally, Section 5.4 investigates the performance of the proposed algorithms.

### 5.1 The Rationale

Assuming  $\varepsilon_{hr} = 0.5$ , next we deploy the running example of Figure 1 to illustrate an improved way of computing anchor segments. First of all, we collect the set of starting and ending timestamps of all objects’

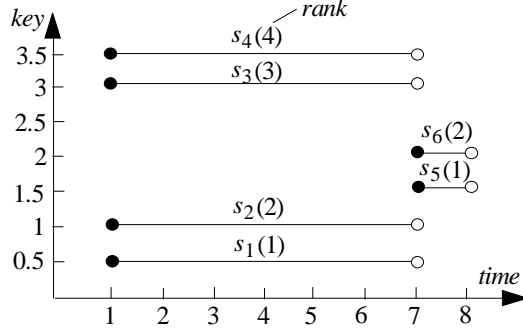


Figure 3: Anchor segments obtained by the algorithm in Section 5

lifespans (i.e., the set is  $\{1, 2, \dots, 8\}$ ) and call them the *event timestamps*. Equivalently, the set consists of the distinct timestamps of all the updates in the temporal stream of  $DB$ . Imagine a vertical sweeping line perpendicular to the time axis. Initially, the line is positioned at  $t = 0$ ; we will move it gradually towards right, and carry out a set of operations whenever the line reaches an event timestamp.

When the sweeping line hits  $t = 1$ , we retrieve all the objects whose current ranks (at time  $t$ ) are multiples of  $\lceil \varepsilon_{hr} \cdot N_{alive}(1)/2 \rceil = \lceil 0.5 \times 4/2 \rceil = 1$ , namely, all the objects  $o_1, o_2, o_6$ , and  $o_7$  alive at time 1 are retrieved. According to object  $o_7$ , we create an anchor segment  $s_1$ , represented with a triplet:

$$\begin{aligned} \{s_1.key = 0.5, & \quad \text{(Note: 0.5 is the key of } o_7 \text{ at time 1)} \\ s_1.rank = 1, & \quad \text{(Note: 1 is the rank of } o_7 \text{ at time 1)} \\ s_1.[t_-, t_+) = [1, *) \} \end{aligned}$$

indicating that  $s_1$  has a key 0.5, an associated rank 1 (note that both  $s_1.key$  and  $s_1.rank$  are *constants* at all timestamps), and a lifespan  $s_1.[t_-, t_+) = [1, *)$  (i.e., the ending time is currently unknown). Similarly, from  $o_6, o_2, o_1$  respectively, we produce another three anchor segments  $s_2 = \{1, 2, [1, *)\}$ ,  $s_3 = \{3, 3, [1, *)\}$ , and  $s_4 = \{3.5, 4, [1, *)\}$ , as demonstrated in Figure 3 (the part of the figure after time 1 will be explained later). These four anchor segments are incorporated into a set  $S_{anchor}$ , which, in general, contains all the anchor segments alive at the sweeping line.

Any half-ranged query  $q$  with  $q_t = 1$  can be correctly answered by  $S_{anchor}$ , leveraging the reduction to timestamp floor search in Section 4. Furthermore, for such queries,  $S_{anchor}$  ensures error of at most  $\lceil \varepsilon_{hr} \cdot N_{alive}(1)/2 \rceil = 1$ , which is *half* of the target precision requirement  $\lceil \varepsilon_{hr} \cdot N_{alive}(1) \rceil = 2$  of Problem 3. In other words, at time 1, we *take more anchor segments than necessary* (Figure 2b shows that only two segments would have been sufficient), the purpose of which is to postpone the creation of new segments as much as possible, because the same  $S_{anchor}$  may still provide acceptable answers for queries at many subsequent timestamps.

Continuing our example, at the next event timestamp 2, we do not generate any new anchor segment, but simply use those segments in  $S_{anchor}$ , by allowing their lifespans to evolve. Consider a query  $q$  with  $q_t = 2$  and  $\vec{q}_k = [-\infty, 2.5]$ , whose real answer is 3 (since  $QS(q_t, \vec{q}_k) = \{o_3, o_6, o_7\}$ ).  $S_{anchor}$ , instead, provides an answer 2 (equal to the rank of the anchor segment  $s_2$  retrieved by  $TF(2, 2.5)$ ), whose error 1 is indeed smaller than  $\varepsilon_{hr} \cdot N_{alive}(2) = 0.5 \times 5 = 2.5$ . In fact, it is easy to verify that all queries at time 2 can be correctly solved from  $S_{anchor}$ .

By this reasoning, at each event timestamp  $t$ , we check whether the current  $S_{anchor}$  can support all queries at time  $t$ ; if yes, no new anchor segment is spawned, and  $S_{anchor}$  remains valid. In the dataset of Figure 1, the first timestamp at which  $S_{anchor}$  becomes invalid is 7. To understand the invalidation, let us examine a query  $q$  with  $q_t = 7$  and  $\vec{q}_k = [-\infty, 3.5]$ , whose real answer is 2 (notice that  $QS(q_t, \vec{q}_k) = \{o_4, o_5\}$ ). Imagine that  $s_1, s_2, \dots, s_4$  were still employed to answer  $q$ ; the result would be 4, i.e., the rank of  $s_4$ . In this case, the error  $4 - 2 = 2$  is larger than  $\varepsilon_{hr} \cdot N_{alive}(7) = 0.5 \times 2 = 1$ .

To fix the problem, we first terminate the lifespans of the segments in  $S_{anchor}$  at time 7, and remove them from  $S_{anchor}$  altogether. Then, we compute a new  $S_{anchor}$ , by finding all the live objects  $o$  at time 7 whose  $o.rank(7)$  is a multiple of  $\lceil \varepsilon_{hr} \cdot N_{alive}(7)/2 \rceil = \lceil 0.5 \times 2/2 \rceil = 1$ . Thus, both  $o_4$  and  $o_5$  are retrieved; accordingly, two new anchor segments are placed in  $S_{anchor}$ :  $s_5 = \{1.5, 1, [7, *]\}$  and  $s_6 = \{2, 2, [7, *]\}$ , where 1.5 and 2 are  $o_4.key$  and  $o_5.key$ , respectively. Finally, since 8 is the last timestamp of the history, the lifespans of  $s_5$  and  $s_6$  are terminated at 8, which completes the derivation of the segments in Figure 3. We refer to timestamps 1 and 7, when anchor segments are born, as the *anchor timestamps*.

We point out that a segment in Figure 3 may not be equivalent to any clipped segment in Figure 2a (actually, a segment in Figure 3 such as  $s_4$  may still be alive even after its spawning object  $o_1$  has died). However, such equivalence is really not compulsory; any segment in the key-time space can be an anchor segment, as long as it helps solving queries. Furthermore, it is not necessary to worry about timestamps  $t$  that are not an event timestamp, because no object can incur a rank change at such a  $t$ .

## 5.2 The Algorithm

We are ready to explain the formal algorithm of generating the anchor segments for arbitrary  $DB$ , based on the rationale discussed in Section 5.1. The input of the algorithm is the temporal stream of  $DB$ , and the output is another temporal stream that decides the anchor segments, and can be used directly to construct an MVB-tree. At a high level, the algorithm follows a simple plane sweep framework, as presented in Figure 4, where the words in small-cap fonts indicate procedures that will be explained shortly.

Specifically, after some initialization work at Lines 1-2, the algorithm processes every event timestamp in ascending order (i.e., moving the sweeping line towards the positive direction of the time axis). To handle

Algorithm GENERATE-ANCHOR-SEGMENT

/\* OUTPUT: A temporal stream that determines a set of anchor segments. \*/

1.  $S_{anchor} = \emptyset$
2. INITIALIZATION
3. for the next event timestamp  $CT$  (in ascending order)
4.     STRUCTURE-UPDATE
5.     if CHECK-VALIDITY = FALSE /\*  $CT$  is an anchor timestamp when the if-condition holds \*/
6.         for every segment  $seg \in S_{anchor}$
7.             write to the output temporal stream:  $\langle LD, seg.key, CT \rangle$
8.          $S_{anchor} = \text{NEW-ANCHORS}$  /\* the computed  $S_{anchor}$  should be able to support any half-ranged query at time  $CT$  with approximation ratio  $\varepsilon_{hr}/2$  \*/
9.         for every segment  $seg \in S_{anchor}$
10.             write to the output temporal stream:  $\langle INS, seg.key, CT \rangle$

Figure 4: The high-level framework of computing anchor segments

an event timestamp  $CT$ , we first carry out the necessary operations to maintain some structures that capture the “sweeping state” (Line 4). Then, Line 5 checks whether the current  $S_{anchor}$  can still support all the queries at time  $CT$ . If the answer is negative, Lines 6-7 terminate the lifespans of the segments in  $S_{anchor}$  at time  $CT$ , write the corresponding logical deletions to the disk, and delete them from  $S_{anchor}$ . Next, Line 8 re-computes an  $S_{anchor}$  from the objects alive at time  $CT$ . Finally, Lines 9-10 append the insertions to the output temporal stream that start the lifespans of the new anchor segments.

In the sequel, we elaborate the details of the algorithm, starting with NEW-ANCHORS and CHECK-VALIDITY, since they determine the other procedures.

**Procedure NEW-ANCHORS.** Let  $N_{alive}(CT)$  be the number of objects alive at the current event timestamp  $CT$ . Among them, NEW-ANCHORS identifies  $m = \lfloor 2/\varepsilon_{hr} \rfloor$  objects  $o_1, o_2, \dots, o_m$  whose ranks (at time  $CT$ ) equal  $\lceil \varepsilon_{hr}/2 \cdot N_{alive}(CT) \rceil, 2 \cdot \lceil \varepsilon_{hr}/2 \cdot N_{alive}(CT) \rceil, \dots, m \cdot \lceil \varepsilon_{hr}/2 \cdot N_{alive}(CT) \rceil$ , respectively. Then, from object  $o_i$  ( $1 \leq i \leq m$ ), NEW-ANCHORS creates an anchor segment  $seg_i$  in  $S_{anchor}$ :

$$\begin{aligned} \{ & seg_i.key = o_i.key, \\ & seg_i.rank = i \cdot \lceil \varepsilon_{hr}/2 \cdot N_{alive}(CT) \rceil, \\ & seg_i.[t_+, t_-] = [CT, *) \}. \end{aligned}$$

(the above triplet generalizes the triplet in Section 5.1). For any query at time  $CT$ , these  $m$  anchor segments provide an answer whose maximum error is twice lower than permitted by Problem 3 (in Section 5.1, we explained this after  $S_{anchor}$  was computed at time 1).

**Procedure CHECK-VALIDITY.** To inspect the validity of  $S_{anchor}$  at the current event time  $CT$ , CHECK-

VALIDITY needs to obtain, for every segment  $seg_i \in S_{anchor}$  ( $1 \leq i \leq m = \lfloor 2/\varepsilon_{hr} \rfloor$ ), a value  $v_i$ , which equals the number of objects alive at time  $CT$  whose keys are less than  $seg_i.key$ . Then,  $S_{anchor}$  is invalid if any of the following *invalidation conditions* is satisfied:

1.  $v_1 \geq \varepsilon_{hr} \cdot N_{alive}(CT)$ ;
2. for any  $i \in [2, m]$ ,  $|v_i - seg_{i-1}.rank| \geq \varepsilon_{hr} \cdot N_{alive}(CT)$ ;
3.  $|N_{alive}(CT) - seg_m.rank| \geq \varepsilon_{hr} \cdot N_{alive}(CT)$ .

To understand Condition 1 (or 2), note that  $v_1$  (or  $v_i$ ) is essentially the *real result* for a half-ranged query  $q$  with  $q_t = CT$  and  $q_{k+}$  being infinitesimally smaller than  $seg_1.key$  (or  $seg_i.key$ ). On the other hand, if we use the current  $S_{anchor}$  to answer  $q$ , then the approximate result equals 0 (or  $seg_{i-1}.rank$ ). Hence, if Condition 1 (or 2) holds, it means that the approximation has error at least  $\varepsilon_{hr} \cdot N_{alive}(t)$ , which violates the requirement of Problem 3. Similarly, Condition 3 implies that  $S_{anchor}$  is not able to correctly support a query  $q$  with  $q_t = CT$  and  $q_{k+} = \infty$ .

To illustrate CHECK-VALIDITY with an example, let us review the situation at timestamp  $CT = 7$  in Figure 3, when  $S_{anchor}$  includes four segments  $seg_1 = s_1$ ,  $seg_2 = s_2$ ,  $seg_3 = s_3$ ,  $seg_4 = s_4$ , whose associated ranks are 1 ( $= seg_1.rank$ ), 2, 3, and 4, respectively, with respect to the dataset in Figure 1 ( $\varepsilon_{hr} = 0.5$ ). At time 7, the values of  $v_1$ ,  $v_2$ ,  $v_3$ , and  $v_4$  are 0, 0, 2, 2, respectively. Since  $N_{alive}(7) = 2$ , Condition 2 is breached, as  $|v_2 - seg_1.rank| = |0 - 1| \geq \varepsilon_{hr} \cdot N_{alive}(7) = 1$ . Furthermore, Condition 3 is also violated because  $|N_{alive}(7) - seg_m.rank| = |2 - 4| \geq 1$ . This confirms our decision in Section 5.2 that  $S_{anchor}$  is invalid at time 7.

**Sweeping-State Structure.** NEW-ANCHORS and CHECK-VALIDITY are built upon two similar, but different, operations on the set of objects alive at the current event time  $CT$ . Specifically, NEW-ANCHORS requires FIND-KEY( $r$ ), which, given a rank  $r$ , retrieves the key of the object whose rank at time  $CT$  equals  $r$ . On the other hand, CHECK-VALIDITY demands FIND-RANK( $k$ ), which, given a key  $k$ , retrieves the number of objects whose keys at time  $CT$  are lower than  $k$ . The goal of the sweeping-state structure is to support both operations efficiently.

We choose the structure to be a modified B-tree that indexes the keys of the objects alive at time  $CT$ . Figure 5 shows an example, assuming nine live objects with keys 1, 2, ..., 9, respectively. Compared to a conventional B-tree, the modification lies in the fact that every intermediate entry is augmented with an aggregate, which counts the number of leaf entries in the subtree. For instance, the first entry “1(2)” in node  $E$  indicates that the routing key of the entry equals 1, and there are two leaf entries in node  $A$ .

A FIND-KEY( $r$ ) request can be processed in  $O(\log_B N_{alive}(CT))$  I/Os, where  $N_{alive}(CT)$  is the number

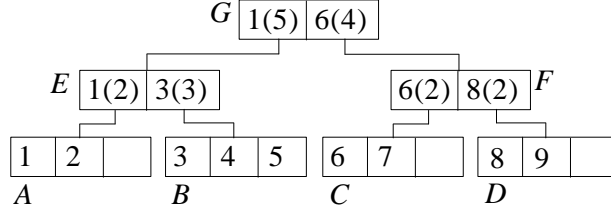


Figure 5: The structure recording the sweeping state

of live objects at time  $CT$ . Assume that we employ the tree in Figure 5 to perform  $\text{FIND-KEY}(7)$ . The operation starts by retrieving the root  $G$ . Since only 5 (i.e.,  $< 7$ ) objects exist in the left subtree of  $G$ , the key we are searching for must lie in the right subtree. Hence, we load node  $F$  into memory, and assert that the target key must appear in the left subtree of  $F$  (notice that any key in the right subtree of  $F$  has a rank larger than  $5 + 2 = 7$ ). Thus, we access node  $C$ , and return the key 7 (i.e., the second smallest in the node). Based on this strategy, any  $\text{FIND-KEY}(r)$  can be solved by accessing a single path of the B-tree.

Processing  $\text{FIND-RANK}(k)$  incurs the same overhead. Specifically, the operation accesses the same nodes as looking for the key  $k$  in a normal B-tree. The only difference is that, at an intermediate node  $X$ , whenever we descend an entry  $e$ , we must add all the counters in  $X$  on the left of  $e$  to our temporary result. For instance, imagine that we need to answer  $\text{FIND-RANK}(7.5)$ . At the beginning, our temporary result  $r$  equals 0. At the root  $G$ , the operation follows the right subtree of  $G$  (since the routing key of the second entry in  $G$  is less than 7.5). Therefore, we add counter 5 of the first entry to  $r$ . At node  $F$ , we follow the left subtree; hence,  $r$  remains unchanged. Since two keys in the leaf node  $C$  are below 7.5, we add 2 to  $r$ , and return the resulting  $r = 7$ .

**Procedures INITIALIZATION and STRUCTURE-UPDATE.** These procedures are mainly responsible for initializing and maintaining the B-tree and the value of  $N_{alive}(t)$ , respectively. Specifically,  $\text{INITIALIZATION}$  simply creates an empty tree, and sets  $N_{alive}(t)$  to 0. On the other hand, at each event timestamp  $t$ ,  $\text{STRUCTURE-UPDATE}$  removes the keys of all the objects that die at time  $t$ , inserts the keys of those born at this timestamp, and updates  $N_{alive}(t)$  according to the numbers of removed and inserted keys. Obviously, inserting/deleting a key takes  $O(\log_B N_{alive}(t)/B)$  I/Os. Finally,  $\text{INITIALIZATION}$  also collects all the event timestamps, and sorts them in ascending order. In practice, the sorting could be avoided, because, under the transaction time model, objects are born and die in ascending order by default.

### 5.3 Eliminating the Distinct-Key Assumption

So far our discussion has been based on an assumption: each pair of objects alive at a timestamp should have different keys. This assumption may be invalid in practice. For example, two accounts in a bank may have an equivalent balance at the same time. Fortunately, our technique can be easily extended to the general

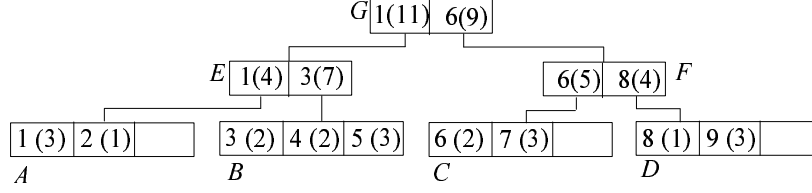


Figure 6: The sweeping-state structure without the distinct-key assumption

scenario where this assumption does not hold, without affecting the asymptotic time and space performance. In the sequel, we elaborate the necessary changes to the algorithm provided in Section 5.2.

**Procedure NEW-ANCHORS.** Recall that this procedure creates a new set  $S_{anchor}$  of anchor segments at the current event timestamp  $CT$ . For this purpose, we adopt a modified version of  $\text{FIND-KEY}(r)$ , which returns the smallest key  $k$  such that at least  $r$  objects alive at  $CT$  have search keys at at most  $k$ . Specifically,  $k$  satisfies all the following conditions:

- at least an object  $o$  alive at  $CT$  has a key equal to  $k$ ;
- the rank  $o.rank(CT)$  of  $o$  at time  $CT$  is at most  $r$  (recall that  $o.rank(CT) - 1$  is the number of objects alive at  $CT$  whose keys are lower than  $o.key$ );
- $o.rank(CT) + s$  is strictly larger than  $r$ , where  $s$  is the number of objects that are alive at time  $CT$ , and have  $k$  as their keys (obviously, these objects have an identical rank at time  $CT$ ).

As a side product,  $\text{FIND-KEY}(r)$  also reports a value  $r' = o.rank(CT) + s - 1$ . Notice that  $r'$  equals precisely the number of objects that are alive at time  $CT$ , and their search keys are at most  $k$ .

Equipped with such an operation, **NEW-ANCHORS** generates the anchor segments as follows. First, it sets  $r$  to  $\lceil \varepsilon/2 \cdot N_{alive}(CT) \rceil$ , and invokes  $\text{FIND-KEY}(r)$ . Let  $k$  and  $r'$  be the values returned by the function. Then, an anchor interval  $\{k, r', [CT, *)\}$  is added to  $S_{anchor}$ . After this, **NEW-ANCHORS** repeats the above process, using a new  $r = r' + \lceil \varepsilon/2 \cdot N_{alive}(CT) \rceil$ , until the new  $r$  is greater than  $N_{alive}(CT)$ .

**Procedure CHECK-VALIDITY.** Let  $m$  be the number anchor segments created by the previous execution of **NEW-ANCHORS** ( $m \leq \lfloor 2/\varepsilon_{hr} \rfloor$ ). Using  $seg_i$  ( $1 \leq i \leq m$ ) to denote the segment with the  $i$ -th lowest key, the procedure **CHECK-VALIDITY** in Section 5.2 can be directly applied.

**Sweeping-State Structure.** The structure is again a B-tree indexing the keys of the objects alive at time  $CT$ . Each leaf entry contains a distinct key, which is associated with a counter, equal to the number of currently live objects carrying that key. Each intermediate entry is also augmented with a counter, which equals the sum of the counters of all the leaf entries in the subtree. Figure 6 demonstrates an example. The first entry in

node  $A$  indicates that 3 objects alive at time  $t$  have a key 1, whereas the counter 11 in the first entry of node  $G$  is the sum of the counters of all the leaf entries in nodes  $A$  and  $B$ .

FIND-KEY( $r$ ) can still be settled by accessing a single path of the B-tree, costing  $O(\log_B N_{alive}(t))$  I/Os. Remember that this operation eventually returns two values  $k$  and  $r'$ . Initially, it sets  $r'$  to 0, and loads the root. In general, at an intermediate node  $X$ , FIND-KEY examines its entries from left to right. Given an entry  $e$ , the operation checks whether the sum of its counter and the current  $r'$  exceeds  $r$ . If no,  $r'$  is increased by the counter of  $e$ , and FIND-KEY inspects the next entry. Otherwise, FIND-KEY accesses the child node of  $e$ , and carries out the above steps recursively, until reaching a leaf node  $X'$ . Given  $X'$ , FIND-KEY scans its entries from left to right, and adds the counter of an entry to  $r'$ , once the entry has been scanned. The scanning stops, as soon as  $r'$  is larger than or equal to  $r$ . The final  $k$  is the key of the last scanned entry, and the final  $r'$  is simply the current value of this variable. In Figure 6, for example, FIND-KEY(14) visits nodes  $G$ ,  $F$ ,  $C$ , and produces  $k = 7$ ,  $r' = 16$ .

FIND-RANK( $k$ ) is performed in the same way as described in Section 5.2. The only difference is that, after reaching a leaf node, we increase  $r$  by the sum of the counters of all entries whose keys are bounded by  $k$ .

**Procedures STRUCTURE-UPDATE.** The procedure is identical to the one in Section 5.2, except that extra efforts are needed to maintain the counters in leaf nodes. Specifically, when an object is born with a key  $k$ , we first check whether there exists a leaf entry in the B-tree having the same  $k$ . If yes, the entry's counter is added by 1, and no new entry is inserted; otherwise, a new entry is added with key  $k$  and counter 1. When an object with key  $k$  dies, we first decrease the counter of the leaf entry with that key. If the counter drops to 0, the entry is removed from the tree.

## 5.4 Analysis

We proceed to prove the asymptotical space consumption and query performance of the solution in Section 5.3. For this purpose, we use  $T$  to represent the total number of anchor timestamps, which are denoted as  $t_1, t_2, \dots, t_T$ , respectively (recall that, if an anchor segment is born at time  $t$ , then  $t$  is an anchor timestamp). Also, let  $\Delta_i$  ( $2 \leq i \leq T$ ) be the total number of updates in the period of  $(t_{i-1}, t_i]$  (e.g.,  $\Delta_i = 3$  if two objects are inserted and one is logically deleted). We first establish a result that reveals an important relationship between  $\Delta$  and  $N_{alive}$ :

**Lemma 1.** For any  $i \in [2, T]$ ,  $\Delta_i \geq \varepsilon_{hr} \cdot N_{alive}(t_i) - \varepsilon_{hr} \cdot N_{alive}(t_{i-1})/2$ .

*Proof.* We prove the lemma only for  $i = 2$ , because the discussion directly extends to other values of  $i$  as well. Let  $S_{anchor}$  be the set of anchor segments computed at time  $t_1$ . Remember that the creation of  $t_2$  is because  $S_{anchor}$  becomes invalid at time  $t_2$ , that is, at least one of the three invalidation conditions (referred

to as Condition 1, 2, and 3 in the sequel) in Section 5.2 is satisfied at  $t = t_2$ . We distinguish three cases.

Case 1 [Condition 1 holds]: At time  $t_1$ , due to the way  $S_{anchor}$  is computed, there are  $v'_1 = \lceil \varepsilon_{hr}/2 \cdot N_{alive}(t_1) \rceil - 1$  objects whose keys are lower than  $seg_1.key$ . At time  $t_2$ ,  $v_1$  objects have keys smaller than  $seg_1.key$ ; therefore:

$$\Delta_2 \geq |v_1 - v'_1| \quad (6)$$

Note that  $v'_1 \geq 0$ ; hence<sup>3</sup>, the right hand side of the above inequality

$$\begin{aligned} &\geq |v_1| - v'_1 \\ \text{(By Condition 1)} &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - (\lceil \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \rceil - 1) \\ &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \end{aligned} \quad (7)$$

Case 2 [Condition 2 holds]: At time  $t_1$ , there are  $v'_i = seg_{i-1}.rank + \lceil \varepsilon_{hr}/2 \cdot N_{alive}(t_1) \rceil - 1$  objects whose keys are lower than  $seg_i.key$ . Thus,

$$\begin{aligned} \Delta_2 &\geq |v_i - v'_i| \\ &= |v_i - seg_{i-1}.rank - (\lceil \varepsilon_{hr}/2 \cdot N_{alive}(t_1) \rceil - 1)| \end{aligned}$$

Since  $\lceil \varepsilon_{hr}/2 \cdot N_{alive}(t_1) \rceil - 1 \geq 0$ , the right hand side of the above inequality

$$\begin{aligned} &\geq |v_i - seg_{i-1}.rank| - (\lceil \varepsilon_{hr}/2 \cdot N_{alive}(t_1) \rceil - 1) \\ \text{(By Condition 2)} &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - (\lceil \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \rceil - 1) \\ &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \end{aligned} \quad (8)$$

Case 3 [Condition 3 holds]: Clearly,

$$\begin{aligned} \Delta_2 &\geq |N_{alive}(t_2) - N_{alive}(t_1)| \\ &= |(N_{alive}(t_2) - seg_m.rank) - (N_{alive}(t_1) - seg_m.rank)| \end{aligned}$$

As  $0 \leq N_{alive}(t_1) - seg_m.rank < \varepsilon_{hr} \cdot N_{alive}(t_1)/2$ , the right hand side of the above inequality

$$\begin{aligned} &\geq |(N_{alive}(t_2) - seg_m.rank)| - (N_{alive}(t_1) - seg_m.rank) \\ \text{(By Condition 3)} &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \end{aligned} \quad (9)$$

According to Inequalities 7, 8, and 9, the lemma holds in all cases, thus completing the proof.  $\square$

<sup>3</sup>In general, given any two numbers  $a$  and  $b$  such that  $b \geq 0$ , it holds that  $|a - b| \geq |a| - b$ .

**Lemma 2.**  $T = O(\min\{N, \frac{1}{\varepsilon_{hr}} \cdot N/\overline{N_{alive}}\})$ , where  $\overline{N_{alive}}$  is the average number of live objects at an anchor timestamp, namely,  $\overline{N_{alive}} = \frac{1}{T} \sum_{i=1}^T N_{alive}(t_i)$ .

*Proof.*  $T$  is obviously  $O(N)$ , because the number of anchor timestamps is bounded by the number of event timestamps. In the sequel, we will prove  $T = O(\frac{1}{\varepsilon_{hr}} \cdot N/\overline{N_{alive}})$ .

Lemma 1 indicates that:

$$\begin{aligned} \Delta_2 &\geq \varepsilon_{hr} \cdot N_{alive}(t_2) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \\ \Delta_3 &\geq \varepsilon_{hr} \cdot N_{alive}(t_3) - \varepsilon_{hr} \cdot N_{alive}(t_2)/2 \\ &\dots \\ \Delta_T &\geq \varepsilon_{hr} \cdot N_{alive}(t_T) - \varepsilon_{hr} \cdot N_{alive}(t_{T-1})/2 \end{aligned}$$

Summing up the left and right hand sides of these inequalities, we obtain:

$$\begin{aligned} \sum_{i=2}^T \Delta_i &\geq \left( \frac{\varepsilon_{hr}}{2} \cdot \sum_{i=2}^{T-1} N_{alive}(i) \right) + \varepsilon_{hr} \cdot N_{alive}(t_T) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \\ &\geq \left( \frac{\varepsilon_{hr}}{2} \cdot \sum_{i=2}^T N_{alive}(i) \right) - \varepsilon_{hr} \cdot N_{alive}(t_1)/2 \end{aligned}$$

Adding  $\varepsilon_{hr} \cdot N_{alive}(t_1)$  to both sides of the above leads to

$$\varepsilon_{hr} \cdot N_{alive}(t_1) + \sum_{i=2}^T \Delta_i \geq \frac{\varepsilon_{hr}}{2} \cdot \sum_{i=1}^T N_{alive}(i)$$

Since there are totally  $2N$  updates in the whole history, the left hand side of the above inequality is bounded by  $2N$ , resulting in

$$\begin{aligned} 2N &\geq \frac{\varepsilon_{hr}}{2} \cdot \sum_{i=1}^T N_{alive}(i) \\ &= \frac{\varepsilon_{hr}}{2} \cdot \overline{N_{alive}} \cdot T \end{aligned}$$

Therefore,  $T \leq \frac{4}{\varepsilon_{hr}} \cdot N/\overline{N_{alive}} = O(\frac{1}{\varepsilon_{hr}} \cdot N/\overline{N_{alive}})$ . □

At each anchor timestamp,  $\lfloor 2/\varepsilon_{hr} \rfloor = O(\frac{1}{\varepsilon_{hr}})$  anchor segments are created; therefore, the total number of anchor segments equals  $O(\frac{1}{\varepsilon_{hr}} \cdot T) = O(\min\{\frac{N}{\varepsilon_{hr}}, (\frac{1}{\varepsilon_{hr}})^2 \cdot N/\overline{N_{alive}}\})$ .

Combining the above result with the reduction to timestamp floor search in Section 4, we can preprocess all the anchor segments into an MVB-tree that consumes  $O(\min\{\frac{n}{\varepsilon_{hr}}, (\frac{1}{\varepsilon_{hr}})^2 \cdot n/\overline{N_{alive}}\})$  space, and answers any query in  $O(\min\{\log_B n, \log_B \frac{n}{\varepsilon_{hr}} - \log_B \overline{N_{alive}}\})$  I/Os. Since each event timestamp can be handled in  $O(\frac{1}{\varepsilon} \log_B n)$  I/Os, all the anchor segments can be generated in  $O(\frac{N}{\varepsilon} \log_B n)$  I/Os, after which an MVB-tree can be built in  $O(n \log_B n)$  time, leading to  $O(\frac{N}{\varepsilon} \log_B n)$  overall preprocessing cost.

## 6 The Zoning Technique

For Problem 3, we have found a solution that requires  $O(\min\{\frac{n}{\varepsilon_{hr}}, (\frac{1}{\varepsilon_{hr}})^2 \cdot n/\overline{N_{alive}}\})$  space, entails  $O(\frac{N}{\varepsilon} \log_B n)$  preprocessing overhead, and guarantees  $O(\min\{\log_B n, \log_B \frac{n}{\varepsilon_{hr}} - \log_B \overline{N_{alive}}\})$  query time. Although the solution also settles Problem 2 (which allows extra error of  $\frac{1}{4\varepsilon_{hr}}$  compared to Problem 3), the appearance of  $\frac{1}{\varepsilon_{hr}}$  in the complexities is a bit disturbing. For instance, given  $\varepsilon_{hr} = 5\%$ ,  $\frac{n}{\varepsilon_{hr}}$  looks like  $20n$ , namely, 20 times larger than the amount of space occupied by the database. Next, we show that it is possible to remove  $\varepsilon_{hr}$  completely, i.e., constraining the space cost to  $O(n)$ , the preprocessing overhead to  $O(N \log_B n)$ , and the query time to  $O(\log_B n)$ . Accordingly, the error bound needs to be slightly relaxed: from  $\varepsilon_{hr} \cdot N_{alive}(q_t)$  to  $\frac{1}{4\varepsilon_{hr}} + \varepsilon_{hr} \cdot N_{alive}(q_t)$ , as permitted by Problem 2.

We start with a simple, yet important observation:

**Lemma 3.** *If the number of event timestamps equals  $O(\varepsilon_{hr} \cdot N)$ , then the algorithm in Section 5.2 generates at most  $O(N)$  anchor segments; furthermore, the preprocessing cost is  $O(N \log_B n)$ .*

*Proof.* The algorithm of Figure 4 generates at most  $2/\varepsilon_{hr}$  anchor segments at each event timestamp. Hence, the maximum number of anchor segments equals  $O((1/\varepsilon_{hr}) \cdot (\varepsilon_{hr} \cdot N)) = O(N)$ . Since handling each anchor timestamp takes  $O(\frac{1}{\varepsilon_{hr}} \log_B n)$  I/O, the overall preprocessing overhead is  $O((\frac{1}{\varepsilon_{hr}} \log_B n) \cdot (\varepsilon_{hr} \cdot N)) = O(N \log_B n)$  I/Os.  $\square$

**Definition 7.** *A dataset is **synchronous** if the number of event timestamps equals  $O(\varepsilon_{hr} \cdot N)$ .*

For a reasonable  $\varepsilon_{hr}$  (e.g., 5%), many practical datasets are indeed synchronous by default. For example, in Application 1 of Section 1, every day (i.e., a timestamp) a bank must handle a huge number (sometimes millions) of deposits/withdrawals. Similarly, in Application 2, all the (thousands of) sensors report their temperature readings at the same intervals. In both cases, the number of insertions or logical deletions at a timestamp is significantly larger than  $1/\varepsilon_{hr}$ . When the original dataset  $DB$  is synchronous, as a corollary of Lemma 3, the solution of Section 5 occupies  $O(\min\{n, (\frac{1}{\varepsilon_{hr}})^2 \cdot n/\overline{N_{alive}}\})$  space, incurs  $O(N \log_B n)$  preprocessing cost, and answers any query in  $O(\min\{\log_B n, \log_B \frac{n}{\varepsilon_{hr}} - \log_B \overline{N_{alive}}\})$  I/Os.

If  $DB$  is not synchronous, we will transform it, based on a “zoning” technique, into a synchronous one  $DB^*$ :

**Definition 8.**  *$DB^*$  has the same cardinality as  $DB$ , and possesses the following property. Given any half-ranged query  $q$ , if we obtain an answer by applying the method in Section 5 to  $DB^*$ , the answer deviates from the true result for  $DB$  by less than  $\frac{1}{4\varepsilon_{hr}} + \varepsilon_{hr} \cdot N_{alive}(q_t)$ , where  $N_{alive}(q_t)$  is the number of objects in  $DB$  alive at  $q_t$ .*

In the sequel, we will elaborate the transformation algorithm, focusing on  $\varepsilon_{hr} < 0.5$ . In fact, for  $\varepsilon_{hr} \geq 0.5$ ,

$\frac{1}{\varepsilon_{hr}} \leq 2$ . Hence, our approach in the previous section consumes  $O(n/\overline{N_{alive}})$  space, requires  $O(N \log_B n)$  preprocessing time, and ensures  $O(\log_B n - \log_B \overline{N_{alive}})$  query cost.

Given the temporal stream of  $DB$ , the algorithm computes another temporal stream determining  $DB^*$ , and outputs the updates in this stream in ascending order of their timestamps. Specifically, we scan the  $DB$  stream in chronological order, and buffer its updates in a set  $BUF$ . The buffering is carried out in the unit of a timestamp; that is, once we decide to place an update at timestamp  $t$  in  $BUF$ , then all the updates at time  $t$  must also appear in  $BUF$ . After buffering all the updates at an event timestamp, we examine whether  $|BUF|$  is at least  $\lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor$ . If no, the algorithm proceeds to buffer all the updates at the next event timestamp, and then compares  $|BUF|$  with  $\lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor$  again. If yes,  $BUF$  overflows.

When an overflow occurs, we first create a *zone*  $z$ , which is a period  $[z.t_+, z.t_-]$  of time. In particular,  $z.t_+$  (or  $z.t_-$ ) equals the smallest (or largest) time of the updates in  $BUF$ . Then, we inspect each update in  $BUF$ , and generate an update in the stream of  $DB^*$  accordingly. Precisely, given an insertion  $\langle \text{INS}, o.key, t \rangle$  in  $BUF$ , we check whether  $t$  equals  $z.t_+$  or  $z.t_-$ , in either case the insertion is retained directly. If neither case holds, we change its timestamp  $t$  to  $z.t_+$ . Similarly, given a logical deletion  $\langle \text{LD}, o.key, t \rangle$  in  $BUF$ , we keep it intact, if  $t$  coincides with either boundary timestamp of  $z$ . Otherwise,  $t$  is replaced with  $z.t_+$ . After examining all the updates in  $BUF$ , we sort them in ascending order of their timestamps. Note that the sorting can be completed in  $O(|BUF|/B)$  I/Os, since now every update carries a timestamp equal to either  $z.t_+$  or  $z.t_-$ . The sorted list is appended to the stream of  $DB^*$ .

Next, the algorithm clears  $BUF$ , and continues to scan the stream of  $DB$ , until  $BUF$  overflows again. The above steps are repeated until all the updates of  $DB$  have been seen. At this point, regardless of  $|BUF|$ , we invoke the overflow handling process to generate the last zone, as well as  $|BUF|$  updates in the stream of  $DB^*$ . The overall transformation requires  $O(n)$  I/Os. The total number of zones is at most  $1 + 2N/\lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor = O(\varepsilon_{hr} \cdot N)$  (since at least  $\lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor$  updates of  $DB$  happen in each zone); therefore,  $DB^*$  is synchronous.

To illustrate the algorithm, Figure 7a shows a  $DB$  with 5 objects, assuming  $\varepsilon_{hr} = \frac{1}{16}$ . After all the updates of  $DB$  at timestamp 3 have been scanned,  $BUF$  contains 4 insertions  $\langle \text{INS}, 5k, 1 \rangle$ ,  $\langle \text{INS}, 4k, 2 \rangle$ ,  $\langle \text{INS}, 3k, 3 \rangle$  and  $\langle \text{INS}, 2k, 3 \rangle$ , and overflows (since  $4 > \lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor = 3$ ). To handle the overflow, we first create a zone  $z_1 = [1, 3]$ , and then examine each update in  $BUF$ . The first insertion is not modified, since its timestamp coincides with the starting time of  $z$ . The time 2 of the second update, however, is not a boundary timestamp of  $z$ ; hence, the time of the update is changed to 1. The next two updates are retained directly. Thus, the overflow handling creates 4 updates in the stream of  $DB^*$ :  $\langle \text{INS}, 5k, 1 \rangle$ ,  $\langle \text{INS}, 4k, 1 \rangle$ ,  $\langle \text{INS}, 3k, 3 \rangle$  and  $\langle \text{INS}, 2k, 3 \rangle$ , which start the lifespans of  $o_1^*$ ,  $o_2^*$ ,  $o_3^*$ , and  $o_4^*$  in Figure 7b, respectively.

Similarly, the second overflow happens at time 6, when  $BUF$  involves  $\langle \text{INS}, 1k, 4 \rangle$ ,  $\langle \text{LD}, 1k, 5 \rangle$ ,  $\langle \text{LD}, 4k, 6 \rangle$

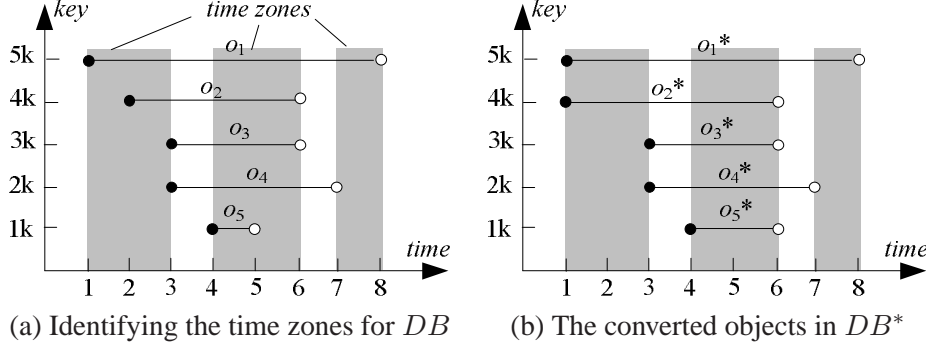


Figure 7: The zoning transformation

and  $\langle \text{LD}, 3k, 6 \rangle$ . In this case, the overflow handling creates another zone  $z_2 = [4, 6]$ , alters only  $\langle \text{LD}, 1k, 5 \rangle$ , and adds the resulting updates to the stream of  $DB^*$ . In particular, the alternation changes the timestamp 5 to 6, which is necessary, because 5 is not the starting/ending time of  $z_2$ . Finally, at the last timestamp 8,  $BUF$  includes  $\langle \text{LD}, 2k, 8 \rangle$  and  $\langle \text{LD}, 5k, 8 \rangle$ . After producing the third zone  $z_3 = [7, 8]$ , we directly append both logical deletions to the  $DB^*$  stream. Now the stream contains 10 updates, which spawn the 5 objects in Figure 7b. In general, the zoning transformation converts each object  $o \in DB$  to another object  $o^* \in DB^*$  with the same key, but perhaps a longer lifespan.

**Lemma 4.** *Given any half-ranged query  $q$  with approximation ratio  $\varepsilon_{hr}$ , let  $x$  be the answer obtained by applying our solution to  $DB^*$ ; then,  $x$  is also a correct result with respect to  $DB$ , i.e.,  $x$  satisfies the precision requirement of Problem 2.*

*Proof.* Let  $v$  be the precise answer of processing  $q$  on  $DB$ . To establish the lemma, we will show that  $|x - v| < \frac{1}{4\varepsilon_{hr}} + \varepsilon_{hr} \cdot N_{alive}(q_t)$ , where  $N_{alive}(q_t)$  is the number of objects in  $DB$  alive at time  $q_t$ .

Let us use  $v^*$  to denote the precise result of  $q$  with respect to  $DB^*$ , i.e.,  $v^*$  is the number of objects  $o^* \in DB$  alive at time  $q_t$  whose keys are in  $\vec{q}_k$ . By the way  $x$  is computed, we know:

$$|x - v^*| < \varepsilon_{hr} \cdot N_{alive}^*(q_t) \quad (10)$$

where  $N_{alive}^*(q_t)$  is the number of objects in  $DB^*$  alive at time  $q_t$ .

Let  $S$  (and  $S^*$ ) be the set of keys of the objects in  $DB$  (and  $DB^*$ ) alive at time  $q_t$ . In the zoning transformation, when  $o \in DB$  is converted to its counterpart  $o^* \in DB^*$ , the lifespan of  $o^*$  always covers that of  $o$ . Hence,  $S \subseteq S^*$ . Furthermore, the transformation guarantees that  $S^*$  contains at most  $\lfloor \frac{1}{4\varepsilon_{hr}} - 1 \rfloor$  elements that do not belong to  $S$ . Therefore:

$$|v^* - v| \leq \frac{1}{4\varepsilon_{hr}} - 1, \text{ and} \quad (11)$$

$$|N_{alive}^*(q_t) - N_{alive}(q_t)| \leq \frac{1}{4\varepsilon_{hr}} - 1. \quad (12)$$

Thus,

$$\begin{aligned}
|x - v| &= |(x - v^*) + (v^* - v)| \\
\text{(By Inequalities 10 and 11)} &< \varepsilon_{hr} \cdot N_{alive}^*(q_t) + \frac{1}{4\varepsilon_{hr}} - 1 \\
\text{(By Inequality 12)} &< \varepsilon_{hr} \cdot \left( N_{alive}(q_t) + \frac{1}{4\varepsilon_{hr}} - 1 \right) + \frac{1}{4\varepsilon_{hr}} - 1 \\
&< \frac{1}{4\varepsilon_{hr}} + \varepsilon_{hr} \cdot N_{alive}(q_t)
\end{aligned}$$

which concludes the proof.  $\square$

Summarizing the results in Sections 4 through 6, we have

**Theorem 2.** *Let  $DB$  be a set of  $N$  temporal objects  $o$ . We can build an MVB-tree that consumes  $O(\min\{n, n \cdot (\frac{1}{\varepsilon})^2 / \overline{N_{alive}}\})$  space, where  $n = N/B$ ,  $B$  is the disk page size, and  $\overline{N_{alive}}$  the average number of live objects at an anchor timestamp. The tree permits any query of Definition 3 to be answered in  $O(\min\{\log_B n, \log_B \frac{n}{\varepsilon} - \log_B \overline{N_{alive}}\})$  I/Os. The preprocessing can be completed in  $O(N \log_B n)$  I/Os, or equivalently,  $O(\log_B n)$  amortized time per object.*

## 7 Supporting Data Updates

Our discussion so far focuses on a static  $DB$ . In this section, we will explain how to apply the proposed technique to a dynamic  $DB$ . For this purpose, let us assume that  $DB$  is initially empty, and then updated with insertions and logical deletions in a temporal stream. Accordingly, we interpret  $N$  as the number of objects that have been inserted (but may or may not have been logically deleted); as before,  $n = N/B$ . We will present a solution that always occupies  $O(n)$  space, answers any half-ranged query in  $O(\log_B n)$  I/Os, and incorporates the next update in amortized  $O(\log_B n)$  I/Os.

Recall that, given a static  $DB$ , we will (i) first perform the zoning transformation to convert it to  $DB^*$ , (ii) then apply the algorithm of Figure 4 on  $DB^*$  to generate anchor segments, and (iii) finally build an MVB-tree on the resulting segments. Note that the three modules, zoning transformation, anchor (segment) generation, and MVB construction, all accept a temporal stream as their inputs, i.e., streams of  $DB$ ,  $DB^*$ , and anchor segments, respectively. Furthermore, the first two modules also output a temporal stream, namely, streams of  $DB^*$  and anchor segments, respectively.

To support a dynamic  $DB$ , we will perform these modules simultaneously in a pipelined fashion, as illustrated in Figure 8. Whenever an insertion or logical deletion (in the stream of  $DB^*$ ) is generated by the zoning transformation, it is fed immediately into anchor generation<sup>4</sup>. In turn, any update (in the stream of

<sup>4</sup>Remember that, to process an event timestamp  $t$ , anchor generation must have all the updates at time  $t$ . Fortunately, this does not create any problem in pipelining, because zoning transformation produces updates of the same timestamp together.

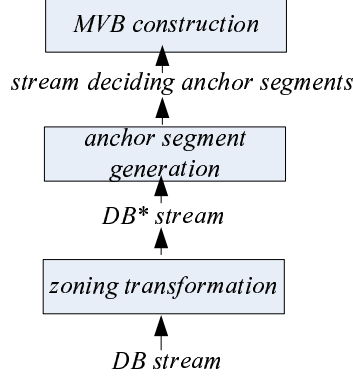


Figure 8: Pipelining for supporting a dynamic  $DB$

anchor segments) produced by anchor generation will be forwarded to MVB construction right away. Other than these connections, the three modules are independent, and work in exactly the same way as in the case of static  $DB$ . Therefore, the amortized I/O cost is identical to that of the static case, i.e.,  $O(\log_B N)$  I/Os, as stated in Theorem 2.

Query answering under the pipelined framework is slightly more complex. Let  $CT_{DB}$  and  $CT_{DB^*}$  be the most recent timestamps that have been fully processed by the modules of zoning transformation and anchor generation, respectively (a timestamp  $t$  is fully processed if all the updates at  $t$  have been tackled). Apparently,  $CT_{DB} \geq CT_{DB^*}$ . The query time  $q_t$  of a half-ranged query  $q$  may equal any value between 0 and  $CT_{DB}$ . Depending on  $q_t$ , query  $q$  should be answered in different ways:

- Case 1 [ $q_t \leq CT_{DB^*}$ ]: The MVB-tree has already captured the anchor segments alive at  $q_t$ . Hence, using the reduction in Section 4, we perform timestamp floor search  $TF(q_t, q_{k-1})$  on the MVB-tree, and return the rank associated with the retrieved anchor segment.
- Case 2 [ $CT_{DB^*} < q_t \leq CT_{DB}$ ]: The anchor segments alive at  $q_t$  have not been computed yet; thus, it is not possible to employ the MVB-tree for solving  $q$ . As a crucial observation, in this case, the  $BUF$  employed by the zoning transformation module contains less than  $\lfloor \frac{1}{4\epsilon_{hr}} - 1 \rfloor$  updates (since  $BUF$  would have been emptied if  $|BUF| \geq \lfloor \frac{1}{4\epsilon_{hr}} - 1 \rfloor$ ). Furthermore, these updates all bear a timestamp larger than  $CT_{DB^*}$  (notice that  $CT_{DB^*}$  equals the time when  $BUF$  was cleared last time).

Remember that, as explained in Section 5.2, the anchor generation module maintains a “sweeping-state” structure, which is an augmented B-tree, indexing the keys of all the objects in  $DB^*$  alive at  $CT_{DB^*}$ . Let us maintain a similar structure, denoted as  $\Upsilon$ , which is an augmented B-tree managing the keys of all the objects in  $DB$  alive at  $CT_{DB}$ . To process  $q$ , we simply invoke  $\text{FIND-RANK}(q_{k-1})$  on  $\Upsilon$ , and directly return the result ( $\text{FIND-RANK}$  is a procedure explained in Section 5.3). Such a result has error less than  $\lfloor \frac{1}{4\epsilon_{hr}} - 1 \rfloor$ , because all the updates on  $DB$  between timestamps  $q_t$  and

$CT_{DB}$  must exist in  $BUF$ , whose size, as mentioned earlier, does not exceed  $\lfloor \frac{1}{4\epsilon_{hr}} - 1 \rfloor$ .  $\Upsilon$  occupies size  $O(N_{alive}(CT_{DB}))$  and can be maintained in  $O(\log_B N_{alive}(CT_{DB}))$  I/Os per update, where  $N_{alive}(CT_{DB})$  is the number of objects in  $DB$  alive at  $CT_{DB}$ , and may be by far lower than  $N$ . The cost of performing  $\text{FIND-RANK}(q_{k-1})$  on  $\Upsilon$  is  $O(\log_B N_{alive}(CT_{DB}))$  I/Os.

**Theorem 3.** *Given a temporal stream  $DB$ , we can maintain several structures that consume totally  $O(n)$  space, and can be maintained in  $O(\log_B n)$  I/Os per update, where  $n = N/B$ ,  $N$  is the total number of insertions in the stream so far, and  $B$  is the disk page size. Using these structures, we can solve any query of Definition 3 in  $O(\log_B n)$  I/Os.*

## 8 Experiments

This section experimentally evaluates the effectiveness and efficiency of the proposed solution, referred to as *AncSeg* (for “anchor segments”) in the sequel. Towards this purpose, we synthesize temporal datasets, each of which simulates the balance changes of 100k bank accounts  $acc_1, acc_2, \dots, acc_{100k}$ , during a history of  $H$  discrete timestamps  $1, 2, \dots, H$ . Specifically, the temporal stream of a dataset is generated as follows. At time 1, 100k insertions are created in the stream, registering the initial balances of all accounts. Then, at every subsequent timestamp  $t$ , the balances of  $a \cdot 100k$  accounts are updated (in a way to be clarified shortly), where  $a$  is a dataset parameter called *agility*. In particular, each update is performed by a logical deletion, followed by an insertion, both at time  $t$ . In other words, there are  $a \cdot 100k$  logical deletions/insertions per timestamp (hence, a dataset with a larger  $a$  is more “agile”). Notice that each account is modified  $a \cdot H$  times in expectation during the entire history.

To generate balance updates, we choose two distributions  $DIST_s$  and  $DIST_e$ , to be the balance distributions at the first and last timestamps of the history, respectively. The initial balance  $acc_i.key_s$  ( $1 \leq i \leq 100k$ ) is randomly decided following  $DIST_s$ . Similarly, we also generate  $acc_i.key_e$  according to  $DIST_e$ , which is the (expected) balance of  $acc_i$  at the final timestamp  $H$ . Then, at each update of  $acc_i$ , we add  $\frac{acc_i.key_e - acc_i.key_s}{a \cdot H}$  to its current balance (so that, after  $a \cdot H$  updates, the balance of  $acc_i$  becomes  $acc_i.key_e$ ). Note that  $acc_i.key_e$  may be smaller than  $acc_i.key_s$ , in which case the addition essentially decreases the balance.

$DIST_s$  and  $DIST_e$  are selected from the following four distributions: *Uniform*, *Zipf*, *CA*, and *LB*, all of which have a key domain  $[0, 10000]$ . Specifically, *Uniform* represents the uniform distribution, whereas *Zipf* stands for a Zipf distribution skewed towards 0 with a coefficient<sup>5</sup> 0.8. *CA*, on the other hand, is obtained from a real spatial dataset, containing 63k two-dimensional points denoting addresses in California. For each point, we take its distance to the center of the two-dimensional data space. After being normalized to

<sup>5</sup>When the coefficient equals 1, all numbers in the distribution equal 0; when it equals 0, the distribution degenerates into uniformity.

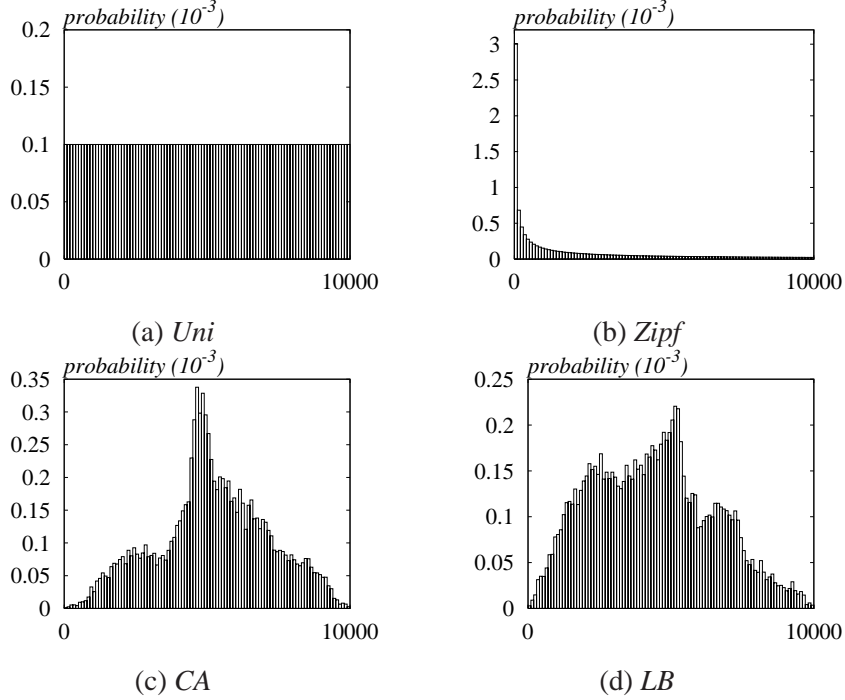


Figure 9: Probability density functions of key distributions

$[0, 10000]$ , all the distances constitute the distribution *CA*. Generating a value according to *CA* means setting the value to a distance randomly. *LB* is created similarly, except from another spatial dataset including 53k locations in the Long Beach county. Both spatial datasets are the products of the TIGER project of the US Census Bureau, and available at <http://www.census.gov/geo/www/tiger/>.

Figure 9 visualizes the probability density functions of *Uni*, *Zipf*, *CA*, and *LB*, respectively. We will denote a temporal dataset by the names of the  $DIST_s$  and  $DIST_e$  used to generate it. For example, *Zipf-CA* indicates a dataset with  $DIST_s = Zipf$  and  $DIST_e = CA$ . Figures 10a through 10d illustrate the balance distributions of *Zipf-CA* (with agility  $a = 5\%$  and history length  $H = 300$ ) at timestamps 60, 120, 180, and

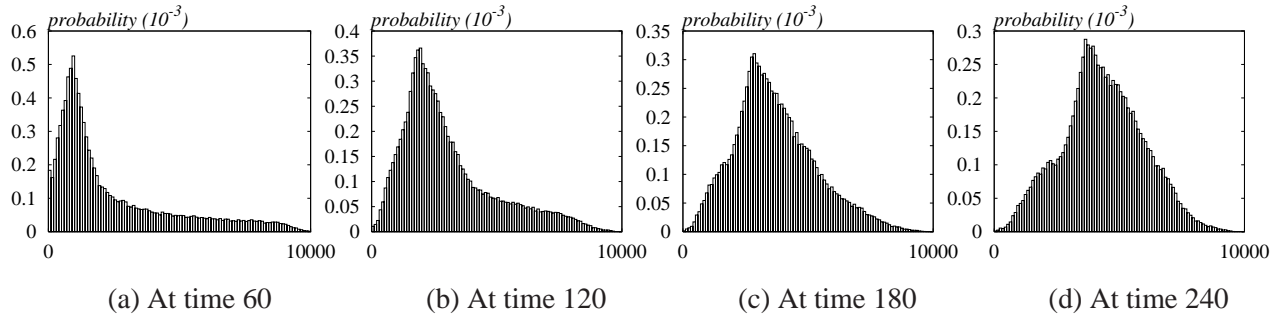


Figure 10: Key distribution transition of *Zipf-CA* ( $a = 5\%$ ,  $H = 300$ )

Parameters	Values
history length $H$	100, <b>300</b> , 500
agility $a$	1%, 2.5%, <b>5%</b> , 7.5%, 10%
approximation ratio $\varepsilon$	<b>1%</b> , 2%, 3%, 4%, 5%
length of a query’s key range $q_k len$	100, 500, <b>1000</b> , 1500, 2000

Table 2: Experiment parameters (bold values are defaults)

$H$	100	300	500
Cardinality	600k	1.6 million	2.6 million

(a) Cardinality vs.  $H$  ( $a = 5\%$ )

$a$	1%	2.5%	5%	7.5%	10%
Cardinality	400k	850k	1.6 million	2.35 million	3.1 million

(b) Cardinality vs.  $a$  ( $H = 300$ )

Table 3: Cardinalities of the datasets in our experiments

240, respectively. Remember that the distributions at timestamps 1 and  $H$  are *Zipf* and *CA*, respectively. We will examine datasets with 4 distribution transitions: *Uni-CA*, *Uni-LB*, *Zipf-CA*, and *Zipf-LB*.

Each *workload* contains 10000 queries  $q$  obtained as follows. The query timestamp  $q_t$  of  $q$  is a random integer in the range of  $[1, 100]$ . To create its key range  $\vec{q}_k = [q_{k+}, q_{k-}]$ , we first decide the value of  $q_{k+}$  in  $[0, 10000 - q_k len]$ , following the key distribution of the underlying dataset at time  $q_t$ . Here,  $q_k len$  is a workload parameter dictating the length of  $\vec{q}_k$ . Then,  $q_{k-}$  is set to  $q_{k+} + q_k len$ .

Table 2 summarizes the parameters of the subsequent experiments, together with their values, of which those in bold are the defaults. Unless specifically stated, each parameter is set to its default. Assuming  $a = 5\%$ , Table 3a demonstrates the cardinalities of the datasets with various  $H$  (regardless of their  $DIST_s$  and  $DIST_e$ ). Similarly, fixing  $H$  to 300, Table 3b presents the cardinalities with respect to different  $a$ . Observe that the cardinality increases linearly with both  $H$  and  $a$ .

The page size is set to 4k bytes in all the experiments. We will compare the proposed *AnsSeg* technique against the *MVSB*-tree, which is the state-of-the-art for precise temporal aggregation, as reviewed in Section 2. Section 8.1 demonstrates that *AnsSeg* consumes significantly less space than *MVSB*. In fact, as shown later, *AnsSeg* actually takes up only a fraction of the space occupied by the original dataset. Section 8.2 proves that our solution retrieves highly accurate answers with very low I/O cost.

## 8.1 Space Consumption

The first set of experiments evaluates the number of anchor segments produced by *AnsSeg*, which depends on the key distribution, history length  $H$ , dataset agility  $a$ , and approximation ratio  $\varepsilon$ . Figure 11a plots

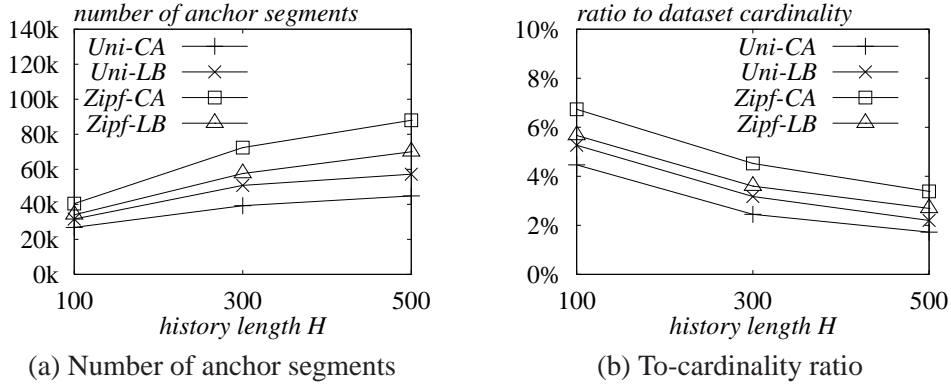


Figure 11: Volume of anchor segments vs.  $H$  ( $a = 5\%$ ,  $\varepsilon = 1\%$ )

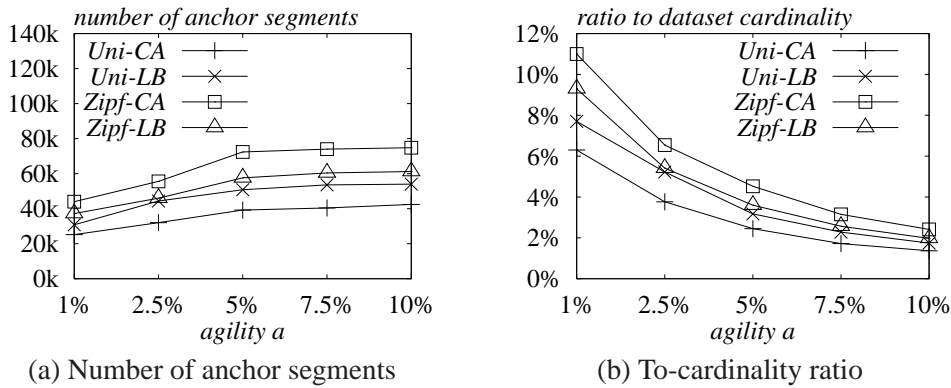


Figure 12: Volume of anchor segments vs.  $a$  ( $H = 300$ ,  $\varepsilon = 1\%$ )

the number as a function of  $H$ , for datasets with different distribution transitions. As  $H$  increases, the dataset cardinality becomes higher, too (see Table 3a). Therefore, we also inspect the *to-cardinality ratio*, equal to the number of anchor segments divided by the cardinality. Figure 11b presents the corresponding ratios for the results in Figure 11a. Unlike the linear growth of cardinality with  $H$ , the number of anchor segments increases only sub-linearly, such that the to-cardinality ratios continuously drops as  $H$  escalates. This phenomenon suggests that *AnsSeg* is well-suited for a practical database, where the history is typically very lengthy.

Figure 12a (12b) illustrates the number of anchor segments (to-cardinality ratio) with respect to various agilities  $a$ . The number of anchor segments again accounts for only a small percentage of the dataset size (maximum 11% of the cardinality), and monotonically decreases as  $a$  grows. In Figure 13, we increase  $\varepsilon$  up to 5%, and measure the volume of anchor segments accordingly. As expected, fewer anchor segments are necessary when the accuracy requirement is weaker.

The next set of experiments examines the space consumption of *AncSeg* and *MVSB*. Focusing on *Uni-CA*, Figure 14a compares the amount of space occupied by *AncSeg*, *MVSB*, and the underlying datasets. Note

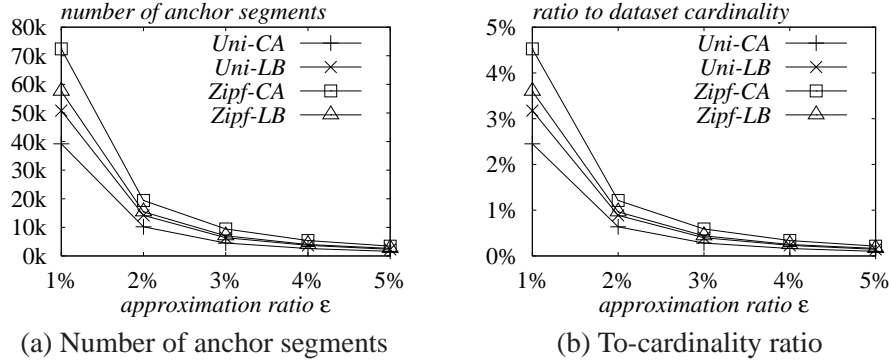


Figure 13: Volume of anchor segments vs.  $\epsilon$  ( $H = 300$ ,  $a = 5\%$ )

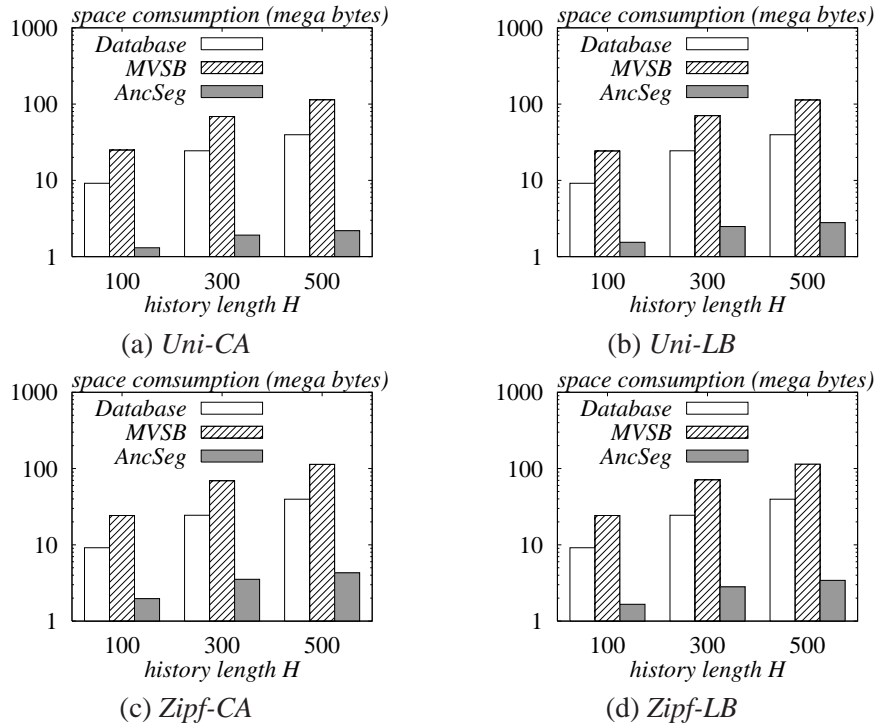


Figure 14: Space overhead vs.  $H$  ( $a = 5\%$ ,  $\epsilon = 1\%$ )

that the y-axis is in logarithmic scale. Due to its  $O(n \log n)$  complexity, an MVSB-tree is several times larger than the original database. The space cost of the proposed technique, on the other hand, is only a small fraction of the database volume, and lower than that of MVSB by *an order of magnitude*. Figures 14b, 14c, and 14d present the comparison for Uni-LB, Zipf-CA, and Zipf-LB, respectively, confirming the same observations.

We repeat the experiments of Figure 14, but by fixing  $H$  to 300, and varying  $a$  instead. The results are presented in Figure 15. The overall phenomena are analogous to those in Figure 14, except that the difference

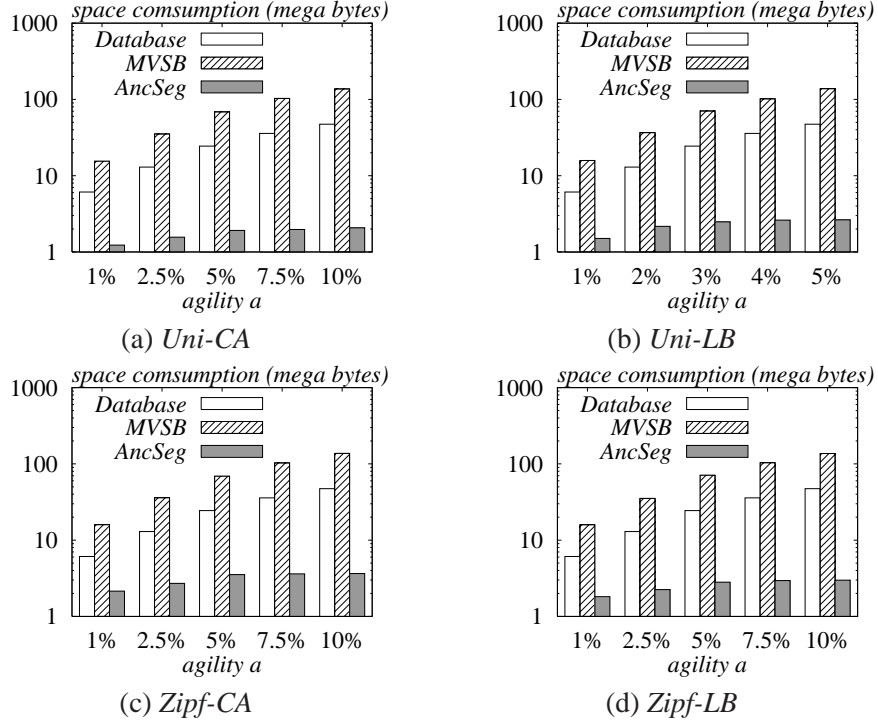


Figure 15: Space overhead vs.  $a$  ( $H = 5\%$ ,  $\varepsilon = 1\%$ )

between *AncSeg* and *MVSB* is ever more significant (recall that, as shown in Figure 12, the number of anchor segments accounts for a smaller percentage of the cardinality, as  $a$  grows).

The last experiment of this subsection evaluates the space volume of *AncSeg*, when the approximation ratio  $\varepsilon$  is raised from 1% to 5%. Figure 16 demonstrates the results. For convenience of comparison, the figure also provides the volumes of the database and *MVSB*, which, apparently, do not vary with  $\varepsilon$ . As is consistent with the findings in Figure 13, the space cost of *AncSeg* decreases drastically as  $\varepsilon$  grows. In particular, for  $\varepsilon \geq 4\%$ , our technique requires less than 100k bytes, for a dataset of over 20 mega bytes!

## 8.2 Query and Update Performance

Having shown that *AncSeg* entails low space overhead, we proceed to examine its query and update efficiency. In particular, we study three aspects of performance in turn: query error, processing cost, and update cost.

The relative error of a query (in the approximate answer returned by *AncSeg*) is calculated as  $|est - act|/act$ , where *act* and *est* denote the precise and approximate results, respectively. Given a workload, we measure the quality of (the answers produced by) *AncSeg* using two metrics: the *median error*, and the *confidence bound*. Specifically, the former is the median of the relative error of all the queries in the workload. The

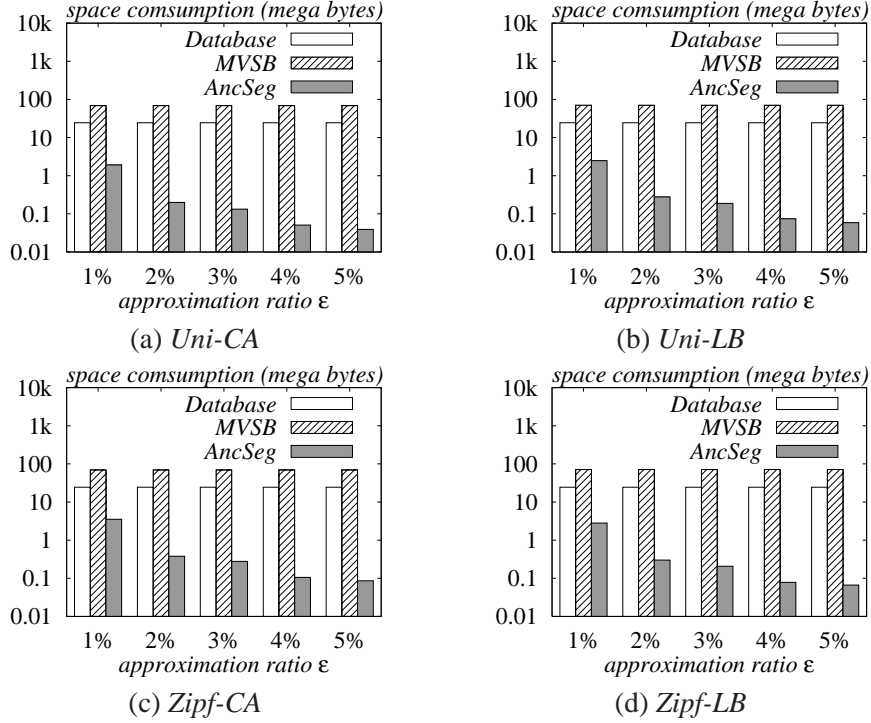


Figure 16: Space overhead vs.  $\epsilon$  ( $H = 5\%$ ,  $a = 5\%$ )

confidence bound, on the other hand, equals the 9000-th largest relative error in a workload. In other words, the relative error of 90% of the queries in a workload do not exceed the confidence bound.

Currently, there does not exist any method that can rigorously match the space consumption and query precision of *AncSeg* at the same time. Thus, we compare *AncSeg* against the histogram methodology, since histograms are a highly popular selectivity estimation approach. Regarding each temporal object and a timestamp range count query as special rectangles, we implemented a solution developed in [20], which is designed to support selectivity estimation of rectangular range search over data rectangles; that solution is referred to as *Histogram* in the sequel. In each experiment, the amount of space allocated to *Histogram* is equivalent to the space occupied by *AncSeg*.

Figure 17 compares the query error of *AncSeg* and *Histogram* by varying  $H$ , using the default settings of  $a$ ,  $\epsilon$ , and  $q_k.len$ . Specifically, Figure 17a shows the results for the dataset *Uni-CA*. The white (grey) columns represent the median error of *AncSeg* (*Histogram*). The horizontal bar above a column indicates the confidence bound of the corresponding median. Figures 17b, 17c, and 17d illustrate similar results with respect to datasets *Uni-LB*, *Zipf-CA* and *Zipf-LB*, respectively. Note that the y-axis is in logarithmic scales.

*Histogram* incurs huge error, which is often over 100%, and higher than the error of *AncSeg* by orders of magnitude. The poor effectiveness of *Histogram* is expected for two reasons. First, (unlike *AncSeg*) this

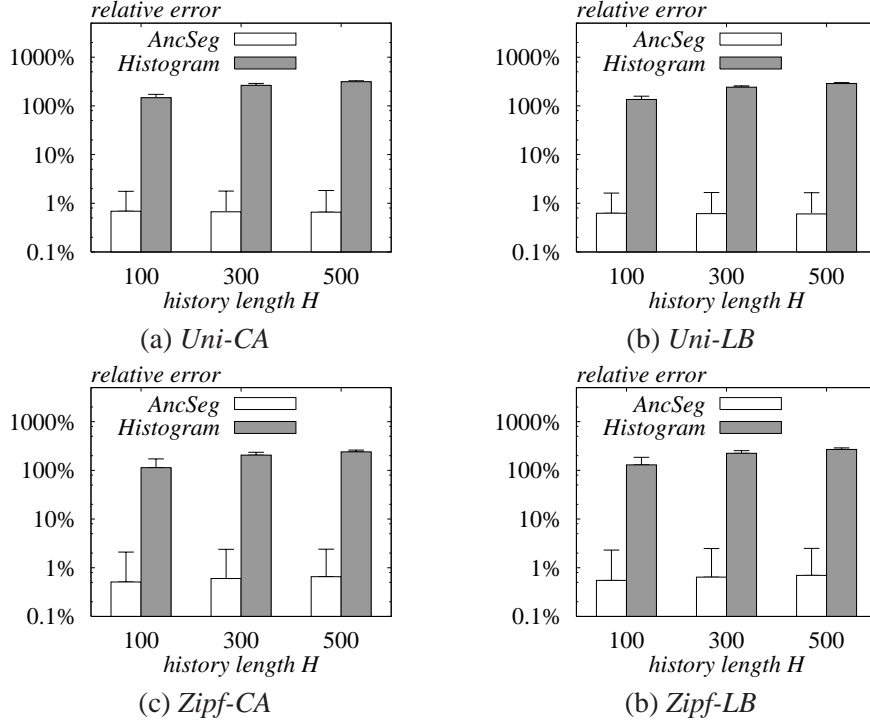


Figure 17: Query precision comparison between *AncSeg* and *Histogram* under different  $H$  ( $a = 5\%$ ,  $\varepsilon = 1\%$ ,  $q_k.len = 1000$ )

solution does not have any provably-good precision guarantees such that in the worst case its absolute error can be as large as the dataset cardinality. Second, it is well-known that histograms perform well only if they are allowed a sufficiently large amount of space. In other words, the space consumption of *AncSeg* is considerably below the threshold where *Histogram* is able to yield accurate estimates. Since *Histogram* is consistently much worse in all the following experiments, we do not discuss this solution further.

Figure 18 demonstrates the precision of *AncSeg* as  $a$  varies from 1% to 10%. Two crucial observations can be made from Figures 17 and 18. First, *AncSeg* produces extremely accurate answers, with maximum confidence bound below 3% in all cases. Remember that such high accuracy is achieved by occupying a tiny fraction (around 1/10) of the space of the database (see Figures 14 and 15). Second, the quality of *AncSeg* is fairly stable, and is hardly affected by  $H$  and  $a$ . Particularly, in Figure 17 (18), the results at various  $H$  ( $a$ ) in the same diagram differ by less than 0.5%.

To study the impact of  $\varepsilon$ , we measure the median error and confidence bound, as  $\varepsilon$  grows from 1% to 5%. As shown in Figure 19, although the answers from *AncSeg* are less precise for a greater  $\varepsilon$ , they are still reasonably accurate. These results, combined with those in Figure 16, indicate that our technique offers a tradeoff between accuracy and space consumption. A user can set  $\varepsilon$  to the lowest value satisfying her/his precision requirement, to achieve the minimum space overhead. In Figure 20, we study the error changes

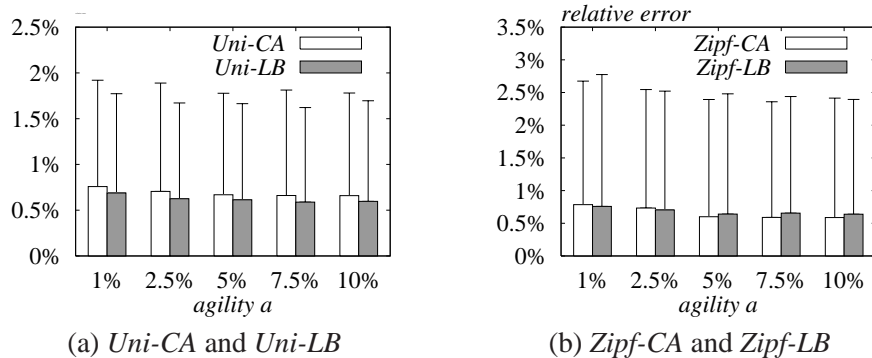


Figure 18: Relative error of *AncSeg* vs.  $a$  ( $H = 300$ ,  $\varepsilon = 1\%$ ,  $q_k len = 1000$ )

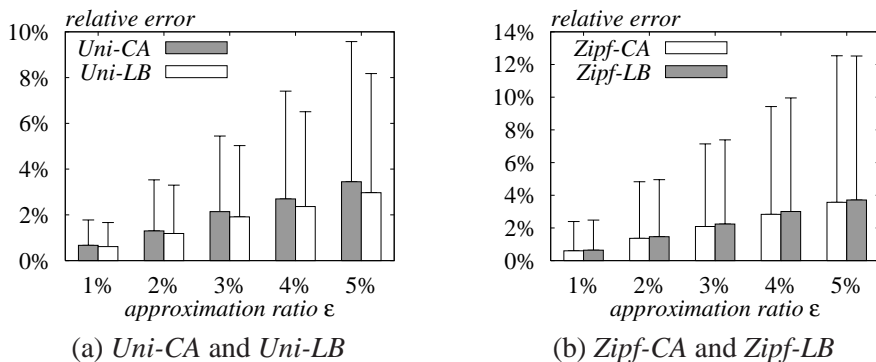


Figure 19: Relative error of *AncSeg* vs.  $\varepsilon$  ( $H = 300$ ,  $a = 5\%$ ,  $q_k len = 1000$ )

with respect to  $q_k len$ . *AncSeg* is especially effective when the query results are large. This is not surprising, because the absolute error is always limited within a bound, which is not influenced by  $q_k len$ ; therefore, given a greater actual result, the relative error decreases.

The next set of experiments investigates the query cost of *AncSeg*, and how it compares with *MVSB*. Recall that the indexing schemes of *AncSeg* and *MVSB* leverage the multi-version methodology described in Section 2.3. In particular, for both solutions, query processing is carried out only in the logical tree (in the multi-version structure) that is responsible for the query timestamp. Assuming that the tree has a height  $h$ , the query cost is always  $2h$ . For *AncSeg*, this is due to the fact that, it answers a query by two timestamp floor retrievals, each of which scans a single path of the tree. The reason for *MVSB* is similar; it reduces a timestamp range count query to two “single-time-less-key” operations [38], each of which also accesses one path of the tree. Therefore, the relative superiority (in query efficiency) between *AncSeg* and *MVSB* is determined by the heights of their logical trees for processing the same query.

The height of a logical tree at a timestamp  $t$ , in turn, is proportional to the number of objects alive at  $t$ . For *AncSeg*, each “object” means an anchor segment, whereas, for *MVSB*, it refers to an ordinary object in the dataset. Hence, a logical tree in *AncSeg* has fewer levels than in an *MVSB*-tree. For all the experiments in

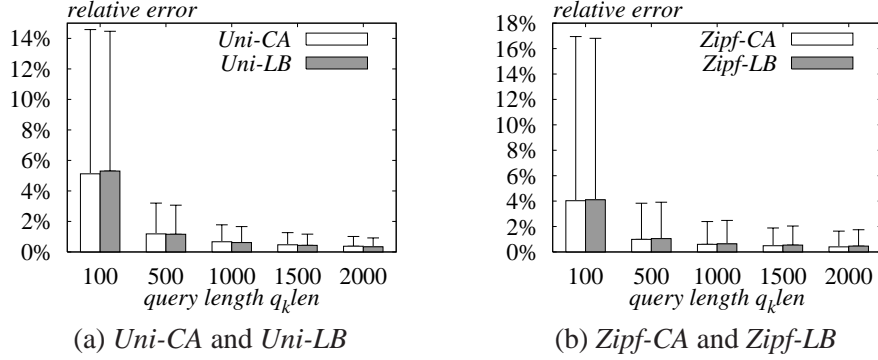


Figure 20: Relative error of *AncSeg* vs.  $q_k \text{ len}$  ( $H = 300$ ,  $a = 5\%$ ,  $\varepsilon = 1\%$ )

Figures 17-20, *AncSeg* answers every query in 4 I/Os (i.e., the height equals 2), compared to 6 I/Os of *MVSB*. The only exception happens in the scenario of Figure 19. Here, for  $\varepsilon \geq 2\%$ , each logical tree in *AncSeg* has only a single level; therefore, the query overhead becomes 2 I/Os (while the *MVSB* cost remains 6 I/Os).

Finally, we discuss the update cost of *AncSeg*. It requires more expensive update overhead than *MVSB*, because each object needs to be inserted to and then removed from two structures. Specifically, as explained in Section 7, when an object  $o$  arrives, it is first added to a structure  $\Upsilon$  in the “zone transformation” module. Later,  $o$  is deleted from  $\Upsilon$ , then included in the sweeping-state structure in the module of “anchor segment generation”, and finally, evicted from that structure as well. Both structures have a height of 3; therefore, the overall update cost of *AncSeg* is 12 I/Os per object. This is 4 times higher than the update overhead of *MVSB* since, for each object, *MVSB* performs a single insertion into an *MVSB*-tree with 3 levels.

## 9 Conclusions

Temporal aggregation is an important operator for two reasons. First, aggregates are the direct target of analysis in a large number of applications of temporal databases. Second, the numbers of objects qualifying various range predicates are essential inputs to many sophisticated data mining tasks, such as association rule mining, decision tree learning, etc. Motivated by the fact that precise aggregation demands expensive space or query overhead, in this paper, we propose a novel technique for efficiently computing approximate results with good quality guarantees. Our solution possesses rigorous theoretical bounds:  $O(n)$  space consumption and  $O(\log_B n)$  query time. Furthermore, as proved by extensive experiments, the solution also exhibits excellent practical performance. In particular, it requires only a tiny fraction (around 1/10) of the space of the database, settles any query in a small number (maximum 4 in all tested cases) of I/Os, and provides answers with median relative error below 5%. The space and query overhead is significantly lower than that of the current state-of-the-art.

For future research, it would be interesting to study temporal aggregation on more complex data. In this paper

we assume only one non-temporal attribute in a query’s predicate whereas, in general, multiple such attributes may be involved. For example, in weather monitoring, a sensor may perform multiple types of measurement such as temperature, humidity, and so on. Hence, a query may retrieve the regions where the temperature and humidity readings fall in certain ranges at a particular timestamp. Extending the proposed solutions to support such queries remains a challenging topic. Another exciting direction is to investigate objects that are modeled as segments with arbitrary orientations (not just horizontal segments, as are discussed in our work). Such objects are abundant in spatiotemporal scenarios. For example, consider the locations of vehicles on a highway. Treating the highway as a one-dimensional key domain, each location (i.e., the key) can be represented as a single value. When a vehicle is traveling with a fixed velocity  $v$ , its trajectory is a line segment in the key-time space, whose slope equals exactly  $v$  (hence, the segment is horizontal only if the vehicle is static). Another segment is created whenever the object alters its velocity. Here, a temporal range count query would “find the number of vehicles on I-95 between Exits 4 and 8 at 11am yesterday”.

## Acknowledgements

This work was partially supported by Grant CUHK 1202/06 from HKRGC.

## References

- [1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. pages 307–328, 1996.
- [2] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *SIGMOD*, pages 439–450, 2000.
- [3] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.
- [4] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Comput. Geom.*, 29(2):147–162, 2004.
- [5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.
- [6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.
- [7] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.
- [8] M. H. Bohlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.
- [9] M. H. Bohlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *TODS*, 25(4), 2000.
- [10] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. The md-join: An operator for complex olap. In *ICDE*, pages 524–533, 2001.

- [11] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.
- [12] S.-J. Chun, C.-W. Chung, J.-H. Lee, and S.-L. Lee. Dynamic update cube for range-sum queries. pages 521–530, 2001.
- [13] S. Geffner, D. Agrawal, A. E. Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic olap data cubes. In *ICDE*, pages 328–335, 1999.
- [14] J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel algorithms for computing temporal aggregates. In *ICDE*, pages 418–427, 1999.
- [15] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *ICDT*, pages 143–157, 2003.
- [16] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.
- [17] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
- [18] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *SIGMOD*, pages 73–88, 1997.
- [19] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD*, pages 299–310, 2004.
- [20] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *ICDE*, pages 525–534, 2000.
- [21] J. S. Kim, S. T. Kang, and M. H. Kim. On temporal aggregate processing based on time points. *Inform. Process. Lett. (IPL)*, 71(5-6):213–220, 1999.
- [22] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, 1995.
- [23] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.
- [24] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE*, pages 362–374, 2004.
- [25] X. Lin, J. Xu, Q. Zhang, H. Lu, J. X. Yu, X. Zhou, and Y. Yuan. Approximate processing of massive continuous quantile queries over high-speed data streams. *TKDE*, 18(5):683–698, 2006.
- [26] I. F. V. Lopez, R. Snoggrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Trans. Knowl. Data Eng.*, 17(2):271–286, 2005.
- [27] B. Moon, I. F. V. Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *TKDE*, 15(3):744–759, 2003.
- [28] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.

- [29] M. Riedewald, D. Agrawal, and A. E. Abbadi. Efficient integration and aggregation of historical information. In *SIGMOD*, pages 13–24, 2002.
- [30] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
- [31] C. Sun, D. Agrawal, and A. E. Abbadi. Exploring spatial datasets with histograms. In *ICDE*, pages 93–102, 2002.
- [32] Y. Tao, D. Papadias, and C. Faloutsos. Approximate temporal aggregation. In *ICDE*, pages 190–201, 2004.
- [33] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *EDBT*, pages 682–700, 2002.
- [34] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *TKDE*, 9(3):391–409, 1997.
- [35] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 12(3):262–283, 2003.
- [36] X. Ye and J. A. Keane. Processing temporal aggregates in parallel. In *Inter. Conf. on Systems, Man, and Cybernetics*, pages 1373–1378, 1997.
- [37] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, pages 646–663, 2002.
- [38] D. Zhang, A. Markowitz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *PODS*, 2001.
- [39] D. Zhang and V. J. Tsotras. Optimizing spatial min/max aggregations. *The VLDB Journal*, 14(2):170–181, 2005.
- [40] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *PODS*, pages 121–132, 2002.