

Exploiting Temporal Correlation in Temporal Data Warehouses^{*}

Ying Feng, Hua-Gang Li, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science,
University of California, Santa Barbara
{yingf, huagang, agrawal, amr}@cs.ucsb.edu

Abstract. Data is typically incorporated in a data warehouse in increasing order of time. Furthermore, the MOLAP data cube tends to be sparse because of the large cardinality of the time dimension. We propose an approach to improve the efficiency of range aggregate queries on MOLAP data cubes in a temporal data warehouse by factoring out the time-related dimensions. These time-related dimensions are handled separately to take advantage of the monotonic trend over time. The proposed technique captures local data trends with respect to time by partitioning data points into blocks, and then uses a *perfect binary block tree* as an index structure to achieve logarithmic time complexity for both incremental updates and data retrievals. Experimental results establish the scalability and efficiency of the proposed approach on various datasets.

1 Introduction

Most applications such as environmental studies, census databases and telecommunication systems generate large amounts of temporal data. The notion of time is critically involved in such applications and its semantics has been recently discussed to make decision-making queries in data warehouses more efficient. In fact, data items often have time-related attributes, e.g., time of a sales transaction or an order and the shipping date of a product.

Temporal datasets distinguish themselves due to the existence of one or more time dimensions. First of all, these time dimensions are usually of high cardinality, which implies that the datasets are sparse when taking the time dimension into consideration. Thus if we just consider the data at one time slice, say the records at one specific date in a daily transaction summary data cube, there might be only one record. Second, there is typically a correlation between the value of time attribute and other attributes of the data items incorporated into the data collections. For example, sales transactions are recorded in a timely manner and hence the earlier a sales transaction posted, the earlier the shipping date. Another example is that the stock price may increase over a period of time and may keep on falling during another period. We refer to such datasets as

^{*} This work has been supported by the NSF under grant numbers CNF-04-23336, IIS-02-23022 and EIA-00-80134.

append-only data [9] since the updates can only affect data items with the latest time coordinate. Third, time evolving data grow rapidly, thus requiring rapid integration into the data warehouses.

These characteristics of temporal datasets impose challenges to data analysis in MOLAP data cubes. The time dimension and its correlated dimensions lead to a high degree of sparsity in a traditional array-structure [14] based MOLAP data cubes, like the *prefix sum cube* [6] and *hierarchical cubes* [3]. Either the result induces huge storage costs because of the large cardinality of time-related dimensions [6], or large query overhead to answer time-parameterized range queries for a data cube is involved to retrieve all data points in the query range.

The temporal aggregation problem was studied in [12, 11, 13, 10] to address the challenges of temporal data. These works concentrate on one time-related attribute and most deal with time-interval data. In addition, Riedewald et al. [9] proposed efficient range aggregation in temporal data warehouses by exploiting the append-only property of the time-related dimension. In practice we also observe that many datasets have multiple time-related attributes and they usually have some semantic relationship. One such semantics is referred to as the Multi-Append-Only-Trend (MAOT) property in [8]. [8] studied the range aggregate queries over datasets with multiple append-only dimensions. The solution assumes some ε -bound to restrict the MAOT property in the datasets, i.e. two time-related attributes cannot deviate by more than ε from each other. In this paper, we show that such restriction is not necessary, and propose a general approach for efficiently answering range aggregate queries over several time-related dimensions in temporal data warehouses.

Multiple range aggregate queries make trend analysis possible. For instance, “*what is the total value of all orders in California which were ordered in the first half of July 2002 and shipped in August or later?*”. Our approach factors out the time-correlated dimensions to reduce the sparsity of MOLAP data cubes. For the time-related dimensions, we capture the local trend by partitioning points into blocks and index blocks with a *perfect binary block tree*. Our index structure allows efficient integration and aggregation of append-only data with logarithmic incremental update and search complexity. Moreover, they can be maintained in an online fashion, which may be used to provide real-time analysis for human analysts in massive data streaming applications [4].

The rest of the paper is organized as follows. In Section 2, a general model for MAOT datasets is presented. In Section 3, we propose data structures exploiting the semantics of time-correlated dimensions, on which the range aggregate query processing algorithm in Section 4 is based. In Section 5, we empirically evaluate our technique by using both synthetic and real datasets. Conclusions and future research work are provided in Section 6.

2 A Data Model

In this section, we propose a simplified and general data model for *Append-Only-With-Trend* datasets. Let \mathcal{D} denote a data set with d dimensional attributes $\delta_1, \dots, \delta_d$. Let (X^d, v) , $X^d = (x_1, \dots, x_d)$, refer to a data point in \mathcal{D} and its measure value is v . A multi-dimensional range query $\text{RQuery}(L^d, U^d)$ specifies a lower-bound query point

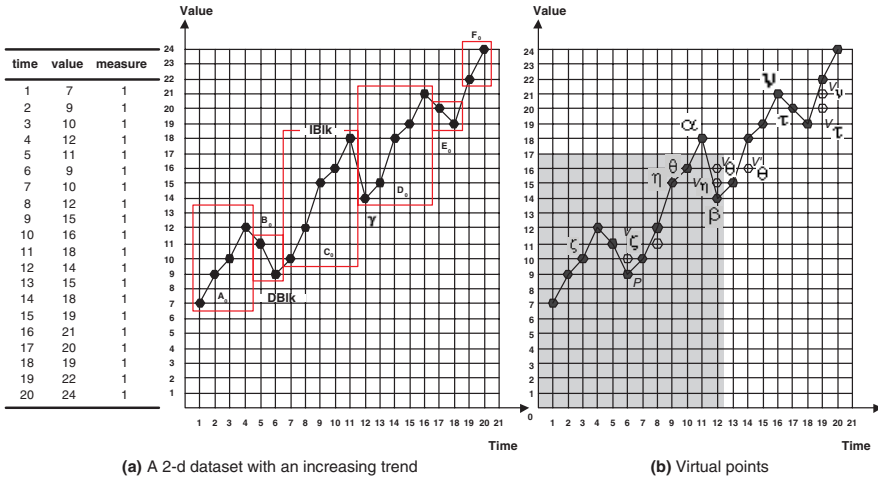


Fig. 1. Example of append-only-with-trend data sets

$L^d = (L_1, \dots, L_d)$ and an upper-bound query point $U^d = (U_1, \dots, U_d)$. The query selects all data points X^d that satisfy $L_i \leq x_i \leq U_i$ for all dimensions δ_i . A **range aggregate query** applies an aggregate operator (e.g. SUM) over the measure values of all the data points selected. It is called a **prefix range aggregate query** when the lower-bound query point is $(0, \dots, 0)$. The d -dimensional dataset \mathcal{D} is an **Append-Only-With-Trend** dataset if it has the following properties:

1. One of its dimensions, say δ_1 , is a *temporal dimension* (T-dimension). Data points are always appended on T-dimension.
2. d_v of its dimensions, say $\delta_2, \dots, \delta_{d_v+1}$, are *value dimensions* (V-dimensions), which are time-correlated dimensions. The V-coordinates maintain either a non-decreasing or a non-increasing trend approximately.
3. v is the measure dimension, on which the aggregate operators could apply.

The value dimension here also captures the *valid time dimension* in bitemporal databases [7]. The approximate trend implies that the probability that a data point is off the trend decreases as the distance increases.

Figure 1 shows an example of a two-dimensional append-only dataset, which exhibits an append-only trend on the time dimension and approximately increasing trend in the value dimension. $RQuery((0, 0), (12.5, 17))$ is a (prefix) range aggregate query whose T-dimension range is $[0, 12.5]$ and V-dimension range is $[0, 17]$.

In the paper, we mainly discuss the case when $d_v = 1$, that is, there is one V-dimension in addition to the temporal dimension, since there is an important class of applications with two time-related dimensions, for example, the bitemporal databases [7]. To simplify the following discussion, we assume that no two data points have the same coordinate in the T-dimension. We will discuss later how to deal with two data points with the same T-coordinates. Furthermore, our discussion is based on the aggregate operator of SUM. However it can be easily extended to incorporate other invertible aggregate operators such as COUNT and AVG [9].

3 Data Structures for Range Queries

In this section we present how to store and index two dimensional append-only-with-trend datasets for efficient range aggregate query processing. The main idea is to partition the dataset into monotonic blocks and then index these blocks using a *perfect binary block tree*.

3.1 Partitioning Data Points

Range queries on a sequence of data points can be performed efficiently using binary search if the data are monotonic in all dimensions. In order to exploit this monotonic property to support efficient range queries, we partition the datasets into blocks. Each block contains a sequence of consecutive data points with a non-decreasing or non-increasing trend. Particularly in terms of a two-dimensional append-only-with-trend dataset, the monotonic trend of a block can be differentiated w.r.t. the V-coordinates of data points as the T-coordinates always follow a non-decreasing trend. Depending on the monotonicity, each block can be categorized as either IBlk or DBlk, where IBlk represents a non-decreasing block and DBlk represents a non-increasing block.

In Figure 1(a), the data points are partitioned into six blocks according to the trend of their V-dimension values. Since the dataset follows an overall non-decreasing trend approximately in this example, there are more IBlk blocks than DBlk blocks, and DBlk blocks contain fewer data points than IBlk blocks.

Partitioning an append-only-with-trend dataset can be performed in an online manner. When a new data point is appended, it is compared with the most recent data point in the current block. If it does not follow the trend of the current block, the current block is ended and the new data point starts a new block. Each block contains at least two data points, since at least two data points determine a trend.

3.2 Maintaining Aggregate Information in Blocks

In order to avoid aggregation on-the-fly when answering range aggregate queries, we maintain cumulative information for each data point so that the range aggregate query can be answered by accessing a constant number of points [6]; Therefore each data point P maintains two kinds of aggregate information: (1) the aggregates of all data points occurring until data point P , named the *prefix sum* (PSUM); (2) the aggregates of those data points occurring until data point P with V-dimension values no greater than data point P 's V-dimension value, named the *partial prefix sum* (PPSUM).

Suppose there is a range sum query, whose T-dimension range is $[7,10]$ and V-dimension range is $[11, 17]$. The range sum query falls into block C_0 shown in Figure 1. Hence the query result is the difference between the PSUM of data point $(10, 16)$ and the PSUM of data point $(8, 12)$. For another example query $((0, 0), (8, 10))$, the PPSUM of data point $(7, 10)$ in Figure 1 is the answer.

From the above, we observe that if all the data points within the T-dimension query range are also within the V-dimension query range, the query can be answered using the PSUM of the data point closest to the T-dimension boundary. Otherwise, some of the data points within the T-dimension query range might jump out of the V-dimension query range. In this case, we need to find the data point closest to both the T-dimension and V-

dimension upper boundaries of a range aggregate query, so that we can use the PPSUM of that data point as an answer. However, it is possible that the data point closest to the T-dimension boundary is not closest to the V-dimension. For the query $((0, 0), (12.5, 17))$ in Figure 1(b), data point β is closest to T-dimension boundary 12.5. However, the earlier point θ is closer to the V-dimension boundary than β . So we introduce the notion of *virtual points* by mapping the V-coordinate of point θ at point β .

A *virtual point* at point P is the projection of an earlier data point at point P , whose V-dimension values fall in the interval between point P and its immediately previous point. For example, the virtual point V_θ at point β is the projection of point θ at point β . When the V-dimension values are monotonic w.r.t the T-dimension, no virtual points need to be kept. The smoother the data points, the less virtual points are kept. Figure 1(b) shows all the virtual points (represented by empty dots) added.

The following summarizes the aggregate information maintained for each data point and the aggregate information stored at an example block D_0 is shown in Figure 2.

- the PSUM of data point P : the aggregate information of all data points whose T-coordinates are no greater than the T-coordinate of P .
- the PPSUM of data point P : the aggregate information of all data points whose T-coordinates are no greater than the T-coordinate of P and V-coordinates are no greater than the V-coordinate of P .
- the V-coordinates of virtual points at point P : the V-coordinates of all data points which are earlier than data point P and whose V-coordinates are between the V-coordinate of P and the V-coordinate of the data point preceding P .
- the PPSUM of each virtual point at point P : the PPSUM of the virtual point whose T-coordinate is no greater than that of data point P and V-coordinate is maintained as above.

The aggregate information and virtual points above can be maintained in an online fashion. We keep a list of all V-dimension values. When a new data point arrives, we check the list to derive the virtual points and their partial prefix sums. The PSUM and PPSUM of the new data point can be computed by its own measure value and the aggregate information of its preceding point. The time complexity to maintain the information above for a new data point is bounded by $\log(n) + k$, where n is the number of data points and k is the number of virtual points in the interval. In worst case, the number of virtual points for the new data point is no more than $\min\{n, D\}$, where D is the cardinality of the V-dimension. If a data set follows some trend approximately, the actual virtual points are much fewer as analyzed later.

3.3 The Index Structure

We index blocks which contain data points using a *perfect binary block tree*. Each leaf node corresponds to a block with its summary information including the interval in the V-dimension and the end point, as well as a pointer to the block as shown in Figure 2. For example leaf node A_0 in Figure 2 corresponds to block A_0 in Figure 1(a).

Every two leaf nodes at level 0 are grouped together by an internal node at level 1. An internal node contains the union of V-dimension intervals of its child nodes. Every two consecutive nodes at level i are grouped together as a node at level $i + 1$ recursively until reaching the root node. For example, in Figure 2, the internal node A_1 contains

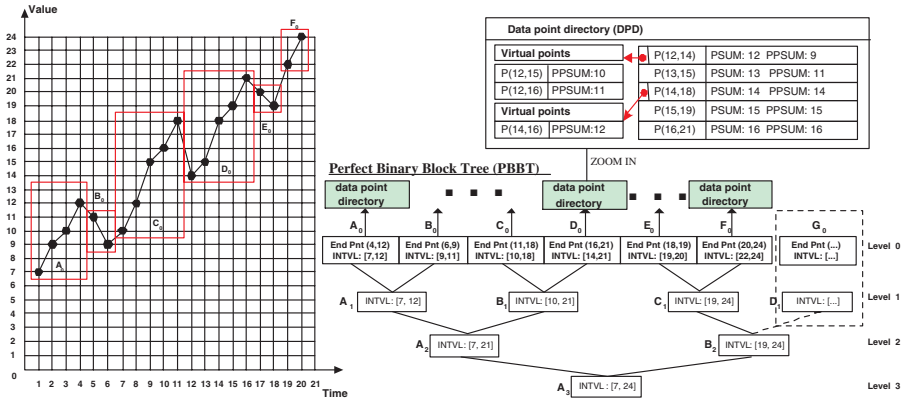


Fig. 2. Perfect binary block tree

the V-dimension intervals of data points in both block A_0 and block B_0 in Figure 1(a), which is the union of the V-dimension intervals of node A_0 and B_0 at the leaf level (level 0). It is possible that the last node at each level could be the only child node (left child) of its parent node, such as node C_1 shown in Figure 2, which is the only single left child node of node B_2 .

A *perfect binary block tree* (PBBT) has some nice properties since it is based on a perfect binary tree. The level of the PBBT tree is no more than $\lceil \log_2 N_B \rceil$, where N_B is the number of blocks. The total number of internal nodes is no more than N_B .

When a new block is appended as a new leaf node, the *perfect binary block tree* can be updated online as follows.

- If the last node in the leaf level is the only child of its parent, the new block is inserted as the sibling of the last node. Then the V-dimension interval of their parent node is updated accordingly to be the union of both nodes' V-dimension intervals. Then their parent node propagates the updates to their ancestor nodes level by level until the root node is reached.
- If every leaf node has a sibling, a new parent node is created in level 1 for the new block with the same V-dimension interval as that of the new block. Similarly this process is propagated level by level until the root node is reached. If a new node needs to be added to the topmost level, the number of levels is increased by one.

Since the total number of levels is $\lceil \log_2 N_B \rceil$, the time complexity to append a new block is logarithmic. For the example in Figure 2, if a new leaf node, G_0 , is added, a new node D_1 will be added as G_0 's parent. Thus node G_0 will be the only child of node D_1 which has the same V-dimension interval as node G_0 . If node D_1 is added at level 1, it is grouped with the only child node of C_1 and the V-dimension intervals of all their ancestor nodes will be updated accordingly to include node D_1 's V-dimension interval.

4 Range Aggregate Query Processing

We start with prefix range aggregate queries, represented as $((0, 0), (U_t, U_v))$ and extend to the general range aggregate queries later. The underlying idea of prefix query

processing is to find data points close to the upper boundaries of the query such that their aggregate information is sufficient to answer the query exactly. In particular, three data points need to be identified sequentially for a prefix query $RQuery((0, 0), (U_t, U_v))$.

- the *anchor point* is the data point whose T-coordinate is the largest but no greater than U_t . Note that its V-dimension value could be greater than or less than U_v .
- the *cross point* is the data point whose V-coordinate is on the other side of the V-dimension boundary w.r.t the anchor point and T-coordinate is closest but less than that of the anchor point. Thus if the anchor point is within the range, the cross point is out of query range and vice versa.
- the *closest point* is the virtual data point stored at the successor of the cross point, whose V-coordinate is closest to and within the V-dimension boundary. If it does not exist, it can be either the cross point or the successor as explained later.

4.1 Prefix Query Processing Algorithm

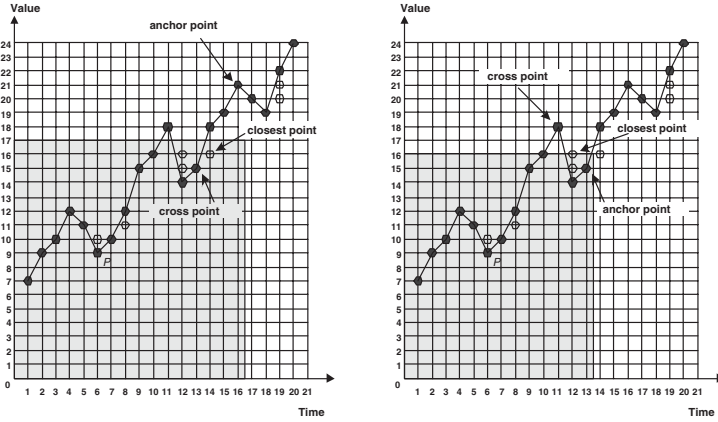
The general idea of the prefix range query algorithm is to find the closest point by finding the anchor point and its cross point. With respect to the relative position of its anchor point, prefix queries can be classified into two categories: (1) Anchor-out prefix query if the anchor point $P_a(a_t, a_v)$ is outside the V-dimension query boundary, i.e., $a_v > U_v$; (2) Anchor-in prefix query if the anchor point is below the V-dimension query upper boundary, i.e., $a_v \leq U_v$.

Anchor-Out Prefix Query Processing For this case, the cross point, P_{cross} , is the data point within the query having the greatest T-coordinate. It can be obtained by searching the perfect binary block tree as follows.

First we check if the start point of Blk_U is inside the query. If so, it means the cross point is in Blk_U and a binary search is performed to find it. Otherwise a bottom-up search process from Blk_U is performed as follows. From Blk_U in level 0, traverse the perfect binary block tree through the parent links until reaching a node which has a left sibling whose V-dimension interval covers U_v or the root node. If the root node is reached, there is *no cross point* and hence the query returns zero as the query result. Otherwise, a top-down search process is performed from the node's left sibling to identify the block Blk_C in level 0, whose V-dimension interval covers U_v . Note that in the top-down search process, the right branch of a node is always searched first as it contains points with greater T-dimension values. Once Blk_C is obtained, a binary search is performed in the data point directory of Blk_C to identify P_{cross} . The general algorithm for identifying the cross point from the anchor point is in Appendix.

If the cross point is close enough to the V-dimension boundary U_v , that is, no other point within the query range is closer to U_v , the cross point is chosen as the closest point and its PPSUM is used to answer the query. However, if there exists such data point within the range whose V-coordinate is closest to U_v , its V-coordinate must be within the V-dimension interval between the cross point and its successor and its T-coordinate must be smaller than the cross point. In this case, the data point is stored as a virtual point as described previously, and hence we choose the virtual point corresponding to the data point closest to U_v as the closest point.

Consider the prefix query $RQuery((0, 0), (16.5, 17))$ shown in Figure 3(a). Blk_U , block D_0 (Figure 1) is first identified by a binary search. Then the anchor point P_{anchor}



(a) An anchor-out prefix query

(b) An anchor-in prefix query

Fig. 3. Two types of prefix queries

(16, 21) is identified by a binary search in the data point directory maintained in D_0 . Since the V-coordinate of $P_{anchor}(16, 21)$ is greater than the upper bound of V-dimension, 17, it is an anchor-out prefix query. Now the cross point needs to be identified by traversing the perfect binary block tree (Figure 2) as described before. In this example, as the V-dimension interval of D_0 covers 17, traversing the tree is not needed. A binary search can be performed directly in the data point directory maintained in D_0 to identify the cross point, which is $P_{cross}(13, 15)$. Then virtual points at $P_{successor}$ whose V-coordinate is between the cross point and its next point are searched to identify the closest point, which results in a virtual point $P_{closest}(14, 16)$. The query returns $P_{closest}(14, 16)$.PPSUM as the result. The process is formalized in the prefix search algorithm in Appendix.

Anchor-In Prefix Query Processing Anchor-in prefix queries can be processed in a similar way to anchor-out prefix queries. The cross point is out of boundary in this case as shown in Figure 3(b) and can be found in the same way as before.

If the cross point exists, it is the data point which is outside the V-dimension range with the greatest T-coordinate. If no such cross point exists, then every data point preceding the anchor point must be within the given query range. Hence, P_{anchor} .PSUM gives the query result. The successor point of the cross point, $P_{successor}$, must have a V-coordinate less than or equal to the V-dimension upper bound. If any virtual point at $P_{successor}$, whose V-coordinate is the largest but less than the V-dimension upper bound, it is the closest point. Otherwise, $P_{successor}$ is the closest point.

As $P_{closest}$.PPSUM gives the aggregate information of all those data points preceding $P_{successor}$ (inclusive) within the query and $(P_{anchor}$.PSUM $- P_{successor}$.PSUM) gives the aggregate information of all those points after $P_{successor}$, the query answer is the $P_{closest}$.PPSUM + $(P_{anchor}$.PSUM $- P_{successor}$.PSUM). The processing of the example query shown in Figure 3(b) is similar to the example query in Figure 3(a). The complete prefix query algorithm is summarized in Appendix.

4.2 Discussion

Range Aggregate Queries Processing Given any d -dimensional range aggregate query, it can be decomposed into 2^d *prefix range aggregate queries* which select half-open ranges in all dimensions. Specifically, a 2-dimensional query $((L_t, L_v), (U_t, U_v))$ is decomposed into four prefix queries [6]: $((0, 0), (U_t, U_v)) - ((0, 0), (L_t, U_v)) - ((0, 0), (U_t, L_v)) + ((0, 0), (L_t, L_v))$. Since they share some boundaries, the processing of the four prefix queries can be further optimized. The details are omitted due to space limit.

The Complexity of the Algorithm We present the result of the algorithm analysis briefly here. The detailed proof is omitted due to the space limit. Let n denote the number of data points.

The time complexity of our range aggregate query algorithm is $O(\log(n))$. This is because only three data points are retrieved and the time complexity to search in the perfect binary block tree is bounded by $O(\log(n))$.

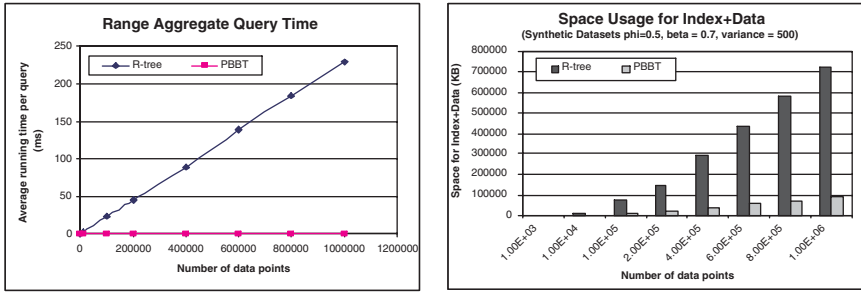
The perfect binary block tree, whose size is bounded by $O(n)$, usually could reside in memory. The block containing the data point of interest can be retrieved from disk. If the V-dimension follows a monotonic trend approximately, the disk space needed for the pre-computed aggregate information is $O(n)$ with high probability. Moreover, if the V-dimension values are from a finite domain, the space complexity is also $O(n)$. However, in the worst case, that is, the domain is infinite and the virtual points stored at each data point increase linearly over time, the disk space cost can be $O(n^2)$.

Data Points with Common T-coordinates The assumption that no two data points has the same T-coordinates can be relaxed as follows. Suppose k data points with the same T-coordinate T_i , and their next data point has T-coordinate T_{i+1} . If the current block at T_i is `IBlk`, that is, non-decreasing block, we order the k data points according to the ascending V-coordinates, otherwise, sort them in the descending order of V-coordinates. We replace the T-coordinates of ordered k data points with $T_{(i+1),1}, T_{(i+1),2}, \dots, T_{(i+1),k}$ accordingly. The new T-coordinates satisfy $T_i < T_{(i+1),1} < T_{(i+1),2} < \dots < T_{(i+1),k} < T_{i+1}$. Since the domain size on the T-dimension does not affect the complexity of the algorithm, the asymptotic cost is not affected by this transformation.

Limiting the Size of Each Block If a long sequence of data points follows the same trend, the block size might grow too large. The size of each block does not affect the logarithmic time bound of range aggregate query processing, but we need to load a large block into memory. In this case, we could split the block into several blocks by imposing a threshold on the number of data points in the block. Since our algorithm does not require two consecutive blocks to have different trends, the correctness of our algorithm still holds.

5 Experimental Evaluation

We perform experiments on both synthetic datasets and real datasets and compare our perfect binary block tree (PBBT) with R^* tree [1], which is the most popular multi-dimensional index structure and implemented in commercial RDBMS. Note that for a



(a) per query time vs. dataset size

(b) space required vs. dataset size

Fig. 4. Evaluating the impact of the dataset size

d -dimensional dataset, we assume two time-related dimensions. We measure the total space cost for both data and index structure, and the average per query time cost. We study how performance changes with varying dataset size and the degree of randomness, as explained in Section 5.1.

The implementations of both approaches are in Gnu C++. We use the R*-tree implementation by M. Hadjieleftheriou [5]. The experiments are performed on Mandrake Linux 9.0 running on AthelonXP1800 1533MHZ PC with 1G RAM and 50G disk. We assign the same amount of memory to both implementations for fair comparisons.

Two kinds of queries are used in the experiments: uniform queries and biased queries. The uniform queries are generated by choosing the T-dimension query range uniformly from t_{min} to t_{max} and the V-dimension range from v_{min} to v_{max} . The biased queries follow the distribution of data points, so that the query selectivity can be controlled. The query boundaries are generated using Gaussian distribution with the expected query range. The details of experimental setup are omitted due to space limit.

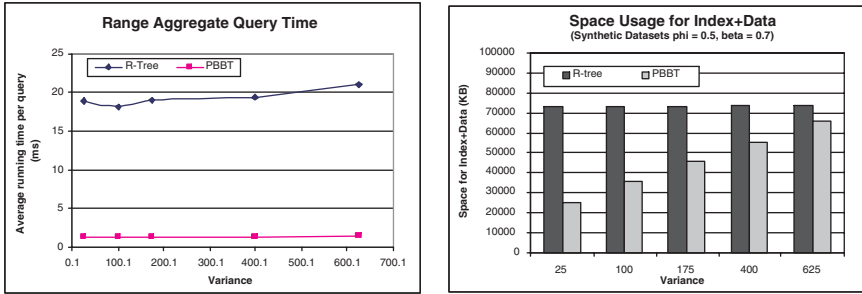
5.1 Experiments on Synthetic Datasets

We evaluate the effect of the dataset size and the degree of randomness using synthetic datasets with uniform queries, which contain queries with different levels of selectivity.

The synthetic datasets are generated from some popular time series models [2]: random walk and Auto-Regression model (AR(1)). We need to add a linear trend to the stationary AR(1) model to satisfy our append-only-with-trend data model. These two models can be captured with the equation below with different parameter settings:

$$V(t) = \phi * V(t-1) + \beta * t + \alpha_t$$

where α_t is a random variable following the normal distribution (a.k.a white noise) with mean 0. The variance of α_t represents how widely the random factor can go. When $\phi = 1$ and $\beta = 0$, it is a random walk. When ϕ is within the range $(-1, 1)$ and $\beta = 0$, it is a stationary AR(1) time series model. β represents the linear trend of the time series data. ϕ represents how the current value is correlated to the previous value. When $\phi = 1$ as in the case of a random walk, the data are smoother than the case of $|\phi| < 1$ in AR(1)



(a) per query time vs. randomness

(b) space required vs. randomness

Fig. 5. Evaluating the impact of randomness

model. The experimental results on random walk data always outperform the results for AR(1) model with linear trend. Given the lack of space, we only report the experiments on the latter case.

Figure 4 demonstrates the scalability of our approach (PBST) with increasing dataset size. The parameters in the synthetic dataset are fixed at $\phi = 0.5$ and $\beta = 0.7$ and the variance of the random variable is 500. We can see our approach is fairly scalable even when the dataset size increased from 1K to 1M, the per query time does not increase more than three times as shown in Figure 4(a). The space in Figure 4(b) increases linearly with the dataset size. This is consistent with our analysis of linear space and logarithmic time complexity. The query time grows linearly for R*-tree, since the number of nodes accessed is proportional to the dataset size in the case. Also the rate of increases in space for R*-tree is significantly larger than that of our approach.

The query time saving of our approach comes from two aspects: precomputed aggregate information and fewer disk I/Os resulting from constant number of disk accesses. We use the perfect binary block tree to decide the block of data that needs to load into memory, so the disk access time is constant per query if the binary tree can always be accommodated in memory. Since the perfect binary tree has the same number of internal nodes as blocks and each node contains only the boundary information, the total size of the index structure is actually small enough to fit in the main memory in practice.

Another factor we consider is the randomness of the data (Figure 5). We change the variance of the random variable α from 25, 100, 175, 400 to 625. More points will jump off the trend further with the increased randomness. In this case, more virtual points need to be stored and the space cost increases. However, since the query time complexity is logarithmic to the number of points as analyzed before, the query time deteriorates slightly with the increased randomness (variance of the trend).

5.2 Experiments on Real Dataset

We also evaluate the performance of the proposed approach on a real dataset: the daily industry average open price of the Dow Jones stocks for the last 74 years. It is available from <http://finance.yahoo.com/>. There are 18,656 data points in the dataset and the value

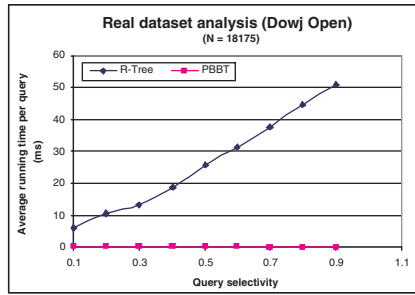


Fig. 6. per query time on stock price dataset

ranges from 41.63 to 11719.19. To maintain the precision to 0.01, we enlarge the domain size 100 times. The time-related dimensions are the date and the daily open price, while we use the stock volume as the measure dimension.

We evaluate the query performance with the selectivity using biased queries. The experiment results in Figure 6 show that our approach has fairly stable query time w.r.t query selectivity. The query time using R*-tree index increases linearly with the selectivity. The reason is that R*-tree needs to visit all data points in the range. In contrast, our approach (PBBT) only used 4M bytes disk space and R*-tree used more than 14M bytes. In summary, the experiments demonstrate the effectiveness of our approach in a variety of time series datasets.

6 Conclusion

We proposed an effective approach for range aggregate query processing in append-only datasets with two time-related dimensions. Our approach improves the MOLAP efficiency on the sparse append-only datasets by factoring out the two time-related dimensions and makes the query cost independent of the query length. A novel data structure, the perfect binary block tree, is proposed to allow logarithmic time complexity to append data and query processing. This idea can be extended to multiple time-correlated dimensions. However, this naïve extension of the two-dimensional algorithm compromises the performance efficiency. In this case, the logarithmic time complexity for query processing cannot be guaranteed. How to improve the efficiency on more time-related dimensions is one of our future works.

References

1. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 322–331, 1990.
2. P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, 2002.
3. C.-Y. Chan and Y.E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 675–686, 1999.

4. Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–334, 2002.
5. Marios Hadjieleftheriou. C++/java spatial index library. <http://www.cs.ucr.edu/~marioh/spatialindex/index.html>, 2004.
6. C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 73–88, 1997.
7. C. S. Jensen and et al. *Temporal Databases - Research and Practice*, volume 1399 of *LNCS*, chapter The Consensus Glossary of Temporal Database Concepts, pages 367–405. Springer Verlag, 1998.
8. Hua-Gang Li, Divyakant Agrawal, Amr El Abbadi, and Mirek Riedewald. Exploiting the multi-append-only-trend property of historical data in data warehouses. In *Proc. Int. Symp. on Spatial and Temporal Databases (SSTD03)*, pages 179–198, 2003.
9. M. Riedewald, D. Agrawal, and A. El Abbadi. Efficient integration and aggregation of historical information. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 2002.
10. Yufei Tao, Dimitris Papadias, and Christos Faloutsos. Approximate temporal aggregation. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2004.
11. Jun Yang. *Temporal Data Warehousing*. PhD thesis, Stanford University, 2001.
12. Jun Yang and Jeniffer Widom. Incremental computation and maintenance of temporal aggregates. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2001.
13. Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. Int. Conf. on Principles of Database Systems (PODS)*, 2001.
14. Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 150–170, 1997.