Efficient Integration and Aggregation of Historical Information*

Mirek Riedewald
University of California
Santa Barbara, CA 93106
mirek@cs.ucsb.edu

Divyakant Agrawal
University of California
Santa Barbara, CA 93106
agrawal@cs.ucsb.edu

Amr El Abbadi
University of California
Santa Barbara, CA 93106
amr@cs.ucsb.edu

ABSTRACT

Data warehouses support the analysis of historical data. This often involves aggregation over a period of time. Furthermore, data is typically incorporated in the warehouse in the increasing order of a time attribute, e.g., date of a sale or time of a temperature measurement. In this paper we propose a framework to take advantage of this append-only nature of updates due to a time attribute. The framework allows us to integrate large amounts of new data into the warehouse and generate historical summaries efficiently. Query and update costs are virtually independent from the extent of the data set in the time dimension, making our framework an attractive aggregation approach for append-only data streams. A specific instantiation of the general approach is developed for MOLAP data cubes, involving a new data structure for append-only arrays with pre-aggregated values. Our framework is applicable to point data and data with extent, e.g., hyper-rectangles.

1. INTRODUCTION

The notion of *time* plays an important role in data warehouses which are designed to support the analysis of historical business data. Data items often contain time-related attributes such as a date of a sales transaction or when a bill was paid. This enables analysts to examine revenue and expense trends over time. Important business queries compare the revenues of the different months of a year, or how a value for a month compares to the same month in other years. Similarly, statistical databases for environmental studies, census databases, and digital libraries all need to support a notion of time. Environmental studies draw conclusions about processes like global warming and the development of the ozone hole; census statistics examine trends like an increase of the number of people with College degrees, and so forth. The importance of time even resulted in a separate thread of research specifically concerned with *temporal databases* [7].

However, time is not only important as an attribute to describe a data item. Typically there is a correlation between the value of the time attribute and when the item is incorporated into the data collection. For instance, sales transactions and phone calls are recorded in a timely manner and hence the earlier a sales event or phone call took place, the earlier it will be recorded in the data warehouse. Similarly, sensors that measure environmental data report the measurements in timely order. We refer to such data sets as *append-only* data. Intuitively, a d-dimensional data set is append-only, if updates can only affect data items with the latest or a greater time coordinate. Hence, updates are restricted to a $(d-1)$-dimensional "time slice" with the greatest time coordinate. However, we also propose solutions for dealing with out-of-order updates that affect historic time slices.

A main characteristic of data collections that allow the analysis of historic data is that the amount of data grows, often rapidly, over time. Popular examples are massive *data streams* as produced by phone call and network monitoring devices, weather sensors all over the globe, and transaction processing systems of large retailers. To make these huge collections of detail data digestible for human analysts, support for efficient aggregation and summarization is vital. An important tool for exploring data collections is the orthogonal range aggregate query, for instance: "what is the overall revenue of all stores in New York over the last month". This query selects ranges on some of the attributes and computes aggregates over the selected data items. Similarly, roll-up and drill-down queries [5] that aggregate on different levels of granularity are often collections of related range queries.

In this paper we present a new framework for supporting efficient aggregation on append-only data sets. Our work concentrates on the important class of invertible operators like `SUM`, `COUNT`, and `average` (when maintained as `SUM` and `COUNT`). Essentially our construction makes the query and update costs independent of the extent of the data in the append-only dimension. Stated differently, no matter how long the recorded history is, the query and update costs are virtually unaffected.

We apply the methodology of the framework to MOLAP data cubes which are maintained in multidimensional arrays. Our construction involves a new technique for append-only arrays in main memory and in secondary memory (hard

*This work was partially supported by NSF grants EIA-9818320, IIS-98-17432, EIA-9986057, and IIS-99-70700.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

disk). It uses dynamic data structures for the currently appended time slice and gradually and adaptively changes the values of historic data items to support faster queries.

One might wonder, if array-based techniques are practically relevant since many real data sets are sparse, especially for high dimensionality. The answer is a clear yes! Business reports often require a high-level view of the data, e.g., sales data grouped by districts and product categories. Results of environmental studies represent summarized data like the ozone concentration on a latitude-longitude grid. Furthermore, aggregated (dense) summary information even plays a vital role in the analysis process for very sparse data sets. The aggregated information provides a quick overview of the underlying base data. This allows the analyst to identify regions of interest where to drill down, avoiding costly processing of irrelevant detail data. For example users of geographical map and image collections typically first browse the contents based on latitude, longitude before being able to construct educated queries that avoid zero-hit or million-result answers. All these are examples for dense, high-level views of underlying sparse detail data. Another example is the data cube operator as proposed by Gray et al. [10] which computes all possible 2^d group-bys for a set of d grouping attributes (all subsets of the set of d attributes are used for grouping). Even for very sparse base data, the groups with a small number of attributes are typically dense enough to be efficiently maintained in array structures. Instead of re-computing dense views from the huge base data from scratch, our approach enables efficient incremental maintenance.

The outline of the paper is as follows. In Section 2 we present the general framework for aggregation over append-only data sets. An instantiation of the framework for MOLAP data cubes is developed in Section 3. Section 4 shows how we can apply our framework to a wide variety of applications by using multiversion data structures. We report the results of an extensive experimental evaluation in Section 5. Section 6 discusses related work. Concluding remarks and open problems are presented in Section 7.

2. A GENERAL FRAMEWORK

In this section we develop a framework to support the efficient incorporation of historical information as well as answering aggregate queries. The framework is quite general and does not assume any particular storage structure for the underlying data, e.g., MOLAP or ROLAP data. This section also introduces important terminology that will be used throughout the paper.

2.1 Terminology

With $\mathcal{D}$ we denote a data set with d dimension attributes $\delta_1, \delta_2, \ldots, \delta_d$ and a single measure attribute m. Note, that our technique easily generalizes to data sets with multiple measure attributes. Let (X^d, v), $X^d = (x_1, x_2, \ldots, x_d)$, refer to a point in $\mathcal{D}$ and its measure value v. A point *exists* in the data set, iff its measure value is not NULL. A multidimensional range query specifies a range (L_i, U_i) in each dimension δ_i, the selection possibly being a single value or the entire domain. The query selects all data points X^d that satisfy $L_i \leq x_i \leq U_i$ for all dimensions δ_i.

The set $\mathcal{D}$ is *append-only*, iff one of its dimensions, say δ_1, is a *transaction time dimension* (TT-dimension). Dimension δ_1 is a TT-dimension, iff the following holds for updates to $\mathcal{D}$. If an update to point X^d arrives at the data set before another update to a point Y^d, then $x_1 \leq y_1$, i.e., the earlier update affects a point with a less or equal coordinate in dimension δ_1.

A time value $t \in \text{domain}(\delta_1)$ in the TT-dimension is an *occuring* time value iff there is a point $(x_1, x_2, \ldots, x_d)$ in the data set such that $x_1 = t$. This notion is important because our technique only generates data structures for occuring time values.

2.2 The Basic Technique

For simplicity the technique is illustrated for a two-dimensional data set $\mathcal{D}$ and the operator `SUM`. Let the dimensions be `time` and `location`. Transactions that commit at a certain location and time update the value of the measure attribute (e.g., a sales amount). Records arrive in the order the transactions commit, i.e., the time dimension is a TT-dimension.

We can model $\mathcal{D}$ as a collection of one-dimensional "time slices", i.e., columns of location coordinates for each time coordinate. Updates can either affect a point in the slice with the greatest time coordinate, or append a new time slice with a greater time and insert a new point there. Let R_1 denote a data structure that supports range queries on a one-dimensional data set whose dimension is `location`, e.g., a B-tree with `location` keys. For each occuring time value t we construct an instance $R_1(t)$ of R_1. $R_1(t)$ maintains all points $X^2 = (x_1, x_2)$ of data set $\mathcal{D}$ whose time coordinate x_1 satisfies $x_1 \leq t$. Hence the instances of R_1 contain *cumulative* knowledge about the data points. Figure 1 shows an example. The data set and its corresponding instances of R_1 are shown for each step while processing a sequence of updates. Note that $R_1(1)$ and $R_1(3)$ do not change any more as soon as a point with greater time coordinate is appended.

This cumulative knowledge enables us to reduce a two-dimensional range query to two one-dimensional range queries as follows. For our example data set let the query compute the sum over all points (x_1, x_2) that satisfy $2 \leq x_1 \leq 4$ and $3 \leq x_2 \leq 5$. We first perform the location range query $3 \leq x_2 \leq 5$ on $R_1(4)$ which is the instance of R_1 with the greatest time value which is less than or equal to the upper value of the selected time range. This query returns 13 (cf. Figure 2), which is the sum over all points whose time coordinate is less than or equal to 4 and whose location coordinate is within the selected range. To answer the actual range query, we have to remove all points whose time coordinate is less than 2. Hence we also perform the location range query on $R_1(1)$ and subtract the result from the initial value, obtaining the correct result of 6.

Note that each single instance $R_1(t)$ by itself can only answer range queries that select half-open time ranges, i.e., "all points in $\mathcal{D}$ whose time coordinate is less than or equal to t". We will use the term *prefix time query* for this type of range queries. Hence our technique reduces any two-dimensional range aggregation query on $\mathcal{D}$ to two one-dimensional prefix time queries. Note that the way the prefix time queries are combined is reminiscent of the Prefix Sum technique of [12]. However, in contrast to prefix-sums which specify half-open ranges in *each* dimension, a prefix time query only specifies a half-open range in the append-only time dimension.

To perform an update, first the appropriate instance of R_1 is selected. Since the data set is append-only, this instance is the one with the highest time coordinate. In the example

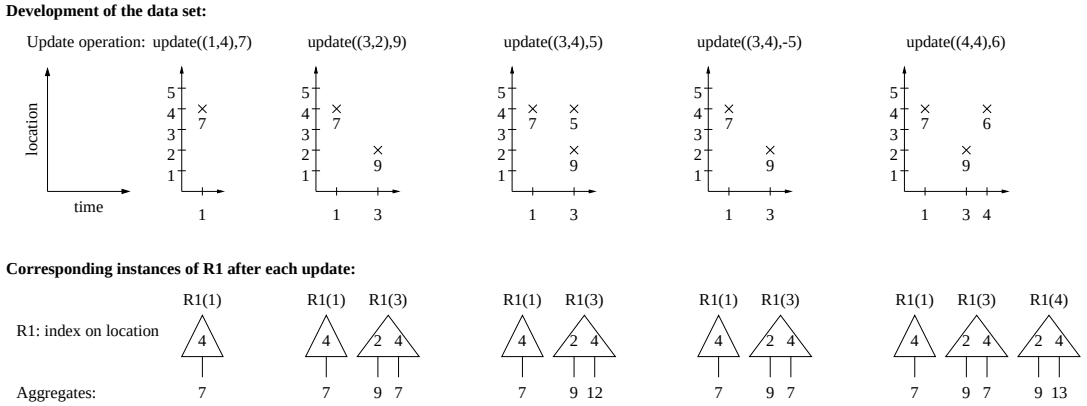


Figure 1: Data set and corresponding data structures

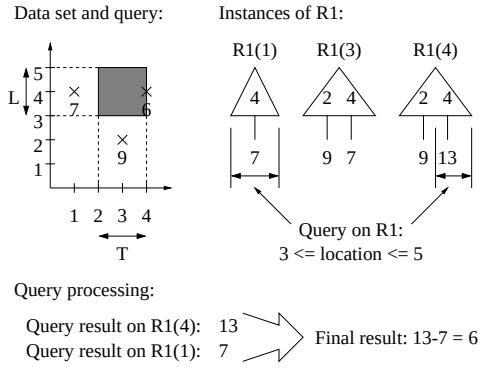


Figure 2: Translation of a two-dimensional range query on $\mathcal{D}$ to two one-dimensional range queries on the cumulative data set

in Figure 1 the next update will have time coordinate 4 or greater. If an update affects a point with a time coordinate that is greater than for the previous update, a new instance of R_1 is created by copying the latest one. In Figure 1 the second update shows an example where the new instance $R_1(3)$ is created. The copying can be quite expensive and results in high redundancy for sparse data sets. We discuss efficient approaches in the particular context of the underlying data models in Sections 3 and 4. Note, that updates to points with historic time coordinates cause a cascade of updates to all instances of R_1 with greater time coordinates. We discuss in Section 2.5 how to deal with such out-of-order updates.

2.3 Formal Description and Analysis

Let $\mathcal{D}$ be a d-dimensional point data set where dimension δ_1 is a TT-dimension. Let R_{d-1} denote an arbitrary data structure for $(d-1)$-dimensional data points that supports update and query operations as specified in Table 1. Examples of recent data structures with specific support for aggregate range queries can be found in [19, 16]. X^{d-1} refers to a data point, represented as a $(d-1)$-dimensional vector of dimension values. The update operation changes the measure value of point X^{d-1} by the value Δ. Insertions and deletions of points are translated to the corresponding update operation. The actual translation depends on the aggregate operator, but is straightforward and hence not discussed here.

For each occuring time instant $t \in \text{domain}(\delta_1)$ there is an instance $R_{d-1}(t)$ that maintains the $(d-1)$-dimensional projections of all points $Y^d \in \mathcal{D}$ where $y_1 \leq t$. On the original d-dimensional data set $\mathcal{D}$ we have to support updates and queries as shown in Table 2. As before insert and delete operations are translated to the corresponding updates. Given R_{d-1}, the operations on $\mathcal{D}$ can be implemented by performing corresponding operations on instances of R_{d-1} for the appropriate times.

Recall, that the time coordinate x_1 of an updated point X^d has to be at least as large as the time coordinate, say t, of the previous update. If x_1 is equal to t, the update affects an existing instance (the latest, to be precise) $R_{d-1}(t)$ of R_{d-1}. Hence the corresponding operation $\text{update}_{R_{d-1}}((x_2, x_3, \ldots, x_d), \Delta)$ is performed on $R_{d-1}(t)$. If an update has a time coordinate x_1 that is greater than t, a new instance $R_{d-1}(x_1)$ is created by copying $R_{d-1}(t)$ and proceeding as described above. For now we simply assume that there is a way of generating the copy in constant time. In Sections 3.3 and 4 we will discuss possible solutions that are equivalent to this constant-time copy process. Hence in total the cost of an update to $\mathcal{D}$ is equal to the cost of performing the update on the latest instance of R_{d-1}.

Operation $\text{query}_{\mathcal{D}}(L^d, U^d)$ is implemented as follows. First the appropriate instances $R_{d-1}(t_l)$ and $R_{d-1}(t_u)$ are selected. Time t_l is the greatest occuring time coordinate that is smaller than l_1 (lower query value in the TT-dimension), t_u is the smallest occuring time coordinate that is greater than or equal to u_1 (upper query value in the TT-dimension). With q_u and q_l we denote the result of $\text{query}_{R_{d-1}}((l_2, l_3, \ldots, l_d), (u_2, u_3, \ldots, u_d))$ on $R_{d-1}(t_u)$ and $R_{d-1}(t_l)$, respectively. Hence, the result of $\text{query}_{\mathcal{D}}(L^d, U^d)$ is computed as $q_u - q_l$. Consequently the cost of the d-dimensional query on $\mathcal{D}$ is the sum of the costs of the two $(d-1)$-dimensional queries on instances of R_{d-1}.

So far, our analysis of update and query cost has not taken into account the cost for finding the appropriate instances of R_{d-1}. We need a directory that maintains the correspondence between time values and instances of R_{d-1}. One can use standard one-dimensional data structures for this purpose, e.g., a B-tree for a sparse or an array for a dense TT-dimension. Since updates only affect the latest instance of

Operation	Comments
update$_{R_{d-1}}(X^{d-1}, \Delta)$	adds Δ to the measure value of point X^{d-1}
query$_{R_{d-1}}(L^{d-1}, U^{d-1})$	returns aggregate of the measure values of all points in the given bounding box (L is the lower, U the upper corner of the rectangle)

Table 1: Operations supported by the $(d-1)$-dimensional data structure R_{d-1}

Operation	Comments
update$_{\mathcal{D}}(X^d, \Delta)$	adds Δ to the measure value of point X^d
query$_{\mathcal{D}}(L^d, U^d)$	returns aggregate of the measure values of all points in the given bounding box (boundary included)

Table 2: Operations to be implemented for $\mathcal{D}$

R_{d-1}, we also maintain a pointer to this instance to guarantee constant time lookups for updates. Note that the pointer is easily maintained in constant time. For a range query we have to perform two lookups of the directory. The cost of a lookup is at most logarithmic in the number of occuring time values.

Since the number of occuring time values can never exceed the number of data points in $\mathcal{D}$, a directory lookup costs at most $\log n$, where n denotes the number of data points in $\mathcal{D}$. Typically the cost of a query on an instance of R_{d-1} (a $(d-1)$-dimensional range query) will be greater than the directory search cost. Based on this assumption our technique is *optimal* in the following sense. Query and update costs for a d-dimensional append-only data set $\mathcal{D}$ are only by a constant factor more expensive than on the $(d-1)$-dimensional projection that is obtained by removing the TT-dimension δ_1. Hence, solving a d-dimensional aggregation problem is essentially reduced to a $(d-1)$-dimensional problem on a projection of the data set. If the above assumption does not hold, i.e., the directory search cost dominates the query cost on R_{d-1}, we would have a data structure that can answer d-dimensional range queries at a cost which is logarithmic in the number of data points–a truly great result by itself.

2.4 Dealing with Extent

For simplicity our framework was described for point data, i.e., a data item is a d-dimensional point with a measure value. In practice data items could be objects that have an extent in multiple (if not all) dimensions. For instance, a map or satellite image covers an area rather than a point. Temporal databases often contain objects which have a certain life time, etc. Our framework covers all these cases.

If an object has extents in any subset of the dimensions $\delta_2, \delta_3, \ldots, \delta_d$ (i.e., any dimension except the TT-dimension), the solution is simple. One can pick any efficient $(d-1)$-dimensional data structure R_{d-1} that supports queries and updates of objects with extent [24]. Then our construction can be applied as described in Section 2.3, using the same query and update transformations.

In the most general case objects could also have an extent in the TT-dimension, i.e., their time "coordinate" is actually an interval which for instance indicates when the object is valid. Our approach as described so far could not answer queries that look for objects whose time intervals intersect, contain, or are contained in a certain query time interval.

We address this problem by using a reduction similar to Zhang et al. [23]. Instead of a single instance $R_{d-1}(t)$ for time t we use two instances $C_{d-1}(t)$ and $B_{d-1}(t)$. $C_{d-1}(t)$ maintains all objects whose time interval *contains* t, while $B_{d-1}(t)$ maintains all objects whose time interval ends *strictly before* t. We explain for the operator `COUNT` how these structures are used to find the number of objects that intersect a time interval $(t_{\text{low}}, t_{\text{up}})$. The query is answered based on the following observation. The number of objects whose time intervals intersect the query interval is equal to $b(t_{\text{up}}) + c(t_{\text{up}}) - b(t_{\text{low}})$. Expressions $b(t_{\text{up}})$ and $b(t_{\text{low}})$ denote the number of objects whose time intervals end strictly before t_{up} and t_{low}, respectively. The term $c(t_{\text{up}})$ refers to the number of objects whose time intervals contain t_{up}. The reader might easily convince her/himself that this equation is correct. Value $b(t_{\text{up}})$ is obtained from $B_{d-1}(\bar{t}_{\text{up}})$ where $\bar{t}_{\text{up}}$ denotes the greatest occuring time that is less than or equal to t_{up}. Similarly, we find $b(t_{\text{low}})$ in $B_{d-1}(\bar{t}_{\text{low}})$ where $\bar{t}_{\text{low}}$ is the greatest occuring time that is less than or equal to t_{low}. $c(t_{\text{up}})$ is obtained from $C_{d-1}(\bar{t}_{\text{up}})$.

Other invertible aggregate operators and containment queries are handled similarly. Using B_{d-1} and C_{d-1} we can answer d-dimensional aggregate range queries on $\mathcal{D}$ with objects that have an extent in the TT-dimension. The query cost increases only slightly. A d-dimensional aggregate range query now requires three instead of two $(d-1)$-dimensional queries (two accesses to B_{d-1}, one access to C_{d-1}). The update cost approximately increases by the same ratio as the following analysis shows. Let t_0 and t_1 be start and endpoint of the time interval of an object. At time t_0 the insertion of the object triggers an insertion to $C_{d-1}(t_0)$. Then at time t_1 the end of the interval triggers a delete operation in $C_{d-1}(t_1)$ and an insert to $B_{d-1}(t_1)$. The storage consumption approximately doubles.

2.5 Dealing with Out-of-Order Updates

The discussion so far has relied on the fact that all updates are append-only. In practice this might not be the case, e.g., when sales are registered late or when historic values are corrected. A nice property of our approach is that it can be extended such that the performance degrades gracefully with an increasing percentage of out-of-order updates.

An out-of-order update by definition affects a historic time slice. Due to the cumulative nature of our construction, an update to point $U^d = (u_1, u_2, \ldots, u_d)$ (i.e., with value u_1 in the TT-dimension) has to be propagated to all instances $R_{d-1}(t)$ with $t \geq u_1$. Instead of applying these potentially expensive updates instantly, we maintain them in a general d-dimensional data structure G_d that supports

the operations in Table 2 without taking advantage of the append-only property. Note that G_d and R_{d-1} are drawn from the same "pool" of data structures, well-known examples being R-tree and X-tree. For an out-of-order update $\text{update}_{\mathcal{D}}(X^d, \Delta)$ G_d stores the d-dimensional point (X^d, Δ). Append-only updates are processed as discussed in Section 2.3 without affecting G_d. Queries first compute an initial result based on the algorithm presented in Section 2.3. Then they are post-processed to take the effect of the out-of-order updates into account. This is done by answering the original range query using G_d and adding the obtained result to the initial value.

Note that the number of points in G_d is equivalent to the number of out-of-order updates and that the query and update cost of G_d are the costs for a general d-dimensional data set. Hence with an increasing percentage of out-of-order updates, query and update costs converge to the corresponding costs on a general d-dimensional data set which is not append-only. This gives us a technique that automatically takes advantage of append-only updates and which gracefully degrades to a general technique if the updates are general.

One can reduce the query cost, especially for data sets with a small percentage of out-of-order updates, by letting an asynchronous background process apply updates that are stored in G_d to the appropriate instances of R_{d-1}, beginning with the latest instance to avoid that the process "chases" newly created time slices. Details are beyond the scope of this paper.

3. APPLYING THE FRAMEWORK TO MOLAP DATA CUBES

Having discussed the general framework, this section develops an instance for MOLAP data cubes. The underlying assumption is, that the data set is maintained in a multi-dimensional array. We will explain our technique for the aggregate operator `SUM`.

3.1 MOLAP Aggregation Techniques

There exists a variety of work that aims at speeding up aggregate range queries for invertible operators on MOLAP data cubes, e.g., [12, 4, 8, 20]. The common idea is to generate a certain tradeoff between query, update, and storage cost by carefully selecting aggregates that are pre-computed and materialized to reduce the amount of on-the-fly computation. The technique by Riedewald et al. [20] generalizes and improves on earlier work and provides a variety of query-update cost tradeoffs. Since this technique allows the combination of different aggregation techniques for the dimensions, it constitutes an ideal basis for implementing our framework which requires the TT-dimension and the other dimensions to be treated differently.

The main idea is as follows. For each dimension a (possibly different) *one-dimensional* pre-aggregation technique is selected. Such an aggregation technique replaces the values in the cells of a one-dimensional array by sums over some cells. By iterating through all dimensions and applying the selected aggregation technique to all one-dimensional vectors "along" that dimension, the original array containing $\mathcal{D}$ is transformed to an array of the same size that contains pre-computed aggregates instead of the original values. We will summarize one-dimensional pre-aggregation techniques

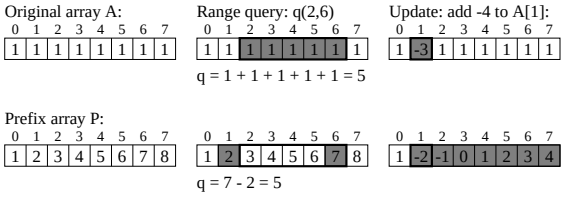


Figure 3: Original array A and prefix array P

that are relevant for this work and then illustrate the technique. For details the interested reader is referred to [20].

Let A be a one-dimensional array with N cells: $A[0]$, $A[1]$, $\ldots$, $A[N-1]$. The aggregate range query $q(l, u)$ computes $\sum_{i=l}^{h} A[i]$. Clearly, a query accesses N cells in the worst case. An update to A that changes the value in a cell affects only this one cell. By replacing each value $A[k]$ by the corresponding *prefix sum* $P[k] = \sum_{i=0}^{k} A[i]$ we can generate a prefix array P. This is referred to as the Prefix Sum Technique (PS) [12]. Using PS we can compute any query $q(l, u)$ as $P[u] - P[l-1]$ (setting $P[-1] = 0$ for notational convenience). Hence, the query cost is reduced to at most two cell accesses. On the other hand an update to $A[i]$ would affect all $P[j]$ where $j \geq i$, i.e., N cells in the worst case. Figure 3 illustrates the different query and update behavior for A and P. The cells that are accessed for processing a query and an update are shaded. The selected cells in the original data set are indicated with boxes.

Arrays A and P represent the two extremes of a variety of tradeoffs between query and update costs. To balance query and update costs [8] proposed a technique called Dynamic Data Cube (DDC). We use a variation of this approach to generate the pre-aggregated array D from A. The technique stores the sum of all values $A[i]$, $0 \leq i \leq N-1$, in $D[N-1]$. In $D[(N-1)/2]$ the sum of all values $A[j]$ in the left half, i.e., $0 \leq j \leq (N-1)/2$, is stored. Then the left and right subarray (excluding $D[(N-1)/2]$ and $D[N-1]$, respectively) are processed recursively as follows. The middle of the subarray contains the sum of the cells $A[i]$ in the left half of this subarray, and so on. The recursive processing establishes a hierarchy of the cells. Figure 4 shows an example where the boxes in the DDC computation tree indicate the cells whose values are added to obtain the corresponding value in D. Any prefix sum $P[k]$ can be computed by conceptually descending the hierarchy until the node with index k is found. On the path a node value contributes to $P[k]$ iff the node's index is less than or equal to the query index. Hence at each level at most one node is accessed resulting in a worst case cost of $\log_2 N$ cell accesses to compute any $P[k]$ using D. As described for PS array P, any general range query can be reduced to two prefix queries, therefore the worst case query cost using D is $2 \log_2 N$. In the example in Figure 4 we obtain $q(2, 6) = P[6] - P[1] = (D[3] + D[5] + D[6]) - D[1]$. Similarly it can be shown that an update affects at most $\log_2 N$ cells.

As proved in [20], multiple one-dimensional techniques can be combined to compute pre-aggregated *multidimensional* arrays. The strength of the approach is that the one-dimensional techniques also provide a simple way of finding all cells in a pre-computed array that have to be accessed for a query or an update. The indices of accessed cells (and possibly a factor like -1 depending on the pre-aggregation

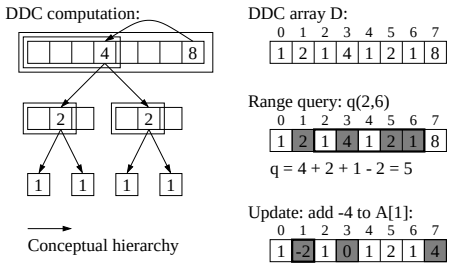


Figure 4: DDC array D

technique) are computed for each dimension independently. The solutions are combined by generating the cross product over all result sets and multiplying the corresponding factors.

3.2 Exploiting the Append-Only Property

As discussed in Section 2.3 our technique maintains each time slice with time coordinate t in a separate instance $R_{d-1}(t)$ of a $(d-1)$-dimensional data structure. Since we are now dealing with arrays, $R_{d-1}(t)$ is also an array whose cells might contain pre-aggregated values as discussed in the previous section. The cumulative construction of the $R_{d-1}(t)$ is equivalent to using PS as the pre-aggregation technique for the TT-dimension. Recall also, that updates only affect the last instance. Hence, historic instances should be optimized for query support, while the latest instance has to support queries and updates efficiently. This makes PS the ideal choice for all dimensions in historic instances, while DDC is the best choice for dimensions δ_2, δ_3, ... ,δ_d in the latest instance. Figure 5 illustrates this data structure. Since the one-dimensional PS technique has a worst case query cost of two cell accesses, any query on the historic part accesses at most 2^d cells. The resulting query cost therefore is independent of the domain sizes and the size of the query region. A query on the latest instance of R_{d-1} costs at most $\prod_{i=2}^{d}(2\log_2 N_i)$ cell accesses where N_i is the domain size in dimension δ_i (based on a query cost of $2\log_2 N_i$ in each dimension due to using DDC). For simplicity and w.l.o.g. we will further on assume that all N_i are equal to N. Hence the worst case query cost in the latest instance is $(2\log_2 N)^{d-1}$. Similarly, the update cost is bounded by $(\log_2 N)^{d-1}$.

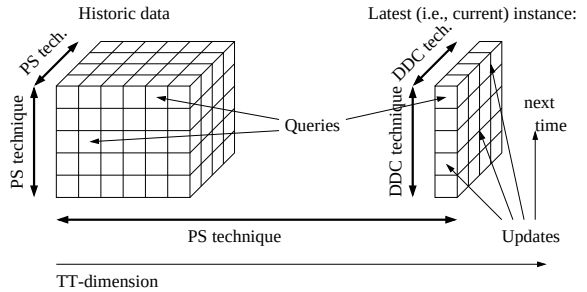


Figure 5: Ideal combination of pre-aggregation techniques for the append-only data set

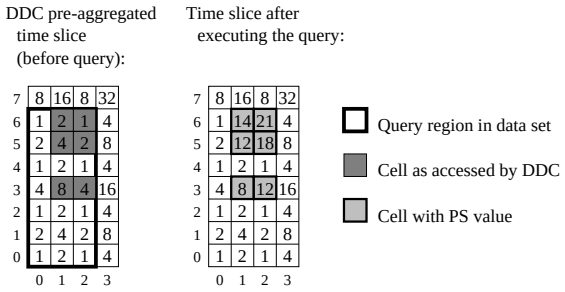
If an update affects a point X^d whose time coordinate x_1 is greater than the time coordinate t of the latest instance $R_{d-1}(t)$, $R_{d-1}(t)$ is copied to create a new instance $R_{d-1}(x_1)$ (cf. Section 2.3). We will describe a technique for making a copy in an efficient manner in the next section. Here we focus on another problem. When $R_{d-1}(x_1)$ replaces $R_{d-1}(t)$ as the latest instance, $R_{d-1}(t)$ now belongs to the historic data. Hence it should be transformed from using DDC to using PS for dimensions δ_2, δ_3, ... ,δ_d. The straightforward solution is to perform the transformation as soon as $R_{d-1}(t)$ becomes historic. However, this would introduce a large overhead (in the order of the size of a whole time slice) to an update that creates a new instance. The benefit of eagerly transforming all values is also questionable, since queries might not access all cells.

We present a more elegant solution that does not require an instantaneous transformation, but instead gradually lets *queries* change the pre-aggregation scheme with a small overhead per operation. We refer to the resulting data cube as the *Evolving Data Cube* (eCube). The main idea is to allow both PS and DDC pre-aggregated values to coexist in the historic instances of R_{d-1}. More precisely, a cell of a historic instance contains a measure value which is either a pre-aggregated value based on using DDC in dimensions δ_2, δ_3, ... ,δ_d, or it is a pre-aggregated value based on using PS in all these dimensions (note, that in dimension δ_1 PS is always used). To distinguish between the cases, a bit is added and used as a flag.

eCube mixes PS and DDC in a single array and hence can not use the standard DDC and PS query algorithms. We developed a combination of the two algorithms that is also defined based on the one-dimensional case. We discuss a two-dimensional eCube example to give a flavor of the approach. Figure 6 shows a time slice with the initial DDC values (obtained for an original array that contained value 1 in each cell). The algorithm is shown for a *prefix* range query that selects range $(L^2, U^2) = ((0,0),(2,6))$. First the cell in the upper right corner is checked if it contains a PS value. If yes, the algorithm could stop and return the value. Here the value is DDC aggregated, therefore the algorithm computes the addresses of all cells that would be accessed by the DDC query algorithm (shaded in left array). Then it uses the PS query algorithm to compute PS(2, 6) from the "neighboring" DDC cells PS(1, 6), PS(2, 5), and PS(1, 5). These PS values are computed recursively in the same way. After computing a PS value, it replaces the former DDC value in a cell. Subsequent queries can take advantage of these PS values and save some if not all recursive calls. In the example, if the next query computes the sum for range $((0,0)(2,3))$ it returns after the first cell access. Note, that the new eCube query algorithm accesses in the worst case as many cells as DDC.

The above algorithm easily generalizes to instances with $(d-1)$ dimensions. General range queries are answered as follows. First the PS algorithm is used to reduce the general range query to at most 2^d *prefix* queries. If the accessed cells contain DDC values, the corresponding PS value is computed recursively using the PS technique as described above. Note, that the indexes for this recursive computation are restricted to the sets of indexes the DDC technique returns for a certain query. For instance in Figure 6 the value PS(2, 5) is computed from PS(1, 5), PS(2, 3), and PS(1, 3) because the DDC technique returned index sets $\{1, 2\}$ and $\{3, 5, 6\}$ for the first and second dimension, respectively. Alternative computations, e.g., based on PS(1, 5), PS(2, 4), and PS(1, 4) were therefore not considered.

The advantage of combining DDC and PS in the described



Steps while processing the query:

PS(2,6): value is DDC => PS(2,6) = PS(1,6)+PS(2,5)-PS(1,5)+DDC(2,6)
  PS(1,6): value is DDC => PS(1,6) = PS(1,5)+DDC(1,6)
    PS(1,5): value is DDC => PS(1,5) = PS(1,3)+DDC(1,5)
      PS(1,3): value is DDC => PS(1,3) = DDC(1,3) = 8, mark as PS
    Return: PS(1,5) = 8+4 = 12, mark as PS
  Return: PS(1,6) = 12+2 = 14, mark as PS
  PS(2,5): value is DDC => PS(2,5) = PS(1,5)+PS(2,3)-PS(1,3)+DDC(2,5)
    PS(1,5): marked as PS, value = 12
    PS(2,3): value is DDC => PS(2,3) = PS(1,3)+DDC(2,3)
      PS(1,3): marked as PS, value = 8
    Return: PS(2,3) = 8+4 = 12, mark as PS
    PS(1,3): marked as PS, value = 8
  Return: PS(2,5) = 12+12-8+2 = 18, mark as PS
  PS(1,5): marked as PS, value = 12
Return: PS(2,6) = 14+18-12+1 = 21, mark as PS

Figure 6: eCube query processing

way is that the resulting $(d-1)$-dimensional data structure for R_{d-1} has the same worst case query cost of $(2\log_2 N)^{d-1}$ as a DDC pre-aggregated array. However, at the same time it supports PS and DDC values in the same structure and transforms the values to PS. Hence the query cost gradually converges towards the PS cost of 2^{d-1} in the worst case. Since only accessed cells are transformed, the actual transformation does not incur any access overhead. At the same time the technique automatically *adapts* to query patterns. When multiple queries hit a certain region, the values are changed to PS and thus considerably speed up all subsequent queries to the same region.

3.3 Copying Instances of R_{d-1} Efficiently

When an update affects a cell X^d whose time coordinate is greater than the time of the latest instance $R_{d-1}(t)$, a new instance $R_{d-1}(x_1)$ has to be created by copying $R_{d-1}(t)$. The straightforward implementation would add $O(N^{d-1})$ cell accesses to the original update cost. To avoid such "copy-bursts" we propose a new strategy for in-memory arrays which can be easily extended to support arrays in external memory (hard disk). The main idea is to amortize the copy cost by distributing it over multiple operations. The justification behind this approach is that arrays are only used if the data set is not "too sparse". Hence in the average there are several updates per time slice which can share the copy cost.

For the sake of simplicity the technique is presented for a two-dimensional array. As usual δ_1 (`time`) is the TT-dimension. Recall, that the PS technique is used for this dimension, while DDC is used for dimension δ_2, say `location`, in the latest instance of R_{d-1}.

For each *historic* time slice a consecutive area that holds the measure values of the N cells is allocated in memory. Apart from that an array `cache` of size N is maintained. This array contains the latest value for each location together with a time stamp. Figure 7 shows an example that begins at an intermediate step after processing some earlier updates. The time stamps in `cache` indicate the time of the last update to that location coordinate. For instance, initially location 0 was last updated to value 3 at time 0, therefore `cache` contains the value 3 and time stamp 0 for that location. This indicates that the historic instances for time 0, 1, and 2 should all contain value 3 for location 0. However, our lazy copy strategy does not guarantee that all historic instances are complete.

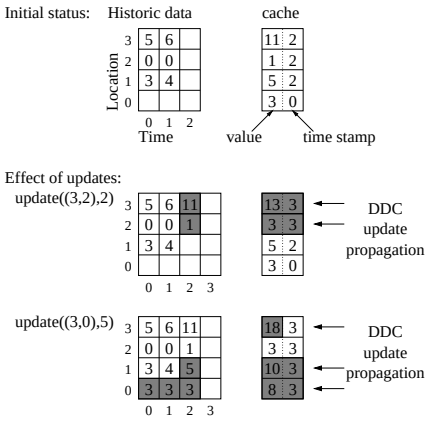


Figure 7: Update processing example

Updates are processed by first invoking the DDC update algorithm to determine which cells in `cache` are affected. Each affected cell is processed as follows. If its time stamp is equal to the time coordinate of the update, only the value is changed accordingly. If the time stamp is smaller, a new version has to be created. To do so, first the old value is copied to all historic time slices whose time coordinate is greater than or equal to the time stamp. Then value and time stamp in `cache` are updated. If an update affects a new time slice, space for this slice is reserved in the part with the historical data. Note, that the whole block of memory is only marked as "reserved", but not actually filled with any values. In Figure 7 the first update appends a new point with time 3, therefore storage is reserved in the historic array. Since location coordinate 2 is affected, the DDC pre-aggregation triggers an update to row 3 as well (indicated by arrows). In both rows the update time coordinate is greater than the time stamp in `cache`, therefore the old values are copied to time slice 2. All updated cells are shaded. The other update is processed similarly.

Unfortunately updates to cells which were not updated for a long time can trigger a large amount of write accesses to historic time slices (e.g., second update in Figure 7). We reduce these costs by using a *copy-ahead* mechanism. Recall, that a single update to a DDC-pre-computed array might affect between one and $\log_2 N$ cells. In the example the first update only affects two rows, while the second affects three rows. It is reasonable to let cheaper updates perform some extra work. More precisely, we first perform the update as described above and keep track of its costs. If the update cost is below a threshold, it additionally copies some other values with old time stamps from `cache` to the corresponding

historic cells. The copy-ahead iterates over the cells until all time stamps in `cache` are current.

We now discuss how a query is processed that selects ranges $L = (l_{\text{low}}, l_{\text{up}})$ and $T = (t_{\text{low}}, t_{\text{up}})$ in location and time dimension, respectively. Recall that this query is reduced to two queries that select range L on time slice $t_{\text{low}} - 1$ and t_{up}, respectively. Based on the DDC algorithm the required cells in the corresponding time slices are determined. For each selected cell it is first checked in `cache` what the latest time stamp of this cell is. If the time stamp is greater than t_{up} (t_{low}), the value is obtained from time slice t_{up} (t_{low}) in the historic array. Otherwise the value in `cache` is used. The correctness of this algorithm is straightforward.

3.4 The Complete Algorithms

Putting it together, we get the complete update and query algorithms for a d-dimensional data set shown in Figures 8 and 9. As shown for the framework, a d-dimensional query is reduced to two $(d-1)$-dimensional queries. Since both DDC and eCube guarantee a worst case query cost of $2^{d-1}(\log_2 N)^{d-1}$, the overall worst case query cost for $\mathcal{D}$ is $2^d(\log_2 N)^{d-1}$. Updates are performed using the DDC update algorithm on `cache` which guarantees a worst case cost of $(\log_2 N)^{d-1}$. For now we assume that the overhead for copying cells from `cache` to the historic instances (during steps (3) and (4) in Figure 8) is in the order of the query cost, i.e., does not affect the analysis. Consequently, as analyzed for the framework, we obtain a technique for maintaining and querying a d-dimensional append-only array where query and update cost are provably within a constant factor of the respective costs on a $(d-1)$-dimensional projection of this array, i.e., where the TT-dimension is removed.

Algorithm `Update`
Input: `cache`, historic time slices `hSlices`, update(X^d, Δ), pointer to cell Z^{d-1} in `cache`, time of last update t_{last}

(1) If ($x_1 > t_{\text{last}}$) reserve space for new time slice with time x_1 in memory and set $t_{\text{last}} = x_1$;
(2) `affected` = set of cells in `cache` that are affected by the update to $(x_2, x_3, \ldots, x_d)$ according to the DDC algorithm;
(3) For all cells $Y^{d-1} = (y_2, y_3, \ldots, y_d)$ in `affected`
  If (`cache`[Y^{d-1}].`timeStamp` == x_1)
    Perform update op on `cache`[Y^{d-1}];
  Else // New time slice
    Copy `cache`[Y^{d-1}].`value` to cell Y^{d-1} in all historic time slices whose time coordinate is $\geq$ `cache`[Y^{d-1}].`timeStamp`;
    Perform update op on `cache`[Y^{d-1}];
    Set `cache`[Y^{d-1}].`timeStamp` to x_1;
(4) While (current total cost of operation is low)
  If (`cache`[Z^{d-1}].`timeStamp` < x_1)
    Copy `cache`[Z^{d-1}].`value` to Z^{d-1} in `hSlices`[`cache`[Z^{d-1}].`timeStamp`];
    Increment `cache`[Z^{d-1}].`timeStamp`;
  Else set Z^{d-1} to next cell in `cache`;

Figure 8: Update algorithm

It remains to analyze the overhead added by copying cells from `cache` to the historic instances of R_{d-1} during steps (3) and (4) of the update algorithm. Our goal is to guarantee that no cell in `cache` has a time stamp that is by more than a small constant smaller than the latest update time stamp. Let the original data set have an average density of θ, $0 < \theta \leq 1$. This implies that on the average there are at least θ update operations per cell. Hence, if we perform an average of $1/\theta$ copy operations per update, then after performing all updates, all time stamps in `cache` are expected to be up to date. Arrays are only efficient if the underlying data set is not too sparse, i.e., has a minimum density $\theta_{\min}$. Hence the average copy overhead is bounded by a constant $1/\theta_{\min}$. Such average bounds guarantee an amortized constant copy cost per operation, but can not avoid higher costs for single updates if the density of different time slices varies widely. Our experiments, however, indicate that even such variances can be handled well.

Algorithm `Query`
Input: `cache`, historic time slices, query(L^d, U^d), pointer to cell Z^{d-1} in `cache`, time of last update t_{last}

(1) result(U^d) = `eCubeQuery`((($l_2, l_3, \ldots, l_d$),($u_2, u_3, \ldots, u_d$)), u_1);
(2) result(L^d) = `eCubeQuery`((($l_2, l_3, \ldots, l_d$),($u_2, u_3, \ldots, u_d$)), $l_1 - 1$);
(3) return result(U^d) - result(L^d);

Function `eCubeQuery`
Input: query range described by L^{d-1} and U^{d-1}, time t

(1) Get PS coefficients to compute range query and combine them to obtain relevant cells;
(2) Recursively compute PS value for each relevant cell;
  /* The computation is based on the eCube alg. and replaces the DDC values by the corresponding PS values in all accessed cells */
(3) Combine the results according to the PS algorithm and return total result;

Figure 9: Range query algorithm

3.5 External Memory Algorithm

If the array is too large to fit completely in main memory, the historic data (cf. Figure 5) has to be maintained on disk. We have developed an I/O-optimized version of our technique. A discussion is omitted due to space constraints. For details please refer to the full version of the paper [21].

4. USING MULTIVERSION STRUCTURES

Section 3 presented an instantiation of our framework for MOLAP data cubes. The main challenge is to find a technique for efficiently maintaining multiple instances of the $(d-1)$-dimensional data structure R_{d-1} (cf. Section 2.3). Simply copying the latest instance $R_{d-1}(t)$ when an update appends a new point X^d with $x_1 > t$ would add an overhead to this update which is linear in the size of $R_{d-1}(t)$. Also, if only a few updates affect points with time x_1, the simple technique will create a large amount of redundancy. For MOLAP arrays this is not a problem since arrays are used for fairly dense data sets. In the following we discuss solutions for sparse data.

A problem similar to maintaining instances of R_{d-1} occurs

when a data structure is made partially persistent. Data structures are typically *ephemeral* or *single-version* in the sense that making a change to the structure destroys the old version, leaving only the new one [6]. For instance, after changing the value in an array cell, the former value is lost. If any of the versions can be queried, the structure is *partially persistent* or *multiversion*. For example, a multiversion array allows a query against both the array before or after the update. Further on we will use the terms single-version and multiversion. Making a single-version structure to become multiversion requires building a data structure that can efficiently represent all versions simultaneously.

The instances $R_{d-1}(t)$ of R_{d-1} represent some of the versions of the $(d-1)$-dimensional projection (projecting the TT-dimension out) of the original data set $\mathcal{D}$. In the example in Figure 1, after the last update the instances $R_1(1)$, $R_1(3)$, and $R_1(4)$ capture the state of the one-dimensional data set (dimension is `location`) after the first, fourth, and fifth update, respectively. A multiversion structure for R_{d-1} would allow queries against each of these versions, and the remaining intermediate versions as well. In that sense the multiversion construction is actually more powerful than what we need for our framework. Stated differently, we can implement the framework described in Section 2.3 by making R_{d-1} multiversion.

Being able to take advantage of the multiversion construction, our technique can draw from a rich body of related research. Driscoll et al. [6] and Brodal [3] propose techniques for making any directed graph whose nodes have bounded size and in-degree multiversion. Their construction can be applied to virtually all index structures. Updates and queries are in the worst case by a constant factor more expensive on the multiversion structure than on the corresponding version of the single-version data structure. The storage requirement is linear in the number of updates.

These techniques are optimal for in-memory structures, but, as Becker et al. [1] point out, are not efficient for block-wise I/O operations. Their multiversion B-tree addresses the issue and offers the same worst case bounds for queries and updates as a standard B-tree, i.e., is asymptotically optimal. Similarly the Multiversion SB-tree [23] technique of Zhang et al. guarantees that aggregate range queries and updates on a two-dimensional append-only data set can be performed at the same asymptotic costs as on a general one-dimensional data set. Note that their technique constitutes an instance of our framework for data sets of time intervals. Other access structures, e.g., the R-tree and its variants can be made multiversion in a similar way [15, 22] and provide better query and update costs than the corresponding single-version tree with an additional time dimension.

Some of the multiversion constructions guarantee that the multiversion feature essentially comes "for free" [1, 3, 23], making it an ideal tool for our framework. Unfortunately, to the best of our knowledge there is no multiversion array where each cell in each version can be accessed in constant time and where updates to the current version incur constant worst case cost as well. This motivated our new approach (cf. Section 3).

5. EXPERIMENTS

We performed an extensive experimental evaluation of our MOLAP aggregation technique. In Section 3.4 query and update cost were already shown to be at most twice as big as or equivalent to the corresponding cost on a $(d-1)$-dimensional time slice of the d-dimensional array. This analysis assumed that the cost of making copies of instances of R_{d-1} could be amortized over multiple update operations. The main goal was to validate this assumption.

Here we discuss experiments for three selected data sets (see Table 3). Experiments with other data, e.g., synthetic uniform and gaussian data sets, and real data sets, lead to the same conclusions. The selected data sets are sparse, and especially `weather4` and `weather6` would be expected to cause difficulties for array based techniques due to their dimensionality. Recall also, that the cost bounds of the pre-aggregation techniques (cf. Section 3.1) grow exponentially with increasing dimensionality. Note, that `weather4` and `weather6` were obtained from the same data set [11] and were selected such that they have approximately the same total number of cells (note the granularities of latitude and longitude in the sets). The reason for the different number of non-empty cells is that both data sets are actually projections of a 20-dimensional original data set.

The query cost was measured for different sets of randomly generated range queries. Here we discuss results for two query sets `skew` and `uni`. If not stated otherwise, `uni` was used. The multidimensional range queries of `uni` are generated as follows. First for each dimension one of the query predicates $\min \leq x \leq A$ (prefix range), $A \leq x \leq B$ (general range), $x = A$ (point query), and $\min \leq x \leq \max$ (complete domain) is selected with probability 0.1, 0.7, 0.1, and 0.1 respectively. Then the values of A and B are selected randomly from the corresponding domain. This selection favors general ranges and generates a wide spectrum of different selectivities. The queries of `skew` are constructed in the same way, however, 80% of them concentrate in an area that is 0.5^d times the size of the complete data space.

In a first set of experiments we analyzed the efficiency of the eCube query algorithm. Recall, that eCube combines DDC and PS query techniques to be able to deal with arrays that contain DDC and PS aggregated values. A major feature of eCube is that it gradually changes a DDC pre-aggregated array with polylogarithmic query costs to PS which has constant bound query costs (2^d for a d-dimensional data set). Figures 10 and 11 show the results for the time slices of `weather4`. Results for other data sets were similar (cf. [21]). The eCube algorithm has a clear tendency of decreasing query costs the more queries are processed. On the skewed query set a faster convergence is observed since queries are skewed towards a region of the data set, i.e., the benefit of changing to PS pays off earlier. As expected, the cost of DDC and PS varies around the same average over time since these techniques do not alter cell values. The values in the graph are the rolling averages over groups of 50 queries to improve the presentation. Note, that all algorithms access a very small percentage of the array, even when comparing to the number of non-empty cells. An interesting observation is that eCube in the beginning has higher query costs than DDC even though both techniques guarantee the same worst case costs. The reason is that eCube always reduces a general range query to two prefix queries, while DDC's direct approach avoids accesses to cells that are added by the first, and then subtracted by the second prefix query (cf. Section 3.1). This makes a small difference in each dimension which is amplified with increasing dimensionality.

Name	Description
`weather4`	`COUNT` data cube for cloud data [11]; 4 dimensions (latitude and longitude at granularity of degrees, total cloud cover, time of measurement); 143,648,037 cells; 1,048,679 non-empty cells (density 0.0073)
`weather6`	`SUM` data cube for cloud data [11]; 6 dimensions (latitude and longitude at granularity of 10 degrees, total cloud cover, lower cloud amount, middle cloud amount, time of measurement); 139,826,700 cells; 549,010 non-empty cells (density 0.0039)
`gauss3`	`SUM` data cube for a data set with 60 dense clusters generated with a gaussian distribution; 3 dimensions (each with a domain of size 271); 19,902,511 cells; 950,633 non-empty cells (density 0.048)

Table 3: Data sets

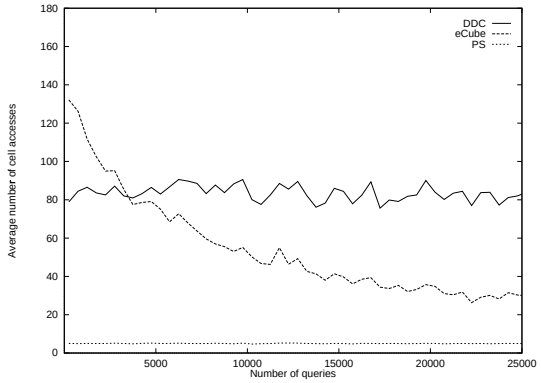


Figure 10: Query cost versus number of executed queries (`weather4`, `uni` queries)

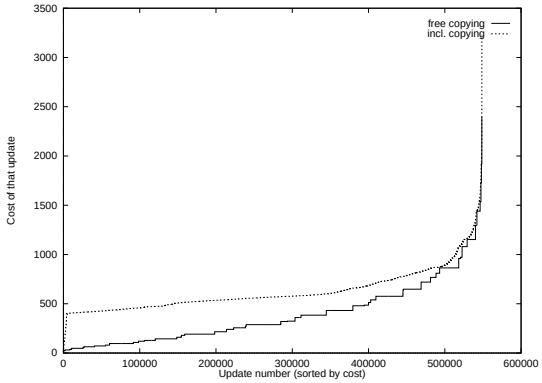


Figure 12: Update cost quantiles with and without copy cost (`weather6`)

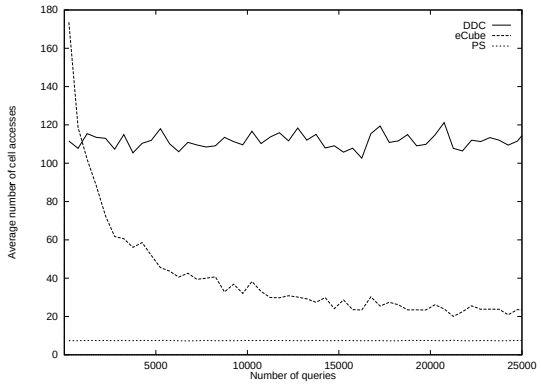


Figure 11: Query cost versus number of executed queries (`weather4`, `skew` queries)

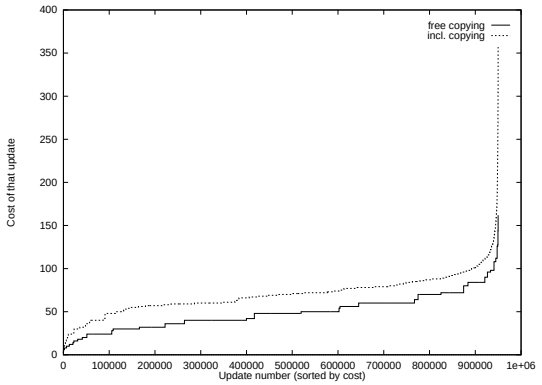


Figure 13: Update cost quantiles with and without copy cost (`gauss3`)

The analysis of the update costs in Section 3.4 was based on the assumption that it is possible to amortize the copy costs over multiple operations. Figures 12 and 13 compare the cost of updates for an ideal case where copies are available instantly and "for free" and for our real algorithm as discussed in Section 3.4. For each update the cost was recorded. The figures show these costs of the single operations in sorted order. The area between the two curves is equivalent to the total copy cost. As can be seen in the graphs, most copies were performed by the cheapest operations, while updates that were already expensive did little extra work. As a consequence, for instance more than 90% of the updates on `weather6` cost less than 900 with and without including copy costs.

The results in Table 4 complement Figures 12 and 13. After each update we measured how many historic instances of R_{d-1} were not completely copied yet. For the *in-memory* algorithm (cf. Figure 8) the number of incompletely copied instances is an upper bound on the number of forced copies during an update (step (3)). Hence, as the results indicate, the additional copy cost in step (3) is bounded by a small constant as predicted in Section 3.4. Note, that the extremal values in Table 4 occur in the beginning of the algorithm. After a short time the most frequent value is reached and maintained. The maximum value 5 for `gauss3` is caused by the variance in the number of updates per time slice (because of the clusters). Nevertheless, most of the time only one historic time slice is incomplete. The *disk based* technique (cf. Section 3.5) copies cells page-wise to disk. The page size was set to 8K and at most one page access per update

was allowed. This was more than enough to copy cells of historic instances of R_{d-1} without ever having more than one historic instance which was not completely copied. The reason is that a page fits 2048 cells (since only the measure values of 4 bytes size each are stored), i.e., a single page write copies 2048 cells.

Data set	Min	Max	Most frequent
`weather4` (in-memory)	0	2	2
`weather4` (disk)	0	1	1
`weather6` (in-memory)	0	2	2
`weather6` (disk)	0	1	1
`gauss3` (in-memory)	0	5	1
`gauss3` (disk)	0	1	1

Table 4: Number of incomplete historic instances of R_{d-1} after each update

Arrays are very efficient for maintaining dense in-memory data sets since each cell's value can be accessed in constant time. However, typically they are not the structure of choice for supporting aggregation on sparse data sets which reside on disk. Our experiments show, that even for sparse data the number of page accesses is much lower than for a bulk-loaded R*-tree. The bulk loaded tree is based on the algorithm presented in [2]. We used a page size of 8K which favors the R*-tree (the smaller the pages, the better for the array which is designed for single cell accesses) and only counted the number of *leaf accesses* for the R*-tree, assuming that all internal nodes could fit in main memory. Apart from that no further caching was used for both techniques. We measured the number of page accesses for the R*-tree and an array that was pre-aggregated using the standard DDC technique (cf. Section 3.1). The cells within a time slice were stored in simple row-major order. We performed 10000 `uni` range queries on `weather6` and report the cost of each single query in Figure 14 in the order of increasing cost. The figure indicates that the index has considerably higher query costs, the average costs being 275.65 and 59.17 for the R*-tree and the array, respectively. The disadvantage of the DDC technique is that its pre-aggregation leads to a storage increase by a factor up to 20 compared to the index. Increasing dimensionality and sparseness at some point will favor the R*-tree index. For more results see [21].

In conclusion the experiments show that our approach using eCube efficiently integrates append-only data. Once data is incorporated the query cost decreases rapidly, approaching constant overhead. Furthermore we showed that this approach outperforms traditional multidimensional index structures, even when they are bulk loaded and hence optimized for queries.

6. RELATED WORK

Note, that our notion of append-only data is closely related to temporal databases, where the *transaction time* of a fact "is the time when the fact is current in the database and may be retrieved" [7]. Transaction times are consistent with the serialization order of the transactions. It is impossible to change the past and tuples are only logically deleted. Hence, a temporal data set that has a transaction time attribute is an append-only data set. The other important

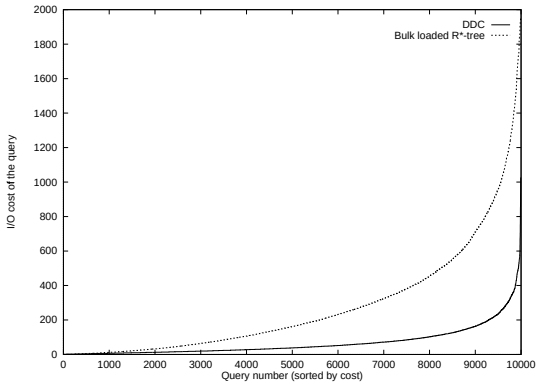


Figure 14: Comparison of I/O cost for DDC array and bulk-loaded R*-tree (`weather6`)

concept in temporal databases is valid time. The *valid time* of a fact "is the time when the fact is true in the modeled reality" [7]. Bitemporal databases support both, valid and transaction time. Our framework is applicable to any data set with transaction time, i.e., also bitemporal databases. Kumar et al. [15] also suggest the multiversion methodology for constructing efficient bitemporal access methods. However, their construction is not optimized for aggregation. A range selected in the TT-dimension would require a query on each time slice whose time coordinate is within that range (instead of accessing at most two time slices as guaranteed by our framework).

The temporal aggregation algorithm proposed by Zhang et al. [23] represents an instance of our framework for two-dimensional append-only data sets that consist of time intervals. Previous work on temporal aggregation [9, 14, 17] mainly focussed on queries that aggregate over the whole range in all non-temporal dimensions.

O'Neill and Burton [18] present a technique for functional arrays where multiple versions of array cell values are maintained. Like our `cache` (cf. Section 3.3) data structure, their scheme is motivated by the fat node method proposed in [6]. Functional arrays are more general than a multiversion array since they also allow updates to historic time slices and hence require more complex algortihms. None of those techniques can guarantee constant single cell accesses and updates. Access to secondary memory is not considered.

The data cube operator [10] has an interesting relation to our MOLAP aggregation technique. Both the DDC pre-aggregation technique as presented here and the PS technique generate arrays whose surface cells correspond to the tuples of the group-bys (cuboids) of the data cube. More precisely, the cells contain prefix-sums over the data cube tuples. Hence, our MOLAP technique is a new way of computing and incrementally maintaining a data structure that supports efficient range queries on the data cube for append-only data sets.

7. CONCLUSIONS

We presented a new framework for the efficient maintenance and aggregation over append-only data sets which play an important role in real applications like data warehouses. Our technique makes the query cost independent of the size of the selected range in the time dimension. This is of high importance since the data set grows along this

dimension. Both the framework in general and our MOLAP instantiation in particular provide efficient means of incrementally maintaining data cubes for append-only data streams (cf. weather data sets in Section 5). Our framework essentially allows adding a TT-dimension and including it in aggregation queries for free.

As a by-product our technique simplifies the task of retiring old data, called *data aging*. Data warehouses distinguish between current detail data, old detail data and summarized data at different levels [13]. While current detail data and summarized data are frequently used, access to older detail data is rare. To deal with the accumulation of huge amounts of data, an aging process moves old detail data to (slower) mass storage. Our technique automatically clusters data based on the time coordinate. This greatly simplifies data aging. In addition to that, aggregates of retired detail data can be retained without additional computation costs at the time of the retirement.

By using existing multiversion data structures our approach can be applied to a wide range of applications. However, as discussed in Section 4, these structures are more powerful than necessary for our framework. An interesting direction of future research is to examine how much of this unnecessary functionality could be traded for better performance within our framework. We also intend to develop new data structures that support disk-based aggregation on sparse data sets.

8. REFERENCES

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB Journal*, 5(4):264–275, 1996.

[2] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 216–230, 1998.

[3] G. S. Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[4] C.-Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 675–686, 1999. Extended version published as Technical Report, Univ. of Wisconsin, 1999.

[5] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences (JCSS)*, 38(1):86–124, 1989.

[7] C. S. Jensen 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] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 237–253, 2000.

[9] J. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel algorithms for computing temporal aggregates. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 418–427, 1999.

[10] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997.

[11] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991, 1996. Data available at http://cdiac.esd.ornl.gov/ftp/ndp026b.

[12] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 73–88, 1997.

[13] W. H. Inmon. What is a data warehouse? White Paper, 2000. Available at http://www.billinmon.com/cif/edw/edw_content.html.

[14] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 222–231, 1995.

[15] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):1–20, 1998.

[16] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 401–412, 2001.

[17] B. Moon, I. Lopez, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 145–154, 2000.

[18] M. E. O'Neil and W. F. Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–514, 1997.

[19] M. Riedewald, D. Agrawal, and A. El Abbadi. pCube: Update-efficient online aggregation with progressive feedback and error bounds. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 95–108, 2000.

[20] M. Riedewald, D. Agrawal, and A. El Abbadi. Flexible data cubes for online aggregation. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 159–173, 2001.

[21] M. Riedewald, D. Agrawal, and A. El Abbadi. Efficient integration and aggregation of historical information. Technical Report 2002-07, University of California, Santa Barbara, 2002.

[22] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 431–440, 2001.

[23] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 237–245, 2001.

[24] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, 2002. To appear.