Time Travel in a Scientific Array Database

Emad Soroush and Magdalena Balazinska

Department of Computer Science & Engineering
University of Washington, Seattle, USA
{soroush,magda}@cs.washington.edu

***Abstract*—In this paper, we present TimeArr, a new storage manager for an array database. TimeArr supports the creation of a sequence of versions of each stored array and their exploration through two types of time travel operations: selection of a specific version of a (sub)-array and a more general extraction of a (sub)-array history, in the form of a series of (sub)-array versions. TimeArr contributes a combination of array-specific storage techniques to efficiently support these operations. To speed-up array exploration, TimeArr further introduces two additional techniques. The first is the notion of approximate time travel with two types of operations: approximate version selection and approximate history. For these operations, users can tune the degree of approximation tolerable and thus trade-off accuracy and performance in a principled manner. The second is to lazily create short connections, called *skip links*, between the same (sub)-arrays at different versions with similar data patterns to speed up the selection of a specific version. We implement TimeArr within the SciDB array processing engine and demonstrate its performance through experiments on two real datasets from the astronomy and earth sciences domains.**

I. Introduction

In many fields of science, multidimensional arrays rather than flat tables are standard data types because data values are associated with coordinates in space and time. For example, images in astronomy are 2D arrays of pixel intensities. Climate and ocean models use arrays or meshes to describe 3D regions of the atmosphere and oceans. They simulate the behavior of these regions over time by numerically solving the governing equations. Cosmology simulations model the behavior of clusters of 3D particles to analyze the origin and evolution of the universe.

At the same time, datasets in science are growing in size. The next generation of telescopic sky surveys such as the Large Synoptic Survey Telescope (LSST) [1] will generate 10s to 100s of petabytes a year of imagery and derived data. The Earth Microbiome Project [2] expects to produce 2.4 petabases in their metagenomics effort.

As a result, scientists need powerful tools to help them manage these massive arrays. Because simulating arrays on top of relations can be inefficient [3], many specialized array-processing systems have emerged [4], [5], [6], [7].

An important requirement that scientists have for these systems is the ability to create, archive, and explore different *versions* of their arrays [3]. Hence, a no-overwrite storage manager with efficient support for querying old versions of an array is a critical component of an array database management system (DBMS).

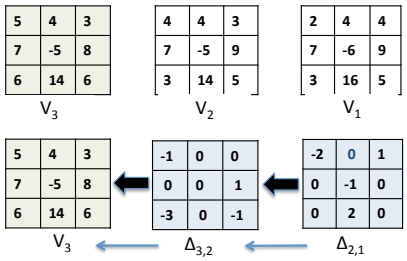


Fig. 1: Illustration of a chain of backward delta versions for a 3x3 array. The most recent version V_3 is materialized. Earlier versions are stored in the form of arrays of cell-value differences.

Such a system must support different types of queries over the versioned array: It must support standard queries that retrieve specific array versions, queries that retrieve subarrays at specific versions, and queries that track the history in the form of a series of subarrays across multiple versions. At the same time, we argue that early array exploration can benefit from faster but *approximate* queries that can quickly identify which versions are relevant to a user and the approximate content of these versions. Finally, all these operations must be performed as efficiently as possible to enable fast data exploration and analysis.

In this paper, we present TimeArr, a new storage manager for array DBMSs that provides a no-overwrite, versioned array storage model together with both precise and approximate time-travel queries over these versioned arrays. Our TimeArr storage manager makes the following contributions:

(1) A backward delta versioning system specialized for arrays. At the heart of the TimeArr storage manager, is a new storage model for efficiently representing and querying a versioned array. First, because scientific datasets can grow to be large, the storage model compresses the data using a *backward delta* encoding method: the most recent version of the array is fully materialized and earlier versions only contain differences in cell values between consecutive versions as illustrated in Figure 1. $\Delta_{i,i-1}$ represents differences in cell values between version V_i and V_{i-1} of an array. The backward delta technique is known to be an efficient compression method. To illustrate the efficiency of this method for scientific arrays, we store 61 versions of the Global Forecast System (GFS) dataset [8] in TimeArr using four methods. The naïve materialization of all versions takes 65.6MB of space on disk. Storing only the values of cells that change between consecutive versions reduces disk-space utilization to 14MB. Storing differences in cell values between each version V_i and the original version

V_z achieves almost no compression compared to storing only materialized versions and takes about 62.7MB. Finally, the backward delta method stores all versions using only 3.5MB, a 19X improvement over the full materialization. Of course, this compression comes at the cost of slower version retrieval. Hence, an important question is how to achieve fast array query processing with this method. Query processing times are also the main reason for always materializing the most recent version of the array, which should be most frequently accessed by applications.

Unlike most other applications of the backward deltas method (*e.g.*, in backup storage [9] or temporal databases [10]), our storage layout is specialized for arrays. The specialization enables TimeArr to achieve both high compression ratios and high query performance. The approach uses three key ideas. First, it applies the notion of array tiling [6], [11], [12], [13], [14] to efficiently limit the changes that must be processed when retrieving old versions of a subarray. This approach significantly speeds up query processing. Second, it uses a variety of compressed bitmasks [6] to encode the regions of an array that change from one version to the next and to identify which subset of changes need to be processed to satisfy a user query. This approach both enables better data compression and speeds-up query processing. Third, our storage model uses variable-length delta encoding across tiles, which helps adapt the compression-level to different magnitudes of changes in different regions of an array and yields better compression ratios for the array data. In addition to these three basic methods, to further speed-up query processing over commonly accessed parts of an array, TimeArr lazily adds connections, called *skip links*, between certain non-consecutive versions of an array. TimeArr's *skip links* are similar to regular backward delta versions except that they contain differences in cell values between two non-consecutive versions. TimeArr utilizes skip links similarly to a skip list data structure [15] with the important difference that TimeArr creates links based on version *content* and not version numbers. Additionally, TimeArr creates skip links lazily during version retrieval to reduce the overhead of maintaining this data structure. Finally, it maintains skip links at the granularity of tiles to increase their efficiency. As a result, regions of the array that are fetched more often create more skip links which reduces their version retrieval time. We present TimeArr's detailed storage model in Section III.

(2) Approximate and customizable array-exploration queries. It is well-known that, when first exploring data, users need a quick query turn-around time and are willing to tolerate some inaccuracy to achieve faster time-to-result [16], [17], [18]. To speed-up the exploration of a versioned array, we leverage this observation and introduce the idea of querying *approximate* versions of an array. In our approach, the user specifies both the degree of approximation tolerable and how that approximation should be computed. Hence, TimeArr's approximate exploration is *highly customizable* and *carefully controlled* by the user. The system efficiently answers approximate queries over a versioned array by aggressively leveraging

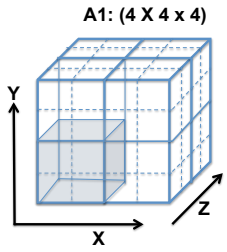


Fig. 2: The 4x4x4 array A1 is divided into eight 2x2x2 chunks.

tiling and skip links and also by maintaining short summary statistics that capture the overall changes between different subarrays at different versions. We present the details of approximate version querying and customization in Section IV.

(3) Prototype implementation and evaluation with real datasets. We implement the above techniques as a C++ prototype storage manager called TimeArr on a branch of the SciDB array processing engine [6]. We evaluate TimeArr on a real dense array containing 61 snapshots from the global forecast system (GFS) model [8] and a real sparse array containing 9 snapshots from an astronomy universe simulation [19]. For precise queries, without using skip links, TimeArr outperforms the current SciDB version-storage technique [6], [20] (which is also based on backward deltas) by a factor of 1.6X to 6.6X in terms of query processing times and up to 40% in terms of version creation time. Skip links further improve query performance by 75%. Furthermore, when a user retrieves only a small fraction of an array, our approach based on a combination of bitmasks and virtual tiling can cut query times by an order of magnitude. For approximate queries, we show that query times are halved when a user is willing to see data off by at most one array version.

The goal of TimeArr is to efficiently support queries for array regions and versions. We do not study additional indexing techniques over array contents.

II. TimeArr Overview

TimeArr is a new storage manager for array database systems. While TimeArr could be integrated with various array systems [5], [6], our design and implementation are based on the SciDB engine [6]. In this section, we present an overview of TimeArr's approach and also TimeArr's core API.

In most array database engines including SciDB, each array is partitioned into chunks, which are small subarrays as illustrated in Figure 2. Array chunking is a well-known method for alleviating dimension dependency [21]. Each chunk maps onto a unit of disk IO (either a disk block or larger). TimeArr assumes a chunked array layout. Furthermore, we assume that chunks are regular. That is, each chunk covers the same space in terms of array coordinates. This layout has been shown to deliver high performance across a wide range of array operations for both dense and sparse arrays [14]. In SciDB, array chunks are further stored using a column-based representation [6]. TimeArr builds on this column-based, chunked storage layout.

Array Updates
`Create(ArrayType` Q`, ArrayName` A`)`
`Append(ArrayName` A`, ArrayContent` C_i`)`
Precise Queries
`Select(ArrayName` A`, Predicate` p`, VersionNb` j`, VersionNb` k`)`
Approximate Queries
`Select(ArrayName` A`, Predicate` p`, VersionNb` j`, VersionNb` k`, ErrorBound` B_1`, ErrorBound` B_2`, Granularity` g`, StatisticID` s`)`

TABLE I: TimeArr Versioned Array API.

TimeArr supports the operations shown in Table I. The `Create` operation creates an initial, empty array of type Q and named A. The array type includes the specification of the array dimensions, the type of each array cell, and how the array should be both chunked and tiled. This operation only creates array metadata in the SciDB catalog.

The `Append` operation appends a new version to array A. The payload of the append operation, C_i, is a new snapshot of the array content at the new version i. Version numbers are incremented automatically.

When an initial array version is created, its data is broken up into chunks as per the chunking specification in the `ArrayType`. Each chunk is stored in a separate file on disk. For example, the array from Figure 2 is stored in eight separate files, one per chunk. Subsequent calls to `Append` add new versions to the array. The new version of each chunk is added to the corresponding file where the earlier version of that chunk is stored. We refer to a file that contains a materialized chunk together with its series of appended versions as a *segment*.

To maintain high performance in the face of a growing number of versions, TimeArr is configured with a maximum segment size F. If a segment grows beyond threshold F for some chunk, a new segment is created for that chunk. Each segment (or file) contains one materialized version of a chunk, which is the most recent version stored in that segment. All prior versions in the same segment are compressed using the backward-delta-based approach described in Section III. Such periodic materialization is a well-known technique adopted in many systems including BigTable [22]. The selection of the threshold value F depends on various parameters such as chunk size and version content. We do not address the problem of optimizing the value of F in this paper.

The `Select` operation returns the content of a subarray of A that satisfies predicate p at versions $v \in [V_j, V_k]$. We refer to this operation as *array history selection*. To retrieve data for a single version, the last argument can be omitted. To retrieve the data for the entire array, the predicate p can be omitted. p is a predicate over array dimensions. For example, in the array from Figure 2, we could select the first chunk with predicate $x \in [1,2] \wedge y \in [1,2] \wedge z \in [1,2]$. We further present TimeArr's storage model and history selection query implementation in Section III.

TimeArr also supports an approximate variant of array history selection to speed-up early array exploration. As shown in Table I, this variant takes four extra arguments as input. The first one, B_1, is an error bound: if a user requests a single array version, V_j, B_1 serves to specify the maximum tolerable loss in accuracy. The selection of a specific array version thus returns the subarray of A at version V_j that satisfies p. The content returned, $c'_j(p)$, satisfies the error condition: $\texttt{Difference}(c'_j(p),\ c_j(p)) < B_1$, where $c_j(p)$ is the precise version of the corresponding subarray. The computation of the `Difference` function is configurable as we show in Section IV. In fact, a user can specify several methods for computing this difference and use different methods in different queries. The `StatisticID` argument to the function specifies which of these methods to use. If not specified, TimeArr uses the *default* `StatisticID`. TimeArr computes version differences at two granularities of tiles or chunks. The user specifies the granularity with the `Granularity` parameter. We further discuss the semantics and computation of these differences in Section IV.

When multiple versions are requested, an extra parameter B_2 must also be specified. The `Select` operation then returns the most recent requested version, V_j, within error bound B_1 as above. It also returns a sequence of versions V such that $\forall\ V_u \in V,\ V_u \in (V_j, V_k] \ \wedge\ \texttt{Difference}(c'_u(p), c_u(p)) < B_1$. Additionally, for each pair, (V_u, V_w) of consecutive returned versions (*i.e.*, no version in between V_u and V_w is returned), we have $\texttt{Difference}(c_u(p), c_w(p)) > B_2$. This operation thus returns the first selected version using the same method as above. It then returns subsequent versions such that each new version's content remains within distance B_1 of the corresponding precise version. Additionally, the query skips over similar versions, returning only the next version that differs by at least B_2. The granularity (tile or chunk) is the same as for B_1. We further discuss this approximate history extraction in Section IV.

III. Version Storage and Retrieval

In this section, we present TimeArr's approach to storing and retrieving array version data.

A. Version Storage

As indicated earlier, TimeArr stores array versions using a backward delta approach: When a new version of a given chunk is appended, TimeArr iterates over both the new version, call it V_j, and the most recent previous version, call it V_{j-1}, of the chunk. It subtracts the cell values in the new chunk from the corresponding cell values in the older chunk. These differences in cell values are called *delta values*. More formally: $d_{j(j-1)k} \leftarrow Subtract(c_{(j-1)k}, c_{jk})$ where d is the delta value and c_{jk} is the k'th cell in the array at version j, assuming that cells are traversed in some order such as the row-major order. The group of delta values for a chunk forms a *delta chunk*. We call the array that wraps all delta chunks the *delta array*. Figure 1 illustrates a materialized array version and two delta arrays.

While the basic idea of storing array versions using backward deltas is not new [20], the details of the version data structures that TimeArr uses are different from prior work. In particular, TimeArr's version storage layout uses four key ideas: (1) it partitions chunks into tiles to limit the amount of work when rebuilding an old version of a subset of an array or when answering an approximate query; (2) it uses bitmasks to quickly identify the tiles or cells that changed between two versions; (3) it uses variable-length delta-encoding to capture changes with as few bytes as possible; it also uses run-length encoding (RLE) to compress its bitmasks; (4) it lazily creates skip links to boost the `Select` query performance over time. We now present these four key techniques.

Figure 3 shows the internal representation that TimeArr uses to store one segment on disk. Each segment contains one materialized version of a chunk and zero or more delta chunks. The materialized version in the segment could be stored using either a sparse or dense representation, with or without compression, etc. [6], [11], [12], [13], [14], [20], [23]. In this paper, we treat the most recent version as a black box.

TimeArr represents each delta chunk with a structure that we call `VersionDelta`. To speed-up range-selection and approximate queries, TimeArr divides a delta chunk into a series of *virtual delta tiles*. Each tile is a subarray within the delta chunk. TimeArr represents virtual delta tiles with a structure that we call `TileDelta`. In the rest of the paper, $\Delta_{i,i-1}$ represents the delta values between version V_i and V_{i-1} of either an array, a chunk, or a tile depending on context.

A `VersionDelta` contains a header that summarizes the changes in the version and a payload that holds the actual changed values. The `VersionDelta` header contains a bitmask with one bit per tile (`VDBitmask` in the figure). `VDBitmask` identifies the tiles that have been modified in the new version of the array. Such tiles have their bit set to `true` in the `VDBitmask`. This approach has successfully been applied in the past to compressing array contents [6]. We apply it here for compactly storing changes between array versions.

Tiles that contain changes are stored in a set of `TileDelta` data structures. A `TileDelta` contains the details of changes in one tile. Because `TileDeltas` have variable sizes, TimeArr uses a standard slot-based approach to locate them on disk: for each tile that includes changes, a slot points to the location of the corresponding `TileDelta` on disk (`TileSlotsMap` in the figure). Prior work studied the tuning of chunk/tile shape, size, and layout on disk for a given workload and for regular chunking [24], [25]. In TimeArr, the virtual tile size determines the finest granularity with which the system can do history and approximation queries. Hence, smaller tiles enable finer-grained operations. On the other hand, larger tiles decrease the metadata overhead and preserve the locality of the data (logically close delta values in the array are physically stored together). However, we do not address the problem of tuning the size of virtual tiles in this paper.

The details of a `TileDelta` structure are illustrated in Figure 4. Similar to the `VersionDelta` structure, each `TileDelta` contains a bitmask with one bit per cell (`TileBitMask` in the figure). The `TileBitMask` is a bit vector that indicates which cells in the tile contain any changes. A `TileDelta` also contains a payload that holds the actual delta values. A conceptual view of a TileDelta is shown in Figure 5. The first delta value in the list corresponds to the first 1 in the bitmask, the second delta value corresponds to the second 1, and so on. Because we use regular tiling, where each tile covers the same number of cells in each direction as other tiles, mapping from the bitmask bits to the cell coordinates happens efficiently in near constant time.

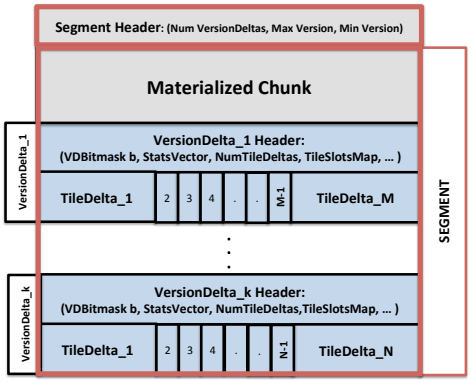


Fig. 3: Representation of a single array chunk with multiple versions.

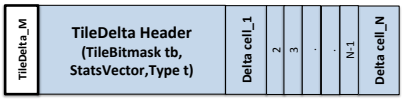


Fig. 4: TileDelta Layout

Following the `TileDelta` header, we store the actual cell updates in the form of backward deltas. Depending on the magnitude of the changes, we can use a different number of bytes to store the delta values. TimeArr chooses the number of bytes to use to store delta values at the granularity of tiles. The `TileDelta` header includes some information about which encoding is used for delta values (Type *t* in the figure).

To save space, TimeArr uses run-length encoding (RLE) to compact all bitmasks. For example, bitmask `1100111000` is RLE encoded as <1,2> <0,2> <1,3> <0,3>. Values of 1 in the bitmask correspond to cells that were updated.

In addition to bitmasks, `VersionDelta` and `TileDelta` headers also include summary vectors, called `StatsVectors`. We describe the `StatsVectors` in Section IV, when we discuss customization and approximation.

B. Skip Links

Initially, the data in a segment corresponds to a fully materialized version of a chunk and a series of consecutive delta chunks. If V_r is the materialized version in the segment then any older version, V_k, of the chunk in the segment is rebuilt as follows: $V_k = V_r + \sum_{i=k+1}^{r}(\Delta_{i,i-1})$ where the "+" operator applies all delta values in one version of the chunk by invoking an `Add` function for each cell:

$c_{(j-1)k} \leftarrow Add(c_{jk}, d_{j(j-1)k})$

where c are cell values, d is a delta value, k represents the k'th cell, and j and $j-1$ represent two consecutive versions.

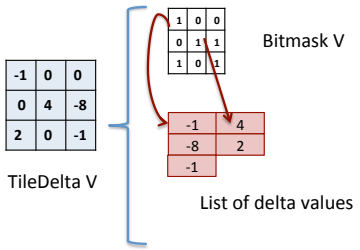


Fig. 5: Internal structure of a TileDelta `V` in TimeArr. The bitmask is represented as a 2D array only for illustration purposes.

The version retrieval time thus grows linearly with the number of versions in a segment. One can use skip lists [15] to maintain the retrieval time for any version of a chunk below $\log(|V|)$ where $|V|$ is the number of versions in a segment. That is, a segment should contain the VersionDelta for consecutive versions but it should also contain additional VersionDeltas for each pair of versions 2^i versions apart. Extra VersionDeltas, however, would significantly increase storage costs. Additionally, as we discussed in Section I, delta chunks for non-consecutive versions that are far apart do not provide much compression compared to materializing the actual array version. Finally, skip lists would significantly increase the time to append a new version due to the creation of multiple extra delta chunks.

To avoid these limitations yet benefit from "shortcut links", TimeArr uses what we call *skip links*, inspired by the skip list technique, to cut the version retrieval costs by *skipping* over multiple versions in one step. To maximize the benefits of these links, TimeArr defines them at the granularity of tiles. The fundamental differences between a skip list and TimeArr's skip links are that (1) skip links *replace* some of the consecutive delta tiles, ($\Delta_{i+1,i}$), with non-consecutive ones, $\Delta_{j,i\ j>i+1}$, and (2) skip links are established only between *similar* versions; that is, only when $\Delta_{j,i}$ is backward delta encoded more compactly than $\Delta_{i+1,i}$:

$$\texttt{sizeof}(\Delta_{j,i}) < \alpha \times \texttt{sizeof}(\Delta_{i+1,i}) \quad (1)$$

where `sizeof` returns the size of an object in bytes and $\alpha \in [0, 1]$ is a tunable parameter that ensures skip links are created between similar tiles rather than between arbitrary ones. We use $\alpha = 0.9$ in our experiments, which we find to suffice to filter out spurious skip links. An abstract example of skip links is shown in Figure 6.

To decide which tile versions to consider for replacement, TimeArr could enumerate all possible $\Delta_{j,i}$ combinations and verify the condition in Equation 1. This approach, however, would be computationally expensive because of the large number of version combinations. Instead, we propose to consider only the *linear* sequence of links. That is, given a most recent version V_r, TimeArr only considers adding skip links of the form $\Delta_{r,i\ \forall i<r}$. This approach is significantly less expensive computationally because it considers fewer options but also because it can compute these options incrementally. Indeed, TimeArr reuses the computation spent on a previous candidate skip link $\Delta_{r,i}$ to recursively construct the new candidate skip link $\Delta_{r,i-1}$. At the same time, we hypothesise (and experimentally demonstrate in Section V) that this approach retains the most useful links, since TimeArr always starts from V_r when fetching older versions.

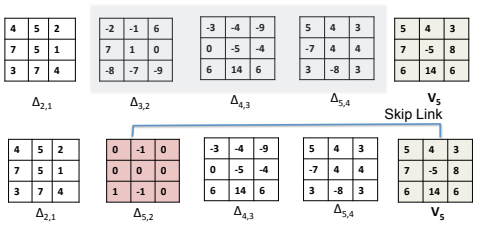


Fig. 6: $\Delta_{5,2}$ is a skip link from V_5 to V_2. $sizeof(\Delta_{5,2}) < \alpha \times sizeof(\Delta_{3,2})$. So $\Delta_{3,2}$ is replaced with $\Delta_{5,2}$ in the chain of backward deltas. Note that $V_2 = V_5 + \Delta_{5,2}$ and $V_1 = V_5 + \Delta_{5,2} + \Delta_{2,1}$.

An important design decision for TimeArr is *when* to create skip links. One approach is to exhaustively consider all linear skip links every time a new version is appended to a chunk. A less expensive variant is to compute skip links only every T new versions appended, where $T > 1$. We study the overhead and gain of different values of T in Section V for version insertion and retrieval.

A third approach is to identify skip links lazily when executing selection queries that retrieve old versions of subarrays. The approach works as follows: Consider a segment with materialized version V_r and the goal is to retrieve version V_i. To test potential linear skip links, TimeArr reconstructs V_i as $V_i = V_r + \Delta_{r,i}$ and it computes $\Delta_{r,i}$ incrementally by computing each intermediate linear skip link, $\Delta_{r,k\ \forall k\ i\leq k<r}$, where V_r is the materialized version in the segment. This approach thus significantly reduces the overhead of finding skip links at the expense of not being able to use this optimization the first time that an old version is retrieved. To further limit overheads, while retrieving old versions, TimeArr keeps track of deltas $\Delta_{r,i}$ that have already been explored as potential skip links. For example, if TimeArr issues two consecutive selection queries to retrieve V_i, only the first one involves the exploration of possible skip links.

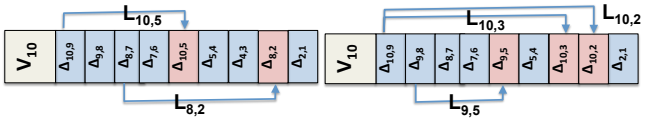
Algorithm 1 describes the tile-based skip link creation procedure. In the algorithm, after $\Delta_{i+1,i}$ is replaced with skip link $\Delta_{j,i}$ for tile t (Line 9), TimeArr puts a *lock* on all the deltas $\Delta_{k+1,k}\ i < k < j$ at tile t. Delta versions that are locked are not eligible to be replaced with any other skip links (which is one reason why spurious links should be avoided by tuning the α parameter). Locks are at the granularity of tiles and are not revertible. This constraint is reflected in Algorithm 1 at Line 8 and line 10. The reason TimeArr locks the deltas is to avoid *overlapping skip links* as illustrated in Figure 7(a). If skip links overlap, TimeArr can reach a dead-end if it does not apply the correct combination of skip links during version retrieval. Locking certain tiles prevents this complication and simplifies the version retrieval algorithm.

C. Query Processing

To support a selection query that retrieves a specific version of an array chunk, TimeArr first selects the set of files –with one file per array chunk– that contain the desired version.

Algorithm 1 Skip Links Creation Procedure for One Tile

Input: Materialized Version V_r, Delta versions $\Delta_{i+1,i}$, Target Version Index k.
Output: V_k.
$i \leftarrow r-1$
$\Delta_{r,x} \leftarrow \phi$
while $i \geq k$ **do**
  $\Delta_{r,x} \leftarrow \Delta_{r,x} + \Delta_{i+1,i}$
  if $\Delta_{r,x} \leq (\alpha \times \Delta_{i+1,i})$ **then**
    if $\Delta_{i+1,i}$ is not *locked* **then**
      Replace $\Delta_{i+1,i}$ with $\Delta_{r,x}$
      Lock all the $\Delta_{n+1,n}$ $i < n < r$
    end if
  end if
  $i = i - 1$
end while
$V_k \leftarrow V_r + \Delta_{r,x}$



(a) Invalid State: V_1 and V_2 are accessible only if TimeArr applies $L_{8,2}$ and not $L_{10,5}$.

(b) Valid State: All versions are accessible through any combination of links.

Fig. 7: Valid *v.s.* Invalid states of skip links. TimeArr must avoid overlapping skip links. The *lock* mechanism prevents Invalid state (a) by prohibiting $L_{10,5}$.

Within each file, TimeArr starts from the most recent materialized chunk version and applies all the changes backwards until it rebuilds the version of interest.

If the selection query includes a range predicate, TimeArr leverages its virtual tiles to identify and process only changes that fall within the region of interest.

Each delta tile $\Delta_{j,j-1}$ keeps track of a constant number of skip links $\Delta_{j,i}$ $_{j-1>i}$ that TimeArr can leverage at version V_j to skip directly to an older version V_i. Each $\Delta_{j,j-1}$ stores the number of versions that $\Delta_{j,i}$ skips as $L_{ji} = (j-i-1)$ as shown in Figure 7. In our experiments keeping track of a small number of L_{ji}'s, $L = 3$, sufficed to hold all skip links for $\alpha < 0.9$. Right before applying $\Delta_{j,j-1}$, TimeArr checks for potential skip links to leverage. TimeArr selects a link that skips the most versions while still landing before or at the desired version. For example in Figure 7(b), in order to retrieve version V_1, TimeArr chooses $L_{10,2} = 7$ at V_{10} and skips 7 delta tiles until it reaches $\Delta_{10,2}$ which means $V_1 = V_{10} + \Delta_{10,2} + \Delta_{2,1}$. These choices are performed separately for each tile.

IV. Approximate Queries

In this section, we present TimeArr's approach to efficiently supporting approximate queries.

A. Distance between Versions

We recall from Section II that, when a user requests the approximate content $c'_j(p)$ of the subarray satisfying predicate p at version number j, the user specifies the maximum tolerable error in the form of an error bound B_1. The system guarantees that the data returned will satisfy the condition $\texttt{Difference}(c'_j(p), c_j(p)) < B_1$. The difference between two subarrays can be computed at the granularity of tiles or chunks as requested by the user. The semantics are as follows:

Listing 1 Distance Function at the Granularity of Tiles

```
// A1 and A2 are two versions of the same tile
double Distance (Subarray A1, Subarray A2)
  Instantiate Statistics object s.
  s.initialize()
  Iterate over all pairs of matching cells (c1,c2)
  where c1 in A1 and c2 in A2 in lock step:
      delta = s.subtract(c1,c2)
      s.process(delta)
  return s.finalize()
```

Listing 2 Distributive Distance Function at the Granularity of Chunks

```
// A1 and A2 are two versions of the same chunk
double Distance (Subarray A1, Subarray A2)
  Instantiate Statistics object s.
  s.initialize()
  Iterate over all pairs of matching tiles t1 and t2 where
  t1 in A1 and t2 in  A2 in lock step:
      delta = Distance(t1,t2)
      s.merge(delta)
  return s.finalize()
```

$$\texttt{Difference}(c'_j(p), c_j(p)) < B_1 \quad iff \quad \forall \text{tiles or chunks } c'_{jk} \in c'_j \quad \texttt{Distance}(c'_{jk}, c_{jk}) < B_1 \tag{2}$$

where $c_j(p)$ is the exact content of the subarray at version j and $c_{jk}(p)$ is the exact content of tile or chunk k in that subarray. The computation includes tiles or chunks that partially overlap the subarray $c_j(p)$.

Similarly, the `Difference` between subarrays is equal to B_1 if the `Distance` between all pairs of tiles or chunks is equal to B_1. If the `Difference` is neither less than B_1 nor equal to B_1, then it is considered to be greater than B_1.

`Distance` functions in TimeArr are implemented in a manner analogous to aggregation functions in OLAP data cubes [26] or parallel aggregations [27]. The distance between two tiles is computed by aggregating the delta values of their cells as shown in Listing 1: the `Distance` function takes two versions of the same tile as input ($A1$ and $A2$). It iterates over the two versions and computes the delta value for each pair of cells. The *subtract* method used here is the same as the one introduced in Section III. It then accumulates these differences using a standard aggregation method.

If the difference computation is at the granularity of chunks, to avoid tedious re-computations, TimeArr requires that the `Distance` function be distributive as defined by Gray *et al.* [26]: max(), count(), and sum() are all distributive. That is, to compute the `Distance` of two chunks, TimeArr aggregates the `Distance` of the underlying tiles as shown in Listing 2.

The user can redefine the subtract and aggregate operations involved in these distance computations as we describe shortly.

TimeArr also requires the `Distance` function to be a *metric* and thus to satisfy the triangle inequality:

$$Distance(A1, A3) \leq Distance(A1, A2) + Distance(A2, A3) \tag{3}$$

where A_i is a subarray. An example of a metric is a `Distance` function that computes the maximum delta value for all cells in the array.

At the core of these `Distance` functions is the `Statistics` object, which defines how the delta values are computed

Listing 3 Statistics Interface

```
interface Statistics
  CellValue add (CellValue, CellValue)
  CellValue subtract(CellValue, CellValue)

  void initialize()
  process(CellValue c)
  merge(Statistics s2)
  double finalize()
```

and aggregated. To implement a new `Distance` function, a user only needs to provide a new class that implements the `Statistics` interface as shown in Listing 3.

The `add` and `subtract` methods operate on delta values as described in Section III. `CellValue` can be any numeric atomic type including integer and real.

TimeArr allows users to provide multiple classes that implement the Statistics interface. TimeArr also provides a *default* `Distance` function using a *default* `Statistics` class that computes a value difference for `subtract` and a value sum for `add`. It also maintains the *absolute* maximum delta value across versions as the aggregate distance returned by finalize.

Next, we present how TimeArr uses these `Distance` functions to answer approximate queries.

B. Approximate Version Selection

Given a segment with a fully materialized chunk version V_r and VersionDeltas for earlier versions V_{r-1} down to V_z (the original version in the segment), the goal is to return some desired version V_j within an error bound B_1.

In the absence of approximation, TimeArr will start with V_r and it will apply delta chunks in sequence (using skip-links when possible) until it gets back to version V_j. With approximation specified at the granularity of tiles, for each tile separately, we want to stop the delta application process as soon as we reach a version V_j' that satisfies the error condition: $\texttt{Distance}(c_{j'k}, c_{jk}) < B_1$, where c_{jk} is the content of tile k at version j and $c_{j'k}$ is the content of that same tile at version j'. If the approximation is specified at the granularity of chunks, TimeArr returns a set of tiles in the approximate chunk $c_{j'k}$ that all have the same version j' and it checks the error condition at the granularity of the whole chunk.

The key question is how to efficiently verify these error conditions? It is impractical to compute the `Distance` function between all versions of each tile in a chunk. Instead, TimeArr computes only two distances for each version V_u: $\texttt{Distance}(\texttt{c}_{\texttt{uk}}, \texttt{c}_{\texttt{u-1,k}})$ and $\texttt{Distance}(\texttt{c}_{\texttt{uk}}, \texttt{c}_{\texttt{zk}})$. We call the former distance the $\texttt{LocDiff}_{uk}$ or *Local Difference at version u and tile index k* because it is a difference between consecutive versions . We call the latter distance the $\texttt{CumDiff}_{uk}$ or *Cumulative Difference at version u and tile index k* because it is the distance to the oldest version in the chunk.

For each new version V_u appended to a chunk, TimeArr computes $\texttt{CumDiff}_{uk}$ and $\texttt{LocDiff}_{uk}$ at the granularity of tiles and stores the results in the `TileDelta StatsVector` for version V_{u-1} (since version V_u will be materialized). Figure 8 illustrates the `CumDiff` and `LocDiff` computation for a small

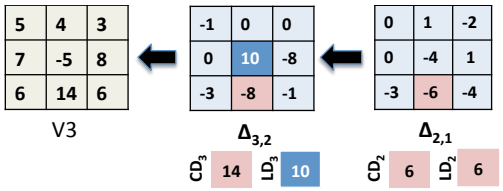


Fig. 8: A 3x3 array with 3 versions. `V3` is materialized and `V2` and `V1` are backward delta encoded. CumDiff (CD) and LocDiff (LD) are calculated for the two $\Delta_{3,2}$ and $\Delta_{2,1}$. Highlighted cells are the ones to contribute to the CD and LD calculations, which use the default distance (maximum absolute difference between any two cells). The VersionDelta for $\Delta_{3,2}$ contains CumDiff_3 and LocDiff_3. Similarly, the VersionDelta for $\Delta_{2,1}$ contains CumDiff_2 and LocDiff_2.

array. Finally, TimeArr merges these `CumDiff` and `LocDiff` values for all tiles in a chunk and stores the chunk-level `CumDiff` and `LocDiff` in the `VersionDelta StatsVector`.

Hence, the `StatsVector` is a vector of pairs `(CumDiff,LocDiff)`, with one pair for the system's default `Statistics` object and extra pairs for all the user-defined `Statistics` objects. In the API in Table I, the arguments that refer to statistics are indexes into the StatsVector.

To verify that the error condition is satisfied, TimeArr leverages the fact that `Distance` is a metric and verifies two conditions. First, since $\text{CumDiff}_{j'k}$ is defined as $\texttt{Distance}(c_{j'k},c_{zk})$, and the distance function is a metric, we have:

$$\texttt{Distance}(c_{j'k}, c_{jk}) \leq \text{CumDiff}_{j'k} + \text{CumDiff}_{jk} \quad (4)$$

Therefore:

$$\text{IF } \text{CumDiff}_{j'k} + \text{CumDiff}_{jk} \leq B_1 \Rightarrow \texttt{Distance}(c_{j'k}, c_{jk}) \leq B_1 \quad (5)$$

Second, TimeArr performs a similar check using LocDiffs:

$$IF \sum_{u=j'}^{j} \text{LocDiff}_{uk} \leq B_1 \Rightarrow \texttt{Distance}(c_{j'k}, c_{jk}) \leq B_1 \quad (6)$$

If either condition holds, $c_{j'k}$ is an approximate version of $c_{jk}^{'}$ that satisfies the error threshold B_1, which avoids further processing of tile k until version V_j.

Algorithm 2 shows how TimeArr utilizes `CumDiff` and `LocDiff` in Equation 5 and 6 to answer approximate selection queries of the form: "Select version V_j of array A with predicate p and ErrorBound B_1". For simplicity, the algorithm is only described for the error computation at the tile granularity, but it follows a similar description for the chunk granularity. For each tile, Algorithm 2 finds the version $V_{j'}$ that satisfies either of the inequalities in Equation 5 or 6. Then it reconstructs and updates $C_j^{'}$ as the approximate version content. It repeats the process for all the tiles separately.

Algorithm 2 uses two bitmasks `ChunkRangeBitmask` and `TileRangeBitmask` that keep track of the chunks and tiles that require to be processed further toward version j. Whenever the `CumDiff` or the `LocDiff` of a given tile satisfies the inequality in Equation 5 or 6, respectively, the corresponding bit value in `TileRangeBitMask` is set to 0, which avoids further triggers of the `ApplyDelta()` function for the same tile. The `ApplyDelta(`c_{jk}`,`$C_j^{'}$`)` executes the `Add()` function on all the corresponding pairs of cells in c_{jk} and $C_j^{'}$.

Algorithm 2 Approximate Selection Queries

1: **Input:** ArrayName A, Predicate p, End VersionNumber j, ErrorBound B_1.
2: **Output:** $C_j^{'}$, approximate content of A at V_j.
3: $C_j^{'} \leftarrow V_r$, $i \leftarrow r$ //current VersionNumber i, materialized VersionNumber r.
4: ChunkRangeBitmask bit set for chunks with cells that satisfy p.
5: TileRangeBitMask bit set for tiles with cells that satisfy p.
6: **while** $i \geq j$ **do**
7: **for all** Chunks C in A **do**
8: $c \leftarrow$ chunk index C
9: **if** ChunkRangeBitMask.getBit(c) == $false$ **then**
10: continue.
11: **end if**
12: **for all** Tiles T in c **do**
13: $t \leftarrow$ tile index T
14: **if** TileRangeBitMask.getBit(t) == false **then**
15: continue.
16: **end if**
17: **if** $\mathrm{CumDiff_{it}} + \mathrm{CumDiff_{jt}} \leq B_1$ or $\sum_{u=j}^{i} \mathrm{LocDiff}_{ut} \leq B_1$ **then**
18: TileRangeBitMask.unsetBit(t).
19: **end if**
20: ApplyDelta($C_j^{'}$,T)
21: **end for**
22: **if** TileRangeBitMask IS ALL ZERO **then**
23: ChunkRangeBitMask.unsetBit(c).
24: **end if**
25: **end for**
26: **if** ChunkRangeBitMask IS ALL ZERO **then**
27: $break$.
28: **end if**
29: $i = i - 1$
30: **end while**

We include $\texttt{CumDiff}_u$ together with $\texttt{LocDiff}_u$ for the approximate version selection computation because it significantly improves the bound. However, one challenge with this approach lies in the efficient computation of the $\texttt{CumDiff}_u$ values. $\texttt{CumDiff}_u$ corresponds to $\texttt{Distance}(V_u, V_z)$ where V_u is the most recent version and V_z is the original version in the segment. In order to calculate $\texttt{CumDiff}_u$, we need to compute $\texttt{Distance}(V_u, V_z)$ at the granularity of tiles and chunks, which means that we need to have a helper `VersionDelta` that keeps track of delta values corresponding to $\Delta_{u,z}$. We name this auxiliary `VersionDelta` aux. At version V_u insertion time, in addition to the regular computation of $\Delta_{u,u-1}$, TimeArr applies $(\Delta_{u,z} + \Delta_{u,u-1})$ to update delta values in the aux `VersionDelta`. Unlike $\texttt{CumDiff}_u$, $\texttt{LocDiff}_u$ values are easy to compute during version insertion since they aggregate delta values between consecutive array versions.

C. Approximate History Selection

`LocDiff` also serves to skip over similar versions during approximate history selection. These are versions for which $\texttt{Difference}(V_{u+1}, V_u) \leq B_2$, which is directly captured by the `LocDiff` values.

Algorithm 3 shows the details of how TimeArr extracts approximate history at the granularity of tiles (algorithm for chunk granularity is very similar). The algorithm proceeds in two phases. In the first phase, TimeArr extracts the header information using the statisticsID s specified by the user. From the header information, TimeArr extracts all the `LocDiff` values at the granularity of either tiles of chunks as requested by the user. The algorithm then runs a standard SciDB query to identify all versions V_u that differ by more than B_2 from their successor V_{u+1}. The query issued on line 6 in Algorithm 3 captures the maximum variation between adjacent versions and it checks if the difference is high enough to satisfy the lower-bound constraint B_2. TimeArr outputs the version numbers of these versions into a variable named *res*. Recall that TimeArr checks B2 at the same granularity as B1.

Algorithm 3 Approximate History Queries

1: **Input:** ArrayName A, Predicate p, Start VersionNumber k, End VersionNumber j, ErrorBound B_1, ErrorBound B_2, StatisticID s.
2: **Output:** A sequence of contents C containing all matching version contents $C_j^{'}$.
3: **PHASE ONE: Header Information Extraction.**
4: Extract header info using StatisticID s for the tiles that match p from V_k to V_j.
5: Store result in array A_{head}. // Schema: $A_{head}\{CumDiff,\ LocDiff\ \}$[v][t]
// Extract all versions that meet the B_2 bound (v is VersionNumber, t is TileIndex):
6: res = SELECT v FROM A_{head} WHERE EXISTS (SELECT t FROM A_{head} WHERE $A_{head}[v][t].LocDiff \geq B_2$)
7: **PHASE TWO: Bulk Approximate Version Selection**.
8: **for all** i in res **do**
9: $C_i^{'} \leftarrow$Select(A_p, p, i,B_1,s)
10: C.add($C_i^{'}$)
11: **end for**

The second phase retrieves the actual version contents using the version numbers and a bulk approximate selection query that scans all past versions and returns the desired ones with the required degree of precision B_1.

V. Evaluation

In this section, we evaluate TimeArr's performance on two real datasets and two synthetic datasets. All experiments are performed on dual quad-core 2.66GHz Intel/AMD OpteronPentium-based machines with 16GB of RAM running RHEL5. We use the following datasets.

Astronomy Universe Simulation dataset (Astro). The first dataset comprises 9 snapshots from an astronomy simulation named *cosmo50* [19]. Each snapshot is 1.6 GBs in size and represents the universe as a set of particles in a 3D space. To represent the data as an array with integer dimensions, we create a $(500 \times 500 \times 500)$ array and project the array content. Following SciDB's column-based array representation, we perform all experiments on the array containing the data for the `mass` attribute of the particles. We divide this array into 8 chunks, each containing one eighth of the logical size of the array and each having 1000 virtual tiles.

Global Forecast System Model dataset (GFS). We also experiment with the "GFS" dataset from the National Oceanic and Atmospheric Administration (NOAA) [8]. We use data from a 180 hour weather forecast simulation sampled every 3 hours, for a total of 61 grids, each about 1MB in size. Each grid is a (720×360) two dimensional array (one array in one chunk) and we consider 100 virtual tiles for this single chunk.

Gaussian Distribution (Synthetic dataset 1). The synthetic dataset comprises a single, dense two-dimensional array chunk with 1000×1000 cells. The chunk is divided into one hundred 100×100 tiles unless mentioned otherwise. We create synthetic versions by randomly updating the array. The probability that a cell will be updated follows a normal distribution centered at coordinate $[500][500]$. For a normal distribution, 99.8% of all values fall within 3 standard deviations of the mean. Hence we pick sigma to be $\frac{1}{6}$ of the dimension length. Each update consists of the addition of a marginal value ($<$127) . Each snapshot is 8 MB in size.

Uniform Distribution (Synthetic dataset 2). The synthetic dataset follows the same description as the Gaussian Distribu-

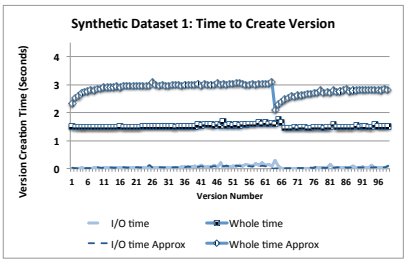


Fig. 9: Time to create 100 versions of a two-dimensional array with normally distributed updates. A new segment is initialized at version 65 in the non-approximate setting and version 62 when approximations are enabled. Each new version adds a constant overhead. I/O overhead is insignificant.

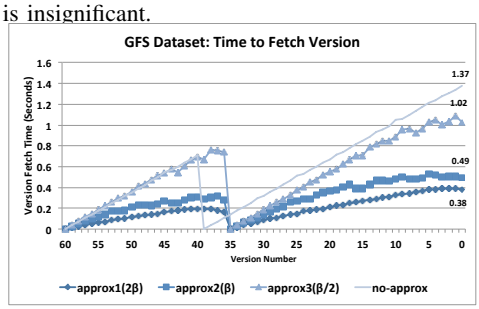


Fig. 10: Time to fetch each version in the GFS dataset. β, 2β, and $\frac{\beta}{2}$ refer to the error bounds, where β is the average maximum change observed in two adjacent versions. I/O times are insignificant and not shown. For the GFS dataset the maximum segment size is 12 MBs. The segments reaches its full size at version number 38 and 36 in the non-approximate and approximate settings respectively.

tion except the probability that a cell will be updated follows a uniform distribution.

A. Basic Version Creation and Retrieval Performance

We first evaluate the performance when appending new versions to an array. Figure 9 shows the time to create 100 new versions for the first synthetic dataset. Each new version adds a constant overhead (1.5 seconds) in the non-approximate setting. The overhead is constant primarily because the total number of updates is approximately the same for each version. As we show later, the version creation time grows almost linearly with the number of updated cells per version. The overhead of creating a new version is higher with approximation enabled. The extra overhead comes primarily from updating the aux `VersionDelta` in addition to the main `VersionDelta`, doubling processing times. The I/O times in both cases are insignificant. The CPU cost of computing delta values dominates the runtime. In this experiment, we arbitrarily set the segment size to 48 MBs. When a new segment is created the version creation time with approximation is close to the non-approximate setting primarily because the aux VersionDelta starts-off empty and is thus quick to update. We observe the same trend with the real datasets, not shown due to space constraints, but available in our technical report [28].

Next, we study the query processing time to fetch each version either precisely or approximately in synthetic and real datasets. For the experiments with approximation, we consider an error bound β equal to the average maximum change observed between any two adjacent versions. With such an error bound, the user may see values that are in aggregate off by at most one array version. As Figure 10 shows, the cost of retrieving a version precisely decreases linearly with the version number. With high approximation (error bound β), for the GFS dataset, query times decrease by factors between 25% and 300%. Similarly, we observed that retrieving any version in the synthetic dataset (not shown in the figure) takes half the time or less compared with retrieving the exact version. Even with a small approximation (error bound $\beta/2$), performance gains are above 35%. We observed similar trends for the `Astro` dataset (not shown).

Overall, TimeArr's approach to approximate query processing thus adds overhead during version insertion. This overhead, however, is paid only once. At the same time, approximation enables the system to cut query times significantly when users can tolerate approximate results. These savings are repetitive. Interestingly, the performance gains of approximation increase as we query older versions while the version creation overhead remains constant.

We also studied the effect of the number of updates on the version creation time. We calculated the total time to create 50 versions from the `synthetic dataset 2` with different numbers of updates ranging from one thousand to one million updates per version. As expected version creation time grows almost linearly with the number of updates per version. Going from one thousand to one million updates always added 7 to 8 seconds to the total version creation time. We did a similar experiment keeping the number of updates constant, but increasing the updated values. We observed no significant increase in the version creation time (although the size of the version in terms of bytes changed rapidly). We do not show graphs of those experiments due to space constraints.

We also evaluated the benefit of using virtual tiles and variable-length delta encoding on the storage space at version creation time and we observed up to 70% space savings compare to the case with no-tile settings and no variable-length delta encoding. The space savings, however, do not come for free. The finest tile settings in the experiment had up to 25% version creation time overhead compare to the no-tile settings.

We now evaluate the benefits of using virtual tiles to speed-up historical queries over subsets of a chunk (*i.e.*, range selection queries over array coordinates). Figure 11 shows the performance of the following query: `Return the original version of the rectangular subarray [`C_1`;`C_2`]`, where C_1 and C_2 are the upper-left and lower-right corners of a region. In Figure 11(a), we use a single chunk with 100 virtual tiles and Synthetic dataset 1. The rectangular regions have the same center as the chunk and range from one tile to the whole chunk. The performance gains that we achieve using virtual tiles depend on the granularity of the tiles and the size of the fetched region. In the tile setting in this experiment, we fetch a single tile 50 times faster than the setting with no virtual tiles used. Even with the window sizes that retrieve as much as 25% of the chunk, TimeArr runs significantly faster than the setting without virtual tiles. Figure 11(b) shows the performance of the range selection query on the astronomy dataset. Similar

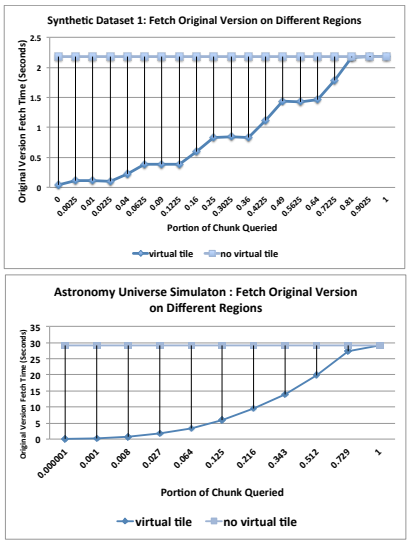


Fig. 11: Time to fetch the original version where the region window changes from one tile to the whole chunk. (a) Synthetic dataset with 20 versions. (b) Astronomy dataset with 9 versions.

to what we observed in the synthetic dataset, the benefits of using virtual tiles to speed-up historical queries over subsets of an array are significant. The trend is the same for the GFS dataset as well.

B. Version Retrieval with Links Support

The advantage of the skip link technique is highlighted on datasets such as the *Global Forecast System Model* (GFS) where similar data patterns are repeated at different versions. Figure 12 shows the time to fetch 60 versions of the GFS dataset. The segment size is chosen such that all the versions reside in one segment. In this experiment, TimeArr periodically computes skip links after each T versions appended. As illustrated in Figure 12 the performance gain to fetch the oldest version with skip links is 42% for $T = 20$ and 75% for $T = 1$ compared to the no-link case. However, the skip link computation incurs overhead at version insertion time. Table II summarizes the overhead for different values of T. Although exhaustive computation of skip links ($T = 1$) improves the performance in Figure 12, it incurs significant overhead when inserting new versions. Finding the optimal interval T is left for future work. Instead, TimeArr uses *lazy* computation of skip links whose performance is shown in Figure 13. The query workloads, `Q-NORM` and `Q-UNIFORM` are as follows: TimeArr appends 61 versions from the GFS dataset in total and between each append operation, we issue 5 original-version retrieval queries (305 queries in total). Each original-version retrieval query only fetches a few tiles from the array. The tiles to be fetched are selected randomly based on either normal distribution (`Q-NORM`) or uniform distribution (`Q-UNIFORM`). Figure 13(a) shows the advantage of the *lazy* link computation with the `Q-NORM` workload. `Q-NORM` simulates a workload with a *hot spot* region; *i.e.*, a number of tiles are fetched many times while other tiles are fetched only once. Figure 13(a) shows that lazy computation of skips links is better than the skip-link computation at version insertion time

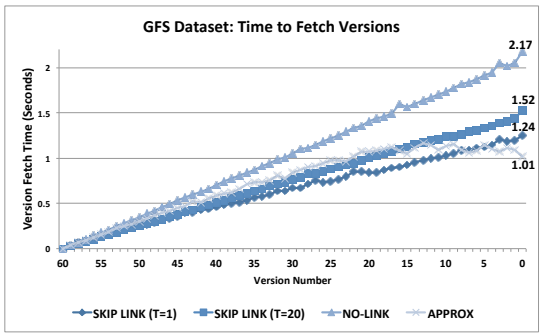


Fig. 12: Time to fetch each version in a single-chunk array with 60 versions. The result with skip link is competitive with approximate result with β error bound.

	NoLink (Lazy)	Link (T=20)	Link (T=5)	Link (T=1)
Add Versions (sec)	13.16	12.00	13.48	14.56
Create Links (sec)	0	3.68	11.11	47.13

TABLE II: GFS dataset: Skip links overhead at version insertion time. TimeArr computes skip links each T consecutive versions appended.

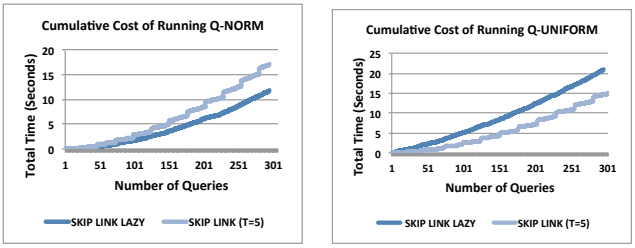
with interval $T = 5$. However, this is not true when TimeArr runs the `Q-UNIFORM` workload (Figure 13(b)), because the skip-link computation overhead for a specific tile at version fetch time is not paid off later. In the `Q-UNIFORM` workload, many tiles are only fetched once. In Figure 13, lazy computation of skip links during version retrieval incurs approximately 2 seconds of overhead in total (not shown in the figure). The algorithm to decide when to compute skip links lazily during version retrieval, when to compute them after certain intervals at version insertion time, and possibly the combination of these two approaches are left for future work.

C. Comparison with SciDB

The current SciDB version storage also uses backwards deltas [20]. Unlike TimeArr, however, it represents each `VersionDelta` using two chunks, one with a sparse and the other with a dense representation. Each cell-value in the `VersionDelta` is either in the sparse or dense chunk.

We compare TimeArr to SciDB's current storage manager using the `synthetic dataset 2`. There is thus a total of 10^6 cells in a single-chunk array. We create four synthetic streams of versions: *mass* updates, *medium* updates, *rare* updates, and *very rare* updates that correspond to 10^6,10^5,10^4, and 10^3 updates between each array version respectively. The approximation feature is turned off in all the experiments. Table III shows the results. TimeArr outperforms SciDB in all four cases. It achieves 40% version creation time savings for medium and mass updates. Table III also shows that version creation time variation in SciDB is much larger than TimeArr in the case of medium and mass updates. In TimeArr the overhead of adding a new version is constant while this is not the case in SciDB.

Figure 14 shows the query processing performance of both approaches when fetching the whole chunk at a specific, precise version. The chunk has 100 tiles. TimeArr achieves a 1.6X to 6.6X performance gain in terms of query processing compared with SciDB for mass and medium updates. For rare updates, improvement is marginal (it is not shown in



(a) (b)

Fig. 13: Cumulative query runtime of workloads `Q-NORM` and `Q-UNIFORM`. Skip links are computed either lazily at version retrieval or at version insertion time after each $T = 5$ versions appended.

updates		mass	medium	rare	very rare
TimeArr (sec)	AVG	1.16	0.60	0.50	0.48
TimeArr (sec)	STD	0.08	0.01	0.04	0.05
SciDB(sec)	AVG	1.80	1.01	0.67	0.55
SciDB(sec)	STD	0.83	0.45	0.85	0.60

TABLE III: Average time to create one version after appending 50 versions on a two-dimensional array with uniformly distributed updates. TimeArr outperforms SciDB in all four cases.

the figure). TimeArr's performance gains compared to SciDB come from the fact that SciDB stores delta values in one dense and one sparse chunk for compactness. When fetching a version, SciDB first needs to combine delta values from both representations, which incurs significant overhead. Also, TimeArr uses bitmask techniques to locate changes efficiently, while SciDB needs to iterate over the whole dense delta chunk. Overall, our design decision to have a single storage representation for delta chunks is a key factor for TimeArr's query time performance.

We also studied the advantage of using virtual tiles in TimeArr compared to the current implementation of SciDB. We did a similar experiment as the one in Figure 11. We observed two orders of magnitude improvement in TimeArr for regions covering only a few tiles (Figure is not shown due to space constraint. The trend is similar to Figure 11).

D. Approximate History Query

We now demonstrate the benefits of approximate history query using the GFS dataset. We execute the following example query: `Select(`A_{GFS}`,true,`V_1`,`V_{61}`,54,260)` where A_{in} is the input array. This query asks for all 61 versions of the dataset such that each version is approximately returned with the error bound $B_1 = 54$ and only versions that differ by at least error bound $B_2 = 260$ are returned. 260 is approximately half of the maximum change observed in two adjacent versions. Figure 15 shows the result of this query. TimeArr quickly identifies that only 9 versions differ by more than the specified threshold and it only requests to approximately fetch these 9 versions. In contrast, with exact history TimeArr has to fetch all the versions. The approximate query runs in less than 7.5 seconds and the equivalent precise query takes 37 seconds to complete, which is a 5X performance difference.

VI. Related Work

Delta encoding is a popular technique in video and image compression. Video compression codecs like `MPEG-1` [29]

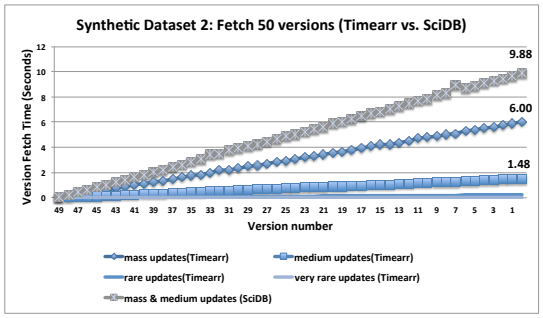


Fig. 14: Time to fetch each version from 1 to 50 on a two-dimensional array with uniformly distributed updates. TimeArr is about 1.6X to 6.6X better than SciDB for mass and medium updates. "Rare updates" and "very rare updates" lines overlap for both systems. Only TimeArr is shown.

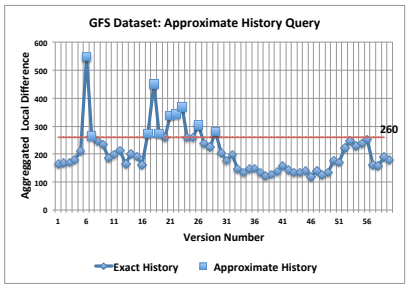


Fig. 15: Approximate history query returns only 9 versions out of 61 with a maximum degree of changes from the previous version greater than 260, while in exact history , TimeArr has to go over all the versions. The performance gain is almost 5X.

apply several delta encoding techniques both within and between frames. Similar to TimeArr, they regularly materialize versions (frames) in a chain of delta frames, They also divide the frames into smaller chunks and compare each chunk to every possible region in a specified radius around its origin. Hence, their version insertion is expensive. Previous work [20] showed that although video compression techniques efficiently compress arrays, the version import time is too expensive and consequently not appropriate for versioning in array systems.

Most array engines being built today, such as RasDaMan [5], are *not* designed as no-overwrite storage systems and consequently cannot naturally support versioning. NetCDF [30], [31] and HDF5 [32] are common data models that provide a portable and efficient mechanism to store and access multidimensional data which are extensively used by scientists, but they also do not support versioning explicitly.

MOLAP [33] systems store data in multidimensional arrays [33], [34]. They focus on aggregation queries and exploit data structures to efficiently compute rollups. The MOLAP system in [35] supports versions to represent changes to the data sources that should be propagated to the data warehouse periodically. But the versioning system is designed to benefit the concurrency control mechanism in order to minimize contention between query and maintenance transactions.

There is a long line of research on temporal databases [10], [36], [37], [38]. Temporal databases have two notions of time: "valid time" and "transaction time". Many databases provide time-travel support along the transaction time dimension [9],

[10], [39], [40]. However, none of these databases is specialized for time travel over array data nor approximate time-travel. In particular, Postgres [39] uses R-trees for version management. This technique is complementary to the approach that we propose in this paper. Immortal DB [10] adds transaction time database support into a database engine. For this, Immortal DB stores versions data as a linked list, while we store versions as delta values. Our versioning system also heavily applies array-oriented techniques including bitmasks, virtual tiles, and skip links. Finally, TimeArr supports a new type of "approximate queries" in the context of scientific array database engines.

Version Control Systems are an old topic in computer science. Versioning techniques such as forward and backward delta encoding and the use of multi-version B-trees have been implemented in various legacy systems. Git [41] is one of the conventional version-control systems and is believed to be faster and more disk efficient than other similar version-control systems. Our system borrows some ideas such as backward delta encoding from other version control systems such as Git, but we also use sophisticated array-oriented optimization techniques to efficiently encode the delta versions and to support approximate queries.

Lastly, the state of the art for versioning in array systems [20] uses a materialization matrix to efficiently find the best versions to materialize. We are similar to this recent prior work [20] in the sense that we also use backward delta versions and store and fetch consecutive deltas together. The ability of our system to add skip links at the granularity of tiles, to approximately answer queries, and our use of virtual tiles to support versioning at fine granularity are the main advantage of our system compared to this prior work [20].

VII. Conclusion

TimeArr is a new storage manager for an array database. Its key contribution is to efficiently store and retrieve versions of an entire array or some sub-array. TimeArr also introduces the idea of approximate exploration of an array's history. To achieve high performance, TimeArr relies on several techniques including virtual tiles, bitmask compression of changes, variable-length delta representations, and skip links. TimeArr enables users to customize their exploration by specifying both the maximum degree of approximation tolerable and how it should be computed. Experiments with a prototype implementation on two real datasets demonstrate the performance of TimeArr's approach.

VIII. Acknowledgements

This work is partially supported by NSF grant IIS-1110370 and the Intel Science and Technology Center for Big Data.

References

[1] Large Synoptic Survey Telescope. http://www.lsst.org/.
[2] Earth microbiome project. http://earthmicrobiome.org/.
[3] Stonebraker et. al. Requirements for science data bases and SciDB. In *Fourth CIDR Conf. (perspectives)*, 2009.
[4] Ballegooij et. al. Distribution rules for array database queries. In *16th. DEXA Conf.*, pages 55–64, 2005.
[5] Baumann et. al. The multidimensional database system RasDaMan. In *SIGMOD*, pages 575–577, 1998.
[6] Rogers et. al. Overview of SciDB: Large scale array storage, processing and analysis. In *Proc. of the SIGMOD Conf.*, 2010.
[7] Zhang et. al. RIOT: I/O-efficient numerical computing without SQL. In *Proc. of the Fourth CIDR Conf.*, 2009.
[8] National Oceanic and Atmospheric Administration. http://nomads.ncdc.noaa.gov/.
[9] Oracle Flashback Technology. (2005). http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm.
[10] Lomet, D. et. al. Transaction time support inside a database engine. In *Proc. of the 22nd ICDE Conf.*, 2006.
[11] Chang et. al. Titan: A high-performance remote sensing database. In *ICDE*, pages 375–384, 1997.
[12] DeWitt et. al. Client-server paradise. In *Proc. of the 20th VLDB Conf.*, pages 558–569, 1994.
[13] Marathe et. al. Query processing techniques for arrays. *The VLDB Journal*, 11(1):68–91, 2002.
[14] Soroush, E. et. al. Arraystore: A storage manager for complex parallel array processing. In *Proc. of the SIGMOD Conf.*, 2011.
[15] Munro, J.I. et. al. Deterministic skip lists. In *SODA '92*, 1992.
[16] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.
[17] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.
[18] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *Proc. of the SIGMOD Conf.*, June 2002.
[19] Loebman et. al. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *IASDS'09*, 2009.
[20] Seering, A. et. al. Efficient versioning for scientific array databases. In *Proc. of the 28th ICDE Conf.*, 2012.
[21] Shimada et. al. A storage scheme for multidimensional data alleviating dimension dependency. In *ICDIM*, pages 662–668, 2008.
[22] Chang, F. et. al. Bigtable: a distributed storage system for structured data. In *Proc. of the 7th OSDI Symp.*, 2006.
[23] Chang et. al. T2: a customizable parallel database for multi-dimensional data. *SIGMOD Record*, pages 58–66, 1998.
[24] Otoo et. al. Optimal chunking of large multidimensional arrays for data warehousing. In *Proc. of the 10th DOLAP Conf.*, pages 25–32, 2007.
[25] Sarawagi et. al. Efficient organization of large multidimensional arrays. In *Proc. of the 10th ICDE Conf.*, pages 328–336, 1994.
[26] Gray, J. et. al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, pages 29–53, 1997.
[27] Yu et. al. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of the 22st SOSP*, 2009.
[28] Soroush, E. et. al. Time travel in a scientific array database. Technical Report UW-CSE-12-11-03, Department of Computer Science, University of Washington, 2012.
[29] International Standards Organization (ISO), Coding of Moving Pictures and Audio. http://mpeg.chiariglione.org/standards/mpeg-1/mpeg-1.htm.
[30] The NetCDF Users' Guide. http://www.unidata.ucar.edu/packages/netcdf/guidec/.
[31] Rew et. al. Data management: Netcdf: an interface for scientific data access. *IEEE Comput. Graph. Appl.*, 10(4):76–82, 1990.
[32] Introduction to HDF5. http://www.hdfgroup.org/HDF5/doc/H5.intro.html.
[33] P. M. Fernandez. Red brick warehouse: a read-mostly rdbms for open smp platforms. *Proc. of the SIGMOD Conf.*, pages 492–, 1994.
[34] Pedersen et. al. Multidimensional database technology. *IEEE Computer*, 34(12):40–46, 2001.
[35] H.G. Kang and C.W Chung. Exploiting versions for on-line data warehouse maintenance in molap servers. In *Proc. of the 28th VLDB Conf.*, pages 742–753, 2002.
[36] Ozsoyoglu, G. et. al. Temporal and real-time databases: A survey. *IEEE TKDE*, pages 513–532, 1995.
[37] Jensen, C.S. et. al. Temporal data management. *IEEE TKDE*, pages 36–44, 1999.
[38] R. Snodgrass and I. Ahn. A taxonomy of time databases. In *Proc. of the SIGMOD Conf.*, pages 236–246, 1985.
[39] M. Stonebraker. The design of the postgres storage system. In *Proc. of the 13th VLDB Conf.*, pages 289–300, 1987.
[40] L. Hobbs and K. England. Rdb: A Comprehensive Guide. Digital Press, 1995.
[41] S. Chacon. the Git SCM community. http://book.git-scm.com/, 2010.