DyVer: Dynamic Version Handling for Array Databases

Amelie Chi Zhou
Shenzhen University

Zhoubin Ke
Shenzhen University

Jianming Lao
University of California Irvine

ABSTRACT

Array databases are important data management systems for scientific applications. In array databases, version handling is an important problem due to the no-overwrite feature of scientific data. Existing studies for optimizing data versioning in array databases are relatively simple, which either focus on minimizing storage sizes or improving simple version chains. In this paper, we focus on two challenges: (1) how to balance the tradeoff between storage size and query time for numerous version data, which may have derivative relationships with each other; (2) how to dynamically maintain this balance with continuously added new versions. To address the above challenges, this paper presents DyVer, a versioning framework for SciDB which is one of the most well-known array databases. DyVer includes two techniques, including an efficient storage layout optimizer to quickly reduce data query time under storage capacity constraint and a version segment technique to cope with dynamic version additions. We evaluate DyVer using real-world scientific datasets. Results show that DyVer can achieve up to 95% improvement on the average query time compared to state-of-the-art data versioning techniques under the same storage capacity constraint.

CCS CONCEPTS

• **Information systems** → **Version management**; **Parallel and distributed DBMSs**;

KEYWORDS

Scientific Data Management, Array Database, Versioning

ACM Reference Format:
Amelie Chi Zhou, Zhoubin Ke, and Jianming Lao. 2023. DyVer: Dynamic Version Handling for Array Databases . In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3577193.3593734

1 INTRODUCTION

Array database management systems have been designed to address the unique challenges of managing scientific datasets [2, 5], which often undergo revisions, updates, and improvements in data collection and analysis methods. For instance, CMIP [12] coordinates and compares climate model simulations worldwide, but evaluating consistency across phases [17] requires extensive data analysis, including examining thousands of versions of data outputs from a single model. In scientific applications, it is common to keep multiple versions of the same data. How to manage versioned data efficiently is one important topic for array databases.

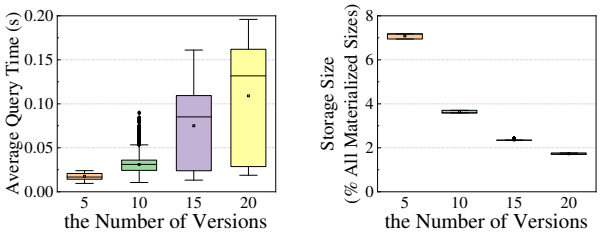


Figure 1: Comparison of the average query time and storage size using different version layouts. Storage size is normalized by the total storage size of all versions materialized.

SciDB [5] is a pioneer scientific database proposed by Michael Stonebraker initially to cope with the special data management requirements of scientific applications [7]. It employs an array-oriented storage model with no-overwrite processing, which means that any modification on arrays will be stored as a new version rather than directly overwriting the original one. To reduce the storage size, SciDB only stores the latest version in its entirety (or called materialization) and stores the differences (or called delta) between adjacent versions. An older version can be recreated by going through the corresponding chain of deltas. Clearly, the time taken to query a particular version is proportional to the number of versions required to be rebuilt [22]. On the other hand, we can trade off some storage size to materialize some versions in the version chain for improving the efficiency of querying older versions. How to balance the trade off between the storage size and version query performance is however non-trivial. In addition, the complete set of versions is not known in advance. How to accommodate the dynamic insertion of new versions and maintain a dynamic balance is another concern of this paper.

In SciDB, each version can be stored in two forms: materialized or delta-ed against other versions. Given N versions, there will be N^N configurations for version layout and finding the best one is very time-consuming when N is large. Figure 5 provides some visual examples of the different version layout configurations. We conducted an experiment to demonstrate how different version layouts can result in varying retrieval performance. Figure 1 presents the average query time and the overall storage size using different version layout solutions for different numbers of versions using the ILATM2 [20] dataset. Specifically, we use Xdelta3 [13] to calculate the delta between versions. The storage size is normalized by the total storage size of all versions materialized. We produced up to 30,000 different layouts at random. Each one only materializes the latest versions. Figure 1 indicates that as the number of versions

ICS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593734

grows, the variance of the average query time becomes larger. In contrast, the storage size does not vary significantly under all number of versions. This means that, even with similar storage size, different layouts can still generate quite different query efficiency for versioned data. Furthermore, as new versions are added, it becomes more and more difficult to find the optimal version storage layout.

Existing studies [18, 19, 22] have proposed various ways to optimize data versioning in SciDB. However, the ways to handle versions proposed in these studies are relatively simple. For example, Seering et al. [18] utilized minimum spanning trees to generate layout solutions with the minimal storage size. Query performance is not specifically discussed in their work. Soroush and Balazinska [19] proposed a method for constructing Skiplinks between non-contiguous versions in a version chain, which may not suitable for more complicated version graphs. Version storage management has been studied in other databases, such as relational databases [9, 14] and graph databases [10, 11]. However, their version handling techniques are difficult to apply to SciDB due to the different data models.

In this paper, we propose DyVer – a versioning framework that balances the trade off between the storage size and version query performance while accommodating dynamic version additions. The goal of DyVer is to minimize version query time while satisfying a given storage capacity constraint. Due to the large solution space, DyVer adopts a two-stage optimization method to quickly generate a good storage layout. Specifically, we first minimize the storage size for all versions and then use the remaining storage capacity to increasingly improve version query performance. For dynamic versions, the time to generate delta is a major performance bottleneck (about 80x of algorithm runtime). To cope with it, DyVer proposes a virtual container named *version segment*. Version segment is sized according to the update gap of the version set. It can quickly find a good layout for new versions without impacting much the existing storage layouts. We integrated DyVer in SciDB and evaluated its efficiency using real-world versioning data. Our experiments show that DyVer can achieve up to 95% improvement on average version query time compared to the SciDB with the state-of-the-art data versioning technique under the same storage constraint.

This paper makes the following contributions:

- **Optimal Storage Layout.** We proposed an efficient two-stage method to optimize the version data layout, in order to minimize data query time while satisfying storage capacity constraint.
- **Version Segment.** Our segment-based design can limit the time needed for calculating deltas and thus greatly reduce version insertion overhead. It help DyVer to achieve a dynamic balance between storage size and query performance in case of version additions.
- **Evaluation.** We implement our methods using a simulator. We evaluate DyVer using real-world scientific datasets. Experimental results indicate that our system can achieve up to 95% improvement on average version query time compared to the SciDB with state-of-art data versioning technique under the same storage constraint.

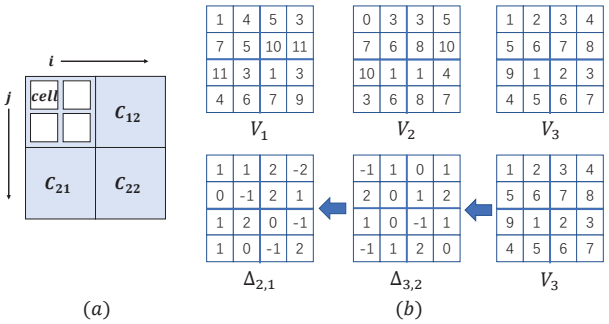


Figure 2: Illustration of version storage in SciDB: (a) a 4×4 array is divided into 4 chunks and each chunk has 2×2 cells; (b) a chain of versions where V_3 is materialized and the rest are stored using backward delta.

The rest of this paper is organized as follows. Section 2 presents the background and related work. Section 3 shows the problem formulation between the storage capacity and querying performance. We describe our design overview in Section 4. We detail our optimal storage layout in Section 5 and version addition technique in Section 6. Finally, we present the performance of our system in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Array Databases

Many scientific applications need to process and analyze large amounts of multi-dimensional data (e.g., multi-dimensional raster data in Earth Sciences). A number of array databases [2, 5, 16] have been proposed to efficiently store and manage multi-dimensional scientific data (or called arrays). For example, RasDaMan [2] is the pioneer array database which was first proposed for geospatial data. It supports dense multi-dimensional arrays of arbitrary size and structure. TileDB [16] is an open-source array storage engine. It treats all data as arrays and provides performant and versatile storage services for different applications. TileDB also supports efficient time travel queries over array versions, which is currently missing in RasDaMan. However, TileDB is not a database management system [21]. It does not yet offer any array processing functions [24], which is not suitable for scientific data analysis. SciDB [5] is another database which treats arrays as the first-class data model. SciDB was designed specifically according to the requirements of scientific applications [7]: the storage layer of SciDB is highly optimized for array data and versioning is also supported with fine-grained optimizations [18, 19, 22]. In this paper, for clarity, we focus on SciDB to discuss the versioning problem in array databases. However, our techniques are general and can shed lights to other array and multi-dimensional databases with versioning support.

2.2 Array Data Versioning

For scientific applications, one important feature required is the support of storing and querying named versions [7]. We first introduce the version storage and querying solutions in SciDB.

Version storage. In SciDB, data objects are modeled as arrays. Arrays are divided into storage chunks, each of which is further divided into multiple cells or elements. For example, Figure 2 shows a 4×4 array stored using 4 evenly sized chunks, each of which contains 2×2 cells. Every cell stores multi-dimensional data specified by users. SciDB uses regular chunking strategy and compresses values by run-length encoding.

SciDB stores data in a "no overwrite" manner. Every operation that updates existing arrays creates a new version of the array. As illustrated in Figure 2, SciDB by default stores the newest version as it is (i.e., materialized) and the older versions using backward deltas. An older version can be recreated by going through the corresponding chain of deltas, e.g., $V_1 = V_3 + \Delta_{3,2} + \Delta_{2,1}$.

Version querying. SciDB adopts a shared-nothing architecture and can be deployed on a cluster of servers. When receiving a query from clients, SciDB distributes fragments of the query plan to the corresponding nodes of the cluster for execution [5]. For example, a user submits a query to access the data of V_1. Then each chunk of V_1 will be recreated separately using the materialized and delta chunks stored in the cluster. Querying performance depends largely on the length of the chain used to recreate the requested version.

2.3 Related Work

The storage layout of versioned data, namely which versions to materialize, is important to the trade-off between storage size and querying performance in array databases. Existing studies have proposed various ways to optimize data versioning in SciDB. For example, Seering et al. [18] formulated the version data storage problem as a graph problem, and adopted a minimum-spanning-tree based algorithm to decide which versions to materialize and how to encode delta versions, in order to minimize the overall storage size. Soroush and Balazinska [19] proposed *skip links*, which are backward delta between non-consecutive versions, in the chain of versioned data to accelerate the access of older versions. Xing et al. [22] enables the convertion from SciDB chunks to HDF5 files, which allows scientists to manipulate versioned array data in SciDB using HDF5-based programs. Although the above studies are proposed specifically for SciDB, the version handling considered in these studies are relatively simple. For example, TimeArr [19] only considered single chains of versions, while the structures of real version graphs are much more complicated due to the collaborative and interactive features of scientific applications [23].

Some studies have noticed the complex relationship between versions for relational databases [9, 14], graph databases [10, 11] and unstructured datasets [3]. Due to the different data model between array databases and these systems, their version handling techniques can hardly be applied to SciDB.

3 PROBLEM FORMULATION

Let $\mathcal{V} = \{V_i | i = 1, 2, 3...n\}$ be a collection of versions stored in the array database. We use a directed acyclic graph (DAG) to represent the relations between versions. Denote the version graph as $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where an edge represents the provenance relationship between two versions. For example, given versions V_i and V_j, edge $\langle V_i, V_j \rangle \in \mathcal{E}$ means that V_i is the parent version of V_j. The weight

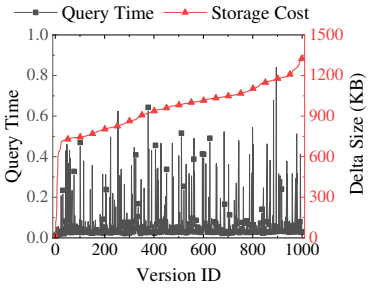


Figure 3: The storage cost and query time resulted from different encode solutions.

of the edge stands for the differences between two versions, i.e., the weight of $\langle V_i, V_j \rangle$ equals to the size of $\Delta_{i,j} = V_j - V_i$.

In this paper, we study the storage size and query time of versioned data in array databases. Given a version V_i, we have two storage choices:

- **Materialized**: the entire version is stored as it is on disk. We denote the storage size of V_i as $\Delta_{i,i}$ and the querying time of V_i (i.e., loading the chunks of the version) as $t_{i,i}$.
- **Recreated**: only the difference of V_i from another version V_j is stored and accessing V_i requires recreating it from V_j. Denote the storage size of the difference as $\Delta_{j,i}$ and the recreation time of V_i from V_j as $t_{j,i}$ (including delta loading and decompression).

Thus, for any version $V_i \in \mathcal{V}$, we can formulate its storage size $S(i)$ and querying time $T(i)$ as below:

$$S(i) = \begin{cases} \Delta_{i,i}, & \text{if } V_i \text{ is materialized} \\ S(p(i)) + \Delta_{p(i),i}, & \text{otherwise} \end{cases} \quad (1)$$

and

$$T(i) = \begin{cases} t_{i,i}, & \text{if } V_i \text{ is materialized} \\ T(p(i)) + t_{p(i),i}, & \text{otherwise} \end{cases} \quad (2)$$

where $p(i)$ is a parent version of V_i over which a delta is stored.

For each version, there can be more than one option of $p(i)$. Especially, the number of options becomes much larger in version *graphs* than *chains*, which increases the complexity of the problem. In this paper, we study the version layout problem which decides which versions to materialize and how to select the best $p(i)$ for recreated versions. Our goal is to minimize the average version querying time while satisfying the storage capacity constraint (i.e., the available disk space or a quota). Formally, we define the problem as below.

$$\min \sum_{i \in \mathcal{V}} T(i) \quad (3)$$

s.t.,

$$\sum_{i \in \mathcal{V}} S(i) < Capacity \quad (4)$$

The above problem is NP-hard. Further, we notice that it is hard to find a clear correlation between the storage cost and query time optimization goals. For example, in Figure 3, we show the storage cost (i.e., $\Delta_{j,i}$) and query time (i.e., $t_{j,i}$) of an early version over other versions using SPL2SMP dataset [6]. The delta files are decompressed using Xdelta3 [13]. The difference between two distant

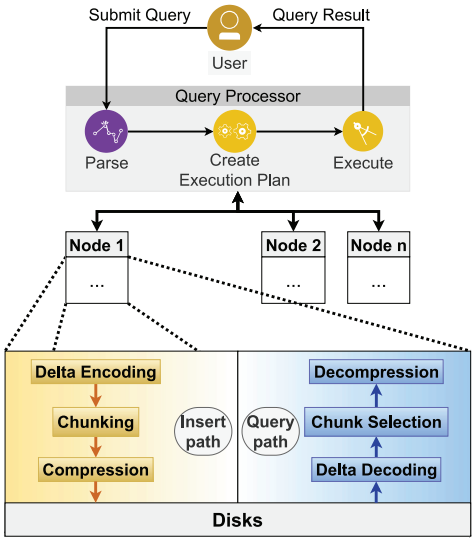


Figure 4: Design overview of DyVer

versions is larger than two close-by versions. Thus the delta file size gets larger with the version ID. However, the query time of the version varies and there is no clear trend. This is because 1) the delta file decompressing time is not linearly correlated to the file size and 2) the I/O performance may vary in the cluster of SciDB nodes. Existing work have also mentioned the complexity of accurately modeling query time [3].

4 DESIGN OVERVIEW

In this section, we present the general structure of the DyVer system designed for SciDB array database, which focuses on improving the storage and query efficiency of dynamic versioning data. Figure 4 shows the fundamental architecture of DyVer.

SciDB uses a shared-nothing architecture and is deployed on a cluster of servers. Each physical server hosts a SciDB instance and is responsible for data processing and local storage. When receiving a request from the user, where the request can either be a data update request or a query for a specific version, the query processor will generate an execution plan for the submitted request on all instances. As each SciDB instance follows the same logic to run the execution plan [18], in the following we mainly discuss the query execution on a single node.

DyVer is integrated into the query processor of SciDB. As shown in Figure 4, when receiving a data update request, a new version is created and inserted into the version graph. DyVer takes three steps to generate the store a new version, including *delta encoding*, *chunking*, and *compression*. The delta encoding step decides how to store the version, i.e., to materialize it or store a delta of it. We develop an efficient optimal storage layout technique for DyVer to encode the versions. The chunking step divides the version into multiple chunks and distributes them across nodes. Finally, the compression stage compresses version chunks and stores them on local disks. When receiving a version query request, DyVer also takes three steps, including *delta decoding* which gives how to recreate the queried version, *chunk selection* to locate the required chunks for the recreation and *decompression* which decompresses the deltas and returns the requested version to the user.

In DyVer, we adopt the standard chunking and compressing techniques as in SciDB and focus on optimizing version encoding/decoding to optimize the storage and query efficiency of versioning data. The key idea of DyVer is to balance the trade off between the storage size and query time considering the dynamic addition of versions. DyVer incorporates *Optimal Storage Layout* and *Version Segment* to achieve this goal. We proposed the optimal storage layout optimization to find a good version storage plan efficiently. However, as mentioned earlier, the overhead of finding good storage layouts increases with the increase of number of versions. To address this problem and improve scalability of DyVer in case of version additions, we further proposed to use version segment to limit the maximum overhead of our storage layout optimization. In the next two sections, we will describe the details of the two components of DyVer.

5 OPTIMAL STORAGE LAYOUT

To optimize the storage layout for a given set of data versions, there are two main challenges. First, the solution space for the storage layout optimization is large. Given N versions, there can be N^N different layout configurations. For a large-scale scientific application with multiple collaborators, N can easily grow to thousands and thus make it impossible to search for the best storage layout in a reasonable time. Second, there is no clear correlation between the storage size and query time optimization goals, thus making it difficult to reduce the problem complexity.

To overcome the above challenges, DyVer takes a two-stage method to reduce optimization overhead. First, it builds the storage layout with minimum storage size. Then, it uses the remaining storage capacity to modify some versions' layout, which will reduce the average version query time. In the following parts, we will introduce details about these two steps.

5.1 Minimizing Storage Size

In order to obtain the layout with minimizing storage size, DyVer will construct a storage matrix M from versions at first. In the storage matrix, the value $M(i, j)$ off the diagonal indicates the size of the delta $\Delta_{i,j}$ between two versions V_i and V_j. By transforming the storage information among versions into a graph represented as a matrix, we can use graph-related algorithms to construct a layout scheme with the minimum storage size. Note that this matrix can be symmetric or asymmetric, depending on the specific differencing mechanism used for computing deltas [3]. For undirected graphs, we can use Prim's algorithm to find the minimum spanning tree (MST); for directed graphs, we can use Chu-Liu/Edmonds algorithm to find the minimum cost arborescence (MCA). However, the above commonly used algorithms need to specify the root node, so finding the minimum storage layout of n versions requires n calculations in assigning the root. In order to reduce the system burden, we introduce a dummy node by making reference to [3]'s method. The dummy node can reach any other node (this is a one-way edge in the directed graph), and the edge weight is the storage size of the corresponding materialized version. Then, it only needs to set the dummy node as the root. The algorithm mentioned above can

Algorithm 1 Storage Layout Minimization Algorithm

Input: $\mathcal{V}$: the collection of versions (the index starts from 1); n: The number of versions;
Output: L: The minimizing storage layout;
1: **Add Dummy Node** V_0 to $\mathcal{G}$;
2: **Create** Storage Matrix M
3: **for each** $V_i \in \mathcal{V}$ **do**
4: **for each** $V_j \in \mathcal{V}$ **do**
5: $M(i, j) \leftarrow \Delta_{i,j}$;
6: **end for**
7: **end for**
8: **for** $i \leftarrow 0, n+1$ **do**
9: $M(i, i) \leftarrow \infty$;
10: $M(0, i) \leftarrow \infty$;
11: **end for**
12: $L \leftarrow$;
13: $MCA \leftarrow MinimalCostArborescence(M)$;
14: $MCA_{improve} \leftarrow BFS(MCA)$
15: **for each** $E(i, j) \in MCA_{improve}$ **do**
16: $L \leftarrow L \cup \Delta_{i,j}$;
17: **end for**
18: **return** L;

be executed once to obtain the layout scheme with the minimum storage size. Note that this layout still needs to be improved. Since we want to avoid too many long backward delta chains, the storage graph should be as "short" as possible. Therefore, the system runs a breadth-first search (BFS) from the dummy node. Some versions can be linked to lower-level versions to reduce the query time without increasing the storage size. After the above steps, the system will obtain an optimized storage layout. Once obtaining the layout scheme, DyVer will calculate the available space, which will be used to further optimize the version query performance for versions.

Algorithm 1 describes how to establish the minimum storage size layout in the directed graph. Note that for undirected graphs, the algorithm 1 does not need to change much. To begin with, we add a dummy node V_0 into the graph and construct a $(n+1) \times (n+1)$ storage matrix M (line 1-2). The value $M(i, j)(i \neq j, i \neq 0)$ indicates the storage size of the $\Delta_{i,j}$ between two versions V_i and V_j (line 3-5). Then we set the value $M(i, i)$ on the diagonal of the matrix and $M(0, i)$ $(0 \leq i \leq n+1)$ to ∞, indicating that all self-loops are removed from the graph (line 6-9). Finally, we use Chu-Liu/Edmonds algorithm [8] to construct the MCA which will be modified through BFS later. The MCA presents the storage strategy for every version, that is, either materializing it or deltaing it against another version. (line 10-14).

Complexity. The storage matrix can be constructed in $O(n^2)$ time. Directed graphs and undirected graphs use different algorithms in finding the minimum storage layout. For directed graphs, we use Chu-Liu/Edmonds algorithm. It can be found with a running time of $O(nlogn + m)$, where m denotes the number of edges. For undirected graphs, we use Prim's algorithm, whose time complexity is $O(n^2)$. In addition, the MCA will be modified in $O(n^2)$ time via BFS. Overall, the complexity of algorithm 1 is $O(n^2)$.

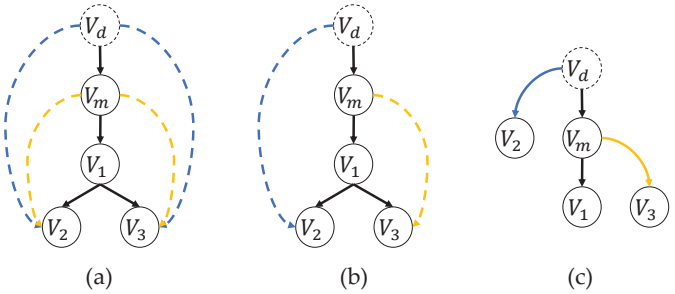


Figure 5: Illustration of the version shortcut. (a) The four versions of the retrieval path are indicated by solid black lines, where V_d is the dummy node and V_m is the materialized version. The dashed lines are the version shortcuts that can be established. The two types of shortcuts are separated into blue and yellow. Note that the V_1's shortcut is omitted. (b) Shortcuts for V_2 and V_3 will be established, where V_2 will be materialized and V_3 will be modified as a delta against V_m. (c) DyVer completes the establishment of shortcuts and removes the redundant delta. This figure presents the new retrieval path for four versions.

5.2 Version Shortcut

When there still has remaining storage capacity, we leverage *Version Shortcut* to speed up the version query process. This technique will only make local adjustments to the existing storage layout and will not overburden the storage management system.

After establishing layout with minimum storage size, any version has exactly one retrieval path. As a result, other than the fully materialized version V_m, any given version V_g can be recreated as follows:

$$V_g = V_m + \sum_{i=g+1}^{m} \Delta_{i,i-1} \quad (5)$$

The distance from the materialized version limits the access time to rebuild the required version [22]. The querying time for some versions could be improved by deltaing them against other versions or materializing them, which might increase the storage size. Soroush and Balazinska [19] implemented *skip links* to improve the version query performance for some versions by skipping over multiple versions in one step. However, the technique is designed on version chains. It does not include query time as a metric for reference when establishing skip links. Bhattacherjee et al. [3] proposed a strategy for combining shortest path trees (SPT) into the minimum storage layout. However, SPT tends to materialize all versions because of lower query time. Only a few versions can be materialized when the storage capacity is limited.

In DyVer, we propose a technique called *Version Shortcut* to improve the version querying performance for versions. Version shortcut is an extension to Skip Link. Compared with the previous studies [3, 18, 19], it not only takes the backtracking time into consideration, but also ties interspersing materialized versions with the Skip Link. It is implemented upon the minimum storage layout, which is constructed in the first step.

Shortcuts can be made with numerous alternative versions for any version V_i. However, exploring all combinations is computationally intensive. Besides, many shortcuts do not significantly improve the version query performance. DyVer employs a greedy strategy by concentrating on two shortcuts. They are established between V_i and the dummy node V_d in one case, and between V_i and the materialized node V_m in the other. The first one indicates to materialize V_i; the second one is to greatly shorten the retrieval path of V_i (except for being materialized). An abstract example of version shortcuts is shown in Figure 5. A materialized version can be queried faster than the non-materialized ones which have to be recreated via other versions. Therefore, DyVer will prioritize materializing some versions. By doing this, the average distance between the required version and materialized version can be reduced. In other words, the average version query performance can be improved. When there is no enough capacity to materialize any version, DyVer will try to establish shortcuts between materialized versions and non-materialized one. This is because storing delta from aother version takes up less storage size than storing version in its entirety [18]. This method can make better use of the storage capacity while continuing to optimize the average version query performance.

The query time of version V_i, as well as all versions containing V_i on the retrieval path, are affected by changing the storage form of version V_i. The subtree rooted at V_i is a collection of these retrieval paths. We establish shortcuts to exchange for a bigger query time reduction at the expense of increasing storage sizes. We define the benefit of a subtree as the ratio of its increased storage size to its reduced query time. Specifically, S_{Inc} represents the increase in storage size of V_i, T_{Dec} represents the decrease in query time of V_i. The number of nodes in the subtree rooted at V_i is represented by n, and the benefit is represented by ρ. The calculation formula of ρ is as follows:

$$\rho = \frac{T_{Dec} \times n}{S_{Inc}} \times 100\% \tag{6}$$

The equation 6 calculates the benefits brought by Shortcuts to each subtree. DyVer sorts them in descending order and establishes the corresponding shortcut in turn.

Algorithm 2 describes how version shortcuts are established to further optimize the storage layout. DyVer calls the algorithm twice. For the first time, it establishes shortcuts to dummy node, that is, materializing versions. For the second time, it establishes shortcuts to materialized versions, which skips over multiple versions in one step. DyVer will establish a shortcut for each non-materialized version V_i and use the equation 6 to calculate its benefit ρ. Each shortcut will be kept in a priority queue PQ, which is sorted in descending order of ρ (line 2-5). Note that the shortcut must contain the version information. Then, DyVer will take each shortcut from PQ and establish it until the storage capacity constraint is exceeded. Having the shortcut added to layout $\mathcal{G}$, DyVer will remove the version's initial storage form. Because this delta is redundant (line 6-13). After the above steps, DyVer will get the optimized storage by shortcuts.

Complexity. Before running the algorithm 2, DyVer will use the DFS to traverse the entire layout in advance. This step will record the ancestor node for each version and the number of nodes in the subtree rooted at it. The time complexity of this process is $O(V+E)$, where V is the number of versions and E is the number of edges. Then, algorithm 2 only needs $O(V)$ to traverse all versions and establish their shortcuts. We use the priority queue to store these shortcuts, so the time complexity of it is $O(VlogV)$. After that, DyVer takes out shortcuts from the priority queue and adds them one by one. The time complexity of this process is $O(V)$. Overall, the complexity of algorithm 2 is $O(VlogV)$.

Algorithm 2 Version Shortcut Creation Algorithm

Input: $\mathcal{G}(\mathcal{V}, \mathcal{E})$: The minimum storage size layout for versions, where $\mathcal{V}$ is the collection of versions and $\mathcal{E}$ denotes the derivation relationship among versions; S: The current storage size for versions; MS: The storage capacity constraint;
Output: $\mathcal{G}$: $\mathcal{G}$ improved by Version Shortcut;

```
1:  PQ ← ∅;
2:  for each non-materialized V_i ∈ V do
3:      Create shortcut for V_i;
4:      compute ρ by equation 6;
5:      PQ.enqueue(shortcut);
6:  end for
7:  while PQ is not empty do
8:      shortcut ← PQ.pop();
9:      compute Δ from shortcut;
10:     if S + Δ > MS then
11:         break;
12:     end if
13:     S ← S + Δ
14:     Add shortcut to G;
15:     Delete E(p(V_i), V_i) from G;
16: end while
17: return G;
```

6 VERSION ADDITION

The full version set is unknown in advance. While optimizing version encoding, we ought to consider how to adjust the technique to take into account the dynamic addition of versions. Especially, dynamic version handling requires two main features: (1) the ability to quickly insert new versions and find a good layout, and (2) to minimize the impact on existing versions.

An straightforward approach would be to compute the deltas for all version pairings and run the layout algorithm whenever new versions are inserted. In this way, we can find a good storage layout by taking advantage of the similarities between every pair of versions, especially those who are far away in time. However, as the number of versions increases, it will be extremely expensive and infeasible to compute all deltas. In addition, when a new version is inserted, the system must modify the layout of all versions in the database. Obviously, this approach dose not satisfy the requirements of dynamic version addition.

In DyVer, we implement a container named *version segment*. This design is tightly related to the two features. First, DyVer uses it to quickly find a good layout for new versions without impacting much the existing storage layout. Second, it can limit the time needed for calculating deltas and thus greatly reduce version insertion

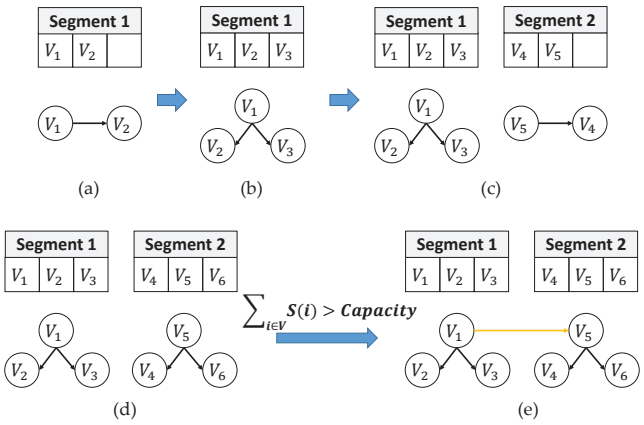


Figure 6: Illustration of the version segment containing 3 versions. (a) Initially, Segment 1 has two versions. (b) New version is added in Segment 1. DyVer checks the number of versions in Segment 1 and runs the version encoding algorithm on these versions. (c) Segment 1 is full. DyVer creates a new segment to contain new versions. The version encoding algorithm is only run on the Segment 2, while the versions in Segment 1 maintains the layout. (d) The total storage size exceeds the available capacity. (e) DyVer stores V_5 as $\Delta_{1,5}$ to minimize the storage size.

overhead. Specifically, the version segment contains N versions. It is the unit to determine the optimal storage layout. It utilizes the version update log file to define the size. When the number of versions in it is equal to N, DyVer will create a new segment to contain others. Additionally, the algorithm will no longer be run on the versions contained in the filled-up segment. Figure 6(a-c) shows the example of the version segment.

The number of versions N contained in the version segment is a critical factor for its operation. Here are some details about how the system determines it. Firstly, DyVer obtains the frequency of version updates by checking the version update log. Since the time to computing deltas is significantly longer than running the algorithm [3], our system only estimates how many deltas can be computing within the update gap. Then, DyVer chooses several version pairs at random, collects the build time for each delta, and calculates the average T_{delta}. Since a delta corresponds to an element of the storage matrix, equation 7 can be used to compute the number of versions N.

$$N = \sqrt{\frac{T_{gap}}{T_{delta}}} \tag{7}$$

T_{gap} in equation 7 is the time gap of the version addition obtained from the update log. Note that the number of versions that each segment contains is alterable. In practice, when the time to computing deltas of N versions exceeds the update gap, DyVer terminates the current segment from adding new versions and creates a new one that can contain $N-1$ versions. By doing this, the possibility of timeout is reduced.

Note that when segments are small (e.g. when the update gap is short), many versions will be materialized. To ensure that the storage size does not exceed the capacity constraint, DyVer stores these materialized versions in a more compact manner. Specifically, for the materialized version V_i in the latest segment, DyVer would find the materialized version V_j which has the smallest difference with V_i. Then, V_i will be stored as $\Delta_{j,i}$. This process can be completed in linear time. Figure 6(d-e) provides a concrete example.

7 EVALUATION

This section presents our experimental results on evaluating the proposed system. Overall, we conduct three major sets of experiments. Specifically, we first use real-world datasets to study the optimization results of our system in comparison with the state-of-the-art approaches in Section 7.2. Further, we experimentally assess the impact of different parameters with sensitivity studies in Section 7.3. Finally, we conduct ablation studies on DyVer in Section 7.4.

7.1 Experimental setup

7.1.1 Datasets. In order to measure the storage and recreation cost optimization of DyVer, we apply our model to 3 datasets from the National Snow and Ice Data Center (NSIDC [15]), namely ILATM2 [20], SPL2SMP [6], and NISE [4].

ILATM2 contains resampled and smoothed elevation measurements of Arctic and Antarctic sea ice, and Greenland, Antarctic Peninsula, and West Antarctic region land ice surface acquired by the NASA Airborne Topographic Mapper (ATM) instrumentation. The data files are in CSV format. Each data contains a total of 11 attributes such as time, latitude and longitude.

SPL2SMP records the surface soil moisture presented on the global 36 km EASE-Grid 2.0 projection. Data are in HDF5 format. All data element arrays are one-dimensional, with the exception of landcover_class and landcover_class_fraction, which are two-dimensional arrays.

NISE provides daily, global maps of sea ice concentrations and snow extent. Daily data are provided in a single HDF-EOS file containing two data fields, extent and age. Extent and age values are stored as binary arrays of unsigned 1-byte (8-bit) data ranging in value from 0 to 255.

The frequency of updates for the above datasets was 1 minute, 49 minutes, and 24 hours, respectively. For each dataset we have selected 1000 files (each file is treated as one version). In addition, in SciDB, the chunk size can be set by the user. Therefore, we arbitrarily set the maximum file size for each dataset to the chunk size of the corresponding one (less than the system bus width).

7.1.2 Comparisons. We assess the effectiveness and efficiency of DyVer by comparing the following six approaches.

- **VC**. Version Chaining (VC) stores a single version chain. It emulates the backward delta encoding method in SciDB, i.e., materializing the latest version as well as storing the delta between each pair of consecutive versions.
- **VC+SL**. Excessively long chains can largely reduce the efficiency of VC queries. Therefore, Soroush and Balazinska [19] proposes Skip Link for optimization. It can fetch old versions by exploiting skip links instead of recreating all the

consecutive versions between them and the most recent one. We call this method VC+SL.

- **VG**. The Version Graph (VG) differs from the VC in that it stores versions based on their original derivative relationships. Specifically, we use GTgraph [1] to generate a graph which implies the derivative relationships. All versions are stored as deltas against their corresponding parent versions, except for the first version that is materialized. As some versions have multiple parents, only the relationship with the smallest delta is kept in our experiments.
- **SOL**. Seering et al. [18] proposes the Space Optimal Layout (SOL) to efficiently determine the most compact representation of a collection of versions using the Materialization Matrix. In order to constrain the size of materialization matrix and avoid very long delta chains, it sets a fixed-size batch (10-100) to adjust the version layout. In the experiment, we set the batch size to 50.
- **LMG**. Considering the case where there is a storage capacity constraint, Bhattacherjee et al. [3] proposes the Local Move Greedy (LMG) algorithm for static, off-line versions. The scheme combines MCA and SPT to optimize the version query without exceeding the storage capacity constraint. In our experiments, we consider the recreation cost as the average query time of a version. Besides, it is designed for static, off-line versions. In the experiment, we regard it as a global layout for all versions.
- **DyVer**. It is our proposed framework, which is a version-graph based database management system. It manages versions of data with full consideration of the storage and recreation trade-off. In addition, the average sizes of the segment set by DyVer in the three version sets were experimentally tested, which are 34, 98, and 923, respectively.

In the next experiment, we set three storage constraints, namely *tight*, *moderate*, and *loose*. The tight constraint is 1.1 times the storage size under the VC method, while the loose constraint is 0.8 times the storage size under all materialized versions, and the moderate constraint is the average of the two above-mentioned constraints. We set the constraint to tight by default. In addition, VC+SL has a short run time. The arrangement of versions impact the construction of skip links. To ensure fairness, we will shuffle the versions several times and run it every time. This is guaranteed that VC+SL has the same run time as DyVer.

7.1.3 Configuration. As for simulations, we leverage the approaches above to manage the storage of data in datasets. We manipulate Xdelta3 [13], an open-source tool for delta compression, to compute deltas between versions. We keep on calculating the storage size and version query time for each dataset while executing. We repeat the simulation for 150 times and then calculate the average execution results for evaluations on every dataset. All experiments are performed on a PC with 1008 GB RAM, Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz, and the operating system is Ubuntu 18.04.5 LTS.

7.2 Simulation results

Fig. 7(a) and 7(b) presents the average version query time and storage size results of the compared algorithms under the tight constraint. The storage size is normalized to the capacity constraint. Fig. 7(c) presents the average time taken to compute required deltas for each algorithm in ILATM2. As other datasets have similar results, we present only one of them here as representative.

In terms of the average version query time, DyVer significantly outperformed the other approaches. Compared to VC, VC+SL, and VG, DyVer reduces the average version query time by over 95%; compared to SOL, DyVer reduces it by 76-89%; and compared to LMG, it reduces it by 3-87%. In terms of the storage size, only the SOL exceeds the constraint in NISE. And in terms of the time to compute the required deltas, VC and VG take the least time, VC+SL, SOL, and DyVer follow, and LMG is the highest. Next, we will compare the above approaches to DyVer in these three aspects.

First, VC stores versions as a single chain. Since only the latest version is materialized, the query time for the other versions is proportional to their reconstruction distance. As the number of versions continues to increase, their average query time rises even further. DyVer materializes some of the versions based on the storage constraint. It balances the reconstruction distances of each version and improves the version query time. Secondly, VC+SL is an improvement to VL using Skip Link. VC+SL takes much longer to compute deltas than VC, because it needs the information on several non-contiguous deltas. However, the improvement brought from Skip Link is limited. This approach only refers to the storage size metric when constructing the Link. When smaller non-contiguous increments cannot be found in a single chain, the system cannot build the Link. In contrast, the shortcut constructed by DyVer references both storage size and version query time. By increasing the storage size, more potential links can be constructed. Then, unlike the previous two methods, VG preserves the derived relationship between versions and stores the corresponding deltas. The situation remains that the chain of backward deltas is too long. DyVer no longer stores versions according to the derived relationship, but instead proposes a more compact layout. In addition, SOL proposes the idea of storing versions by batch. By setting a fixed size batch and having the versions within the batch laid out compactly as a minimum spanning tree. Batch is similar to DyVer's segment. However, the batch under this approach ignores the features of the dataset and is set to a fixed size. Since at least one version is materialized within each batch, a too-small batch will result in too many versions being materialized. Thus, SOL's storage size exceeds the constraint on NISE. Furthermore, it only materializes the root version of each minimum spanning tree. DyVer goes one step further to discover the remaining storage constraint and continues to materialize others, which greatly shortens the average version query time. Finally, LMG, a global adjustment approach, has the same optimization goals as DyVer. While both have similar average version query time and storage size on ILAMT2, LMG needs to calculate much more deltas than either of the other schemes. Thus, we can see in Fig. 7(c) that LMG takes 17472s to compute the deltas before running the algorithm. By processing versions based on segments, DyVer not only reduces the burden on the system to compute the deltas, but also achieves a better result than the global adjustment algorithm.

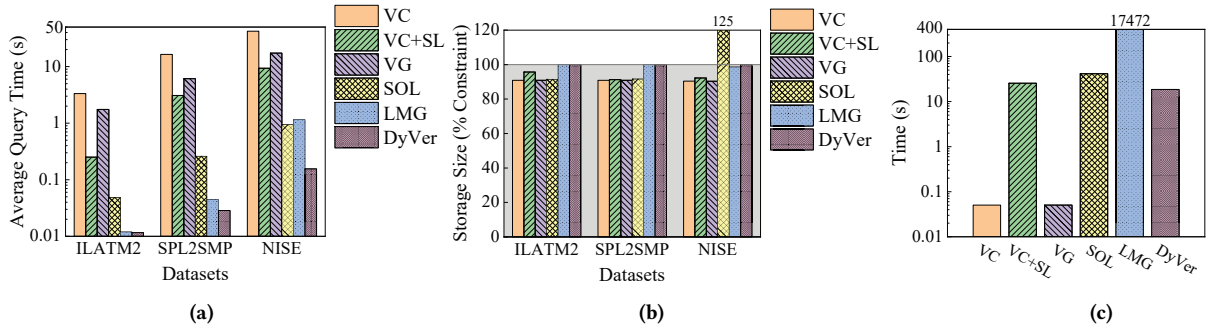


Figure 7: Comparison of the (a) average version query time, (b) storage size in three datasets, and (c) average time taken to compute deltas in ILATM2 when using different approaches under the tight constraint.

7.3 Sensitivity Studies

We also conduct sensitivity studies to exercise our system and isolate key factors, such as storage constraint and segment size, that affect DyVer's performance. First, we present the average version query time of LMG and DyVer under different constraints. Then, we show the impact of different segment sizes on the average version query time and storage size.

7.3.1 Changing Storage Capacity Constraint. We compare DyVer with LMG at different storage capacity constraints to illustrate the effectiveness of DyVer and to further understand its relevance to the constraints. Note that the remaining approaches are insensitive to the storage capacity constraints. Therefore, we omit the comparison of them. Fig. 8 presents the average version query times for LMG and DyVer under the three datasets with different capacity constraints.

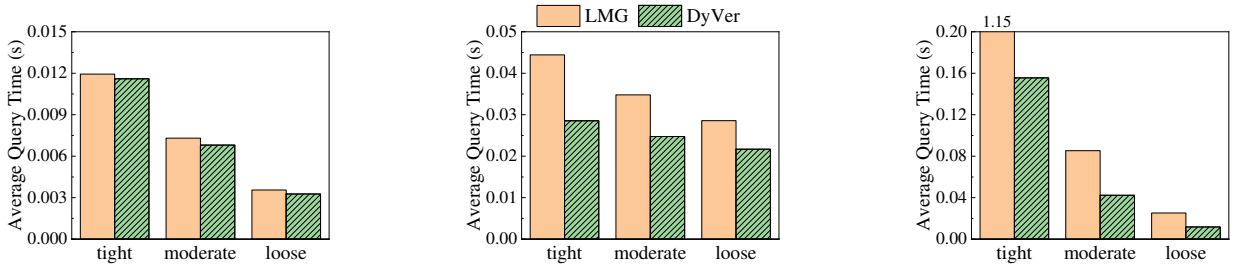
The average version query time for both methods decreases as the constraint gradually increases. However, the version query efficiency of DyVer is consistently better than that of LMG under different constraints. This advantage does not change drastically although the constraint changes. At ILATM2, the average version query time of DyVer is lower than that of LMG by 3%-8%; at SPL2SMP, DyVer is lower than that of LMG by 24%-36%; and at NISE, DyVer is lower than that of LMG by 50%-87%. Although LMG uses SPT to optimize the MCA, due to the larger version size, there are still relatively long backward delta chains. In contrast, after DyVer materializing the most cost-effective version, it still continues to optimize the version query efficiency of some versions by adding shortcuts.

Fig. 9 presents the distribution of version query times for each version at SPL2SMP when using LMG and DyVer under the tight constraint. Although these two methods share a similar median (only 8% difference), more versions require more than 0.1s to be reconstructed when using LMG. Although DyVer has some versions that require more than 0.2s to be reconstructed, nearly 99% of the versions can be reconstructed in less than 0.1s. Thus, the average version query time of DyVer is less than that of the LMG. It also shows that DyVer is more sensitive to storage constraint and can strike a better balance between storage size and version query time.

7.3.2 Changing Segment Size. In this section, we compare different sizes of segments to illustrate the effect of segment size on DyVer under the tight constraint. Since there are 1000 versions of each dataset, we set the number of versions that can be accommodated in each of the three segments to 10, 100, and 1000, respectively. Note that the factor that affects the size of a segment is the update gap of the dataset. Here we use different segments to show the effect of different update gaps on the results. Fig.10(a) presents the average query time for the versions under the three segment settings. In addition, we use the NISE dataset as representative and present the storage size and the time to compute the required deltas in Fig.10(b) and (c), respectively. Specially, the storage size consists of two parts: the solid colour represents the total size of the delta and the grid represents the total size of the materialized version, which together constitute the total storage size. We omit the algorithm's execution time because it is significantly less than the time required to compute the deltas. Even with a segment size of 1000, the algorithm's running time is less than 5 minutes, which is much shorter than the time required to compute the deltas.

First, the larger the segment grows, the shorter its average version query time is. The increase in segment size from 10 to 1000 reduces the average version query time for the three datasets by 39%, 34%, and 61%, respectively. This is because the larger the segment is, the more deltas can be obtained. Therefore, a more optimal layout of versions can be made.

We use NISE as an example to analyze the difference in the storage size between different segments. The storage cost of materialized versions in this dataset is much higher than that of deltas. Therefore, in Fig.10(b), under the tight storage budgets, only a few versions are materialized while most versions are stored in delta form. When the segment size is set to 10 and 100, there is still a portion of the storage constraint left unused. When the size goes up to 1000, its materialized version has the largest total storage size. Since the optimal storage layout is implemented in terms of segments, a setting with small segment size can result in too many materialized versions. When the storage size exceeds the constraint, DyVer will store these materialized versions in a more compact layout to reduce the storage size. Thus, smaller segments will result in more originally materialized versions being stored as compact deltas. Although the storage size is reduced, this leads to an increase in the average version query time (as shown in Fig.10(a)).



(a) ILATM2 **(b) SPL2SMP** **(c) NISE**

Figure 8: Comparison of the average version query time in three datasets when using LMG and DyVer under the different constraint.

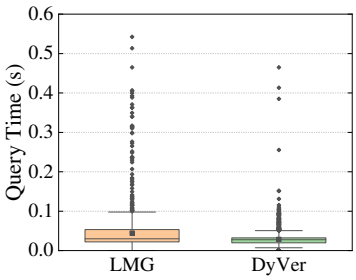


Figure 9: Version query time distribution for versions in SPL2SMP when using LMG and DyVer under the tight constraint.

Finally, we use NISE as an example to illustrate the difference in the average time taken to compute delats in different segments. As the number of segments increases, the number of deltas required to be built also increases significantly. The average time to compute deltas increases nearly 10000 times when the segment size increases from 10 to 1000. Thus, although a larger segment can lower the average version query time, it is costly in terms of time spent on deltas computing.

In general, a smaller segment can cope with workloads where versions are added frequently, as it only needs to compute a small number of deltas to determine the optimal storage layout, which reduces the burden on the system. A larger segment is better suited to workloads with longer update gaps, as it can have more time to compute more deltas for performing the version layout algorithm. This will achieve a dynamic balance between storage size and average version query efficiency.

7.4 Ablation Studies

In this section, we study the improvements that each individual technique of DyVer brings to version storage management, namely Minimizing Storage Size (MSS) and Version Shortcut (VS). MSS is DyVer using MCA alone, while VS is DyVer with Version Shortcut applied to VG. The evaluation results are shown in 11.

For the average version query time, using MSS and VS alone leads to an increase in average version query time of 268-1449% and 251-693%, respectively. This illustrates the effectiveness of Minimum Storage Size and Version Shortuct. MSS shows a lower average version query time in the ILATM2 dataset, while it does not perform as well as VS in the other two datasets because of different segment sizes. MSS is optimized for the minimal storage size. However, the more versions to be processed, the more likely it will be that some versions have long recreating distances. In addition, MSS does not take the storage constraint into account. These low query efficiency versions can often be greatly improved by increasing the storage size. VS focuses on it and optimizes the version query efficiency. The benefit increases as the number of versions processed increases. However, VS is only simply applied to VG, and there are still some versions that have long retrieval distances. The average version query time on the ILATM2 dataset is therefore higher than that of MSS.

In terms of the storage size, MSS has ignored the storage constraint by seeking the lowest storage size. The storage size of it is therefore lower than others'. VS and DyVer balance storage constraint and version query efficiency, so both of them have higher storage size than MSS but within the constraint.

From the above analysis, we know that DyVer was able to significantly improve the version query efficiency of the MSS by utilizing only 7%, 9%, and 12% of the additional storage constraint for the three datasets. This shows that the combination of MSC and VS is a good way to achieve a dynamic balance between storage size and version query efficiency.

8 CONCLUSIONS

In this paper, we presents DyVer, a dynamic version handling framework for SciDB. The key idea of our system is to balance the trade off between the storage costs and version retrieval performance considering dynamically added new versions. DyVer incorporates Optimal Storage Layout and Version Segment to optimize the average retrieval efficiency of versions. The evaluation results show that our system can achieve much more effective and efficient version storage layouts compared to the state-of-art data versioning techniques under the same storage budget.

DyVer does not have any specific assumptions regarding the workload patterns and storage devices. It relies on the underlying filesystem to do the I/O scheduling. For future work, we plan to study how hardware features such as SSD read/write interferences impact the performance of DyVer and propose a hardware-aware DyVer to further improve version handling effectiveness.

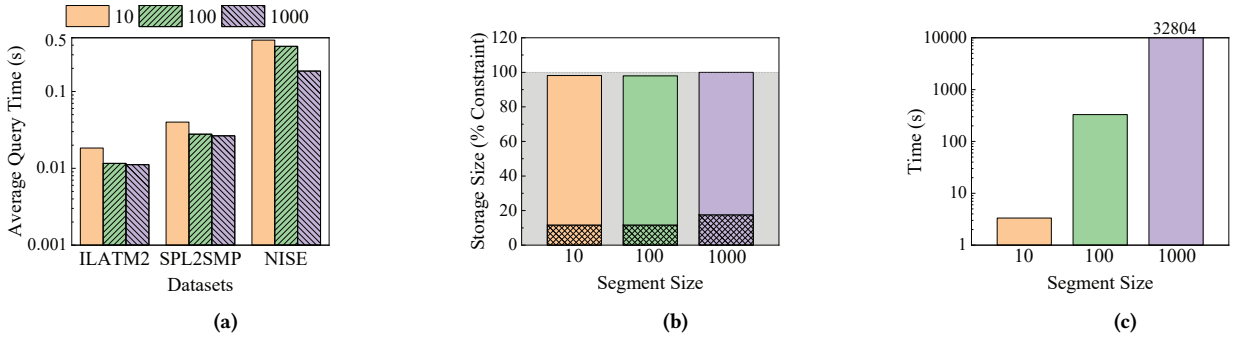


Figure 10: Comparison of the (a) average version query time in three datasets, (b) storage size and (c) the average time taken to compute the required delta in NISE when using different segment size under the tight constraint. The storage size in (b) consists of two parts: the solid colour represents the total size of the delta and the grid represents the total size of the materialized versions.

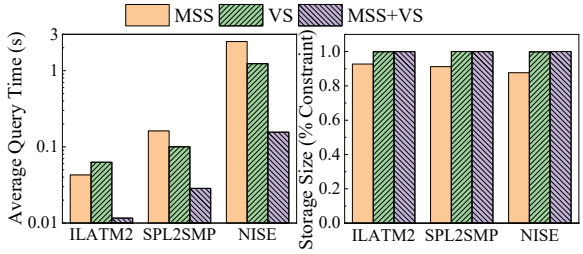


Figure 11: Comparison of the average version query time and storage size when using MSS, VS and combining of them.

ACKNOWLEDGEMENT

This work is supported by the National Natural Science Foundation of China 62172282, Guangdong Natural Science Foundation 2022A1515010122, Shenzhen Science and Technology Foundation (Grant. RCYX20221008092908029, JCYJ20210324093212034 and 20200814105901001) and Guangdong Province Undergraduate University Quality Engineering Project (Shenzhen University Academic Affairs [2022] No. 7).

REFERENCES

[1] David A Bader and Kamesh Madduri. 2006. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February* 38 (2006).
[2] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The multidimensional database system RasDaMan. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 575–577.
[3] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *Proceedings of the VLDB Endowment*. NIH Public Access, 1346.
[4] M. J. Brodzik and J. S. Stewart. 2016. Near-Real-Time SSM/I-SSMIS EASE-Grid Daily Global Ice Concentration and Snow Extent, Version 5. (2016). https://doi.org/10.5067/3KB2JPLFPK3R
[5] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 963–968.
[6] S. Chan, R. Bindlish, P. E. O'Neill, E. G. Njoku, T. Jackson, A. Colliander, F. Chen, M. Burgin, S. Dunbar, J. R. Piepmeier, S. Yueh, D. Entekhabi, M. Cosh, T. Caldwell, J. Walker, A. Berg, T. Rowlandson, A. Pacheco, H. McNairn, M. Thibeault, J. Martinez-Fernandez, A. González-Zamora, D. Bosch, P. Starks, D. Goodrich, J. Prueger, M. Palecki, E. E. Small, M. Zreda, J. Calvet, W. T. Crow, and Y. Kerr. 2016. Assessment of the SMAP passive soil moisture product. In *IEEE Transactions on Geoscience and Remote Sensing*. 6046–6048.
[7] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. 2009. A Demonstration of SciDB: A Science-Oriented DBMS. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1534–1537.
[8] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6, 2 (1986), 109–122.
[9] Silu Huang, Liqi Xu, Jialin Liu, Aaron J Elmore, and Aditya Parameswaran. 2020. Orpheus db: bolt-on versioning for relational databases. *The VLDB Journal* 29, 1 (2020), 509–538.
[10] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 1695–1698.
[11] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 997–1008.
[12] Lawrence Livermore National Laboratory. 2022. Program for Climate Model Diagnosis and Intercomparison. https://pcmdi.llnl.gov/. (2022).
[13] Joshua MacDonald. 2017. Xdelta3. https://github.com/jmacd/xdelta. (2017).
[14] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. In *Proceedings of the VLDB Endowment*. NIH Public Access, 624.
[15] NASA. 2022. National Snow and Ice Data Center. https://nsidc.org/. (2022).
[16] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The tiledb array data storage manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360.
[17] World Climate Research Programme. 2023. WCRP Coupled Model Intercomparison Project (CMIP). https://www.wcrp-climate.org/wgcm-cmip. (2023).
[18] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient versioning for scientific array databases. In *2012 IEEE 28th International Conference on Data Engineering (ICDE)*. IEEE, 1013–1024.
[19] Emad Soroush and Magdalena Balazinska. 2013. Time travel in a scientific array database. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 98–109.
[20] M. Studinger. 2020. IceBridge ATM L2 Icessn Elevation, Slope, and Roughness, Version 2. NASA National Snow and Ice Data Center Distributed Active Archive Center.
[21] TileDB Team. 2018. TileDB Introduction. https://tiledb-inc-tiledb.readthedocs-hosted.com/en/1.6.3/introduction.html. (2018).
[22] Haoyuan Xing, Sofoklis Floratos, Spyros Blanas, Suren Byna, M Prabhat, Kesheng Wu, and Paul Brown. 2018. ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 977–988.
[23] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. 2010. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *2010 IEEE international symposium on parallel & distributed processing (IPDPS)*. 1–12.
[24] Ramon Antonio Rodriges Zalipynis. 2018. Chronosdb: distributed, file based, geospatial array dbms. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1247–1261.