Lawrence Berkeley National Laboratory
LBL Publications

Title

ArrayBridge: Interweaving declarative array processing with high-performance computing

Permalink

https://escholarship.org/uc/item/6jg8h6gj

Journal

CoRR, abs/1702.08327

Authors

Xing, Haoyuan
Floratos, Sofoklis
Blanas, Spyros
et al.

Publication Date

2017-02-27

eScholarship.org Powered by the California Digital Library
University of California

ArrayBridge: Interweaving declarative array processing with high-performance computing

Haoyuan Xing*, Sofoklis Floratos*, Spyros Blanas*,
Suren Byna†, Prabhat†, Kesheng Wu†, Paul Brown§

* The Ohio State University
{xing.136, floratos.1, blanas.2}@osu.edu

† Lawrence Berkeley National Laboratory
{sbyna, prabhat, kwu}@lbl.gov

§ Paradigm4, Inc.
pbrown@paradigm4.com

arXiv:1702.08327v1 [cs.DB] 27 Feb 2017

ABSTRACT

Scientists are increasingly turning to datacenter-scale computers to produce and analyze massive arrays. Despite decades of database research that extols the virtues of declarative query processing, scientists still write, debug and parallelize imperative HPC kernels even for the most mundane queries. This impedance mismatch has been partly attributed to the cumbersome data loading process; in response, the database community has proposed *in situ* mechanisms to access data in scientific file formats. Scientists, however, desire more than a passive access method that reads arrays from files.

This paper describes ArrayBridge, a bi-directional array view mechanism for scientific file formats, that aims to make declarative array manipulations interoperable with imperative file-centric analyses. Our prototype implementation of ArrayBridge uses HDF5 as the underlying array storage library and seamlessly integrates into the SciDB open-source array database system. In addition to fast querying over external array objects, ArrayBridge produces arrays in the HDF5 file format just as easily as it can read from it. ArrayBridge also supports time travel queries from imperative kernels through the unmodified HDF5 API, and automatically deduplicates between array versions for space efficiency. Our extensive performance evaluation in NERSC, a large-scale scientific computing facility, shows that ArrayBridge exhibits statistically indistinguishable performance and I/O scalability to the native SciDB storage engine.

1. INTRODUCTION

Scientists are increasingly turning to datacenter-scale computers to understand phenomena that would otherwise be impossible or intractable to approach experimentally. Domains as diverse as plasma simulation [8], cosmology [20], and climate modeling [45] use massively parallel simulations that naturally generate many terabytes of array data for further analysis. Many of these arrays are stored in scientific file formats like HDF5 [15, 16] and netCDF [29], which have been deployed in production systems for nearly two decades. Usage data from petaflop-scale computing facilities corroborates that HDF5 and netCDF remain among the most popular building blocks for scientific computing [50].

To quickly sift through data in these formats, scientists commonly write, debug and parallelize custom analysis kernels that manipulate array data in an imperative fashion—even for mundane analyses that could be expressed as a succinct declarative query. Despite the growing appreciation by scientists for software that makes low-level manipulations redundant, systems like SciDB [9] and TileDB [27] are not widely used in large-scale scientific computing facilities. One challenge is that loading array data imposes onerous time and storage space requirements. For SciDB, in particular, we have observed that parallel loading takes hours even for modestly-sized (∼100 GiB) arrays and temporarily uses many times more space than the size of the input for data staging. Array-centric database systems should query HDF5 data in their native file format, instead of anticipating scientists to convert existing HDF5 datasets into more efficient formats.

Scientists, however, desire more than a passive *in situ* access method that reads array datasets from a file. In particular, efficiently serializing arrays in popular file formats is important to make declarative array manipulations interoperable with file-centric processes such as simulation or visualization. The technical challenge is that many array file format libraries have an inherent single-writer, multiple-readers (SWMR) design constraint [30, 40]. This poses an inherent scalability bottleneck because a single writing process cannot match the I/O capabilities of the parallel file system found in modern scientific computing facilities.

In addition, many imperative codes (such as simulations) are iterative and naturally produce versioned array objects. Although time travel queries have been supported in array database systems like SciDB, scientists lack a principled way of accessing past object versions that are stored in an HDF5 file. Saving versions as different objects creates bloated files that duplicate unmodified chunks. The opportunity is to intercept versioning semantics from array database systems to discard redundant array chunks, and in turn conserve I/O bandwidth and storage space. The technical challenge in supporting time travel queries over array file formats is backwards compatibility, as former versions should remain accessible from the version-oblivious HDF5 API.

This paper describes ArrayBridge, a bi-directional array view mechanism that allows a declarative array database system like SciDB to produce, access and update versioned array objects directly in scientific file formats. ArrayBridge aims for greater interoperability between declarative array processing and imperative analysis kernels. Our prototype implementation of ArrayBridge uses HDF5 as the underlying array storage library and seamlessly integrates into the SciDB open-source array database system. The unique features of ArrayBridge are:

1. ArrayBridge implements an array-centric *in situ* processing operator to evaluate AQL/AFL queries directly on external HDF5 objects. By interfacing to the SciDB query

execution engine, ArrayBridge does not make scientists wait for hours for a dataset to be loaded and redimensioned before SciDB answers the first AQL/AFL query.

2. ArrayBridge efficiently materializes array objects in the HDF5 file format by writing in parallel through a *virtual view*. The view partitions the writes into independent streams, which bypasses the single-writer design constraint that limits the I/O scalability of file format libraries. When reading, scientists access a single array object through the view without any modifications to their applications.
3. ArrayBridge permits time travel queries from existing applications that access past versions through the unmodified HDF5 file format API. Under the hood, ArrayBridge deduplicates identical regions between object versions to reduce the storage footprint of the HDF5 file.

Our extensive experimental evaluation in NERSC, a large-scale scientific computing facility, shows that parallel loading of TiB-sized binary array datasets in SciDB can take more than 7 hours and requires many times more staging space than the original input. ArrayBridge completely bypasses the cumbersome data loading process and queries multi-TiB array datasets in the HDF5 format in a few minutes. ArrayBridge exhibits statistically indistinguishable I/O scalability and performance to the native SciDB storage engine. In fact, ArrayBridge processes data at a similar rate to a hand-tuned, imperative C kernel up to 32 nodes; scaling slows in larger clusters due to SciDB processing bottlenecks. In addition, ArrayBridge materializes SciDB objects into the HDF5 format nearly as efficiently as SciDB serializes the database in its proprietary backup-oriented format. Finally, ArrayBridge exposes old versions as HDF5 datasets and transparently eliminates redundancy in versioned objects; this feature is particularly useful for interoperability with version-oblivious imperative analyses.

The remainder of the paper is structured as follows. Section 2 presents details of the SciDB database system and the HDF5 format that are relevant for ArrayBridge. The internals of ArrayBridge are described next: Section 3 gives a system overview, Section 4 presents the scan interface, while Section 5 presents the write interface and the time travel support. We then present the experimental evaluation in Section 6 and summarize the lessons learned when designing the ArrayBridge in Section 7. Finally, Section 8 discusses related work and Section 9 concludes.

2. BACKGROUND

2.1 The SciDB database system

SciDB organizes data into *multi-dimensional arrays*. An array schema defines the number of dimensions the array has (its rank) and the length of each dimension. We use *shape* to collectively refer to the rank and the lengths of the dimensions of an array. Every cell in an array contains one or more attributes (values), similar to a tuple in relational databases. The order and the type of the attributes in each cell are the same, and they are defined in the array schema. Array schema metadata is stored in the SciDB catalog, which is a centralized PostgreSQL database.

An array is stored and processed in the granularity of *chunks* to accelerate I/O performance. SciDB uses the *regular chunking* strategy [37] which partitions a multi-dimensional array into identically-shaped hyper-rectangles. SciDB stores a chunk using Run Length Encoding (RLE); the data is stored as one or more segments, each of which is a tuple $\langle length, same, data \rangle$, where *length* is the number of elements in the segment, *same* represents whether the elements in the segment are all the same. If *same* is true, *data* is just the element being repeated; otherwise, *data* is the raw data stored in a vector form.

SciDB adopts a shared-nothing model, and partitions the chunks across all instances in a cluster. Each instance processes its own data, and only redistributes data across instances if the query plan explicitly asks so. One instance acts as the coordinator. During query processing, the coordinator parses and optimizes the query, orchestrates the evaluation of partial query fragments among instances, and returns the final result to the user.

Loading data into SciDB

A user can load data into SciDB using the `load()` operator, which supports loading in parallel from all instances. Parallel loading requires a separate file for each instance. Because popular formats like CSV may serialize multi-dimensional arrays in a different order than SciDB, loading from these formats is a two-step process: First, the user issues a `load()` statement to create a one-dimensional array where the coordinates and the variables are saved as separate attributes, akin to a relational table. This one-dimensional representation needs to be converted to a multi-dimensional array to benefit from the expressive power of the array data model. The user subsequently performs a *redimension* operation to create a multi-dimensional array from the one-dimensional coordinate attributes.

This two-step loading process is inefficient both in terms of computation and storage. Apart from the computation cost of converting and rearranging data, extra staging space is also needed in the process. Loading an r-dimensional array with a attributes requires staging space for $r + a$ attributes to stage the one-dimensional output of the `load()` operation. The following *redimension* operation then compacts this array to store a attributes only. Loading massive multi-dimensional arrays with a single attribute is particularly inefficient: for example, a 5-dimensional, single-variable array that is 1 TiB in size would temporarily require 6 TiB of staging space during loading.

Supported file formats

Plain text formats like comma-separated value (CSV) files are very versatile and widely used, but these formats are also verbose and could impose tokenization and parsing overheads. These disadvantages are exacerbated as the data volume grows. Binary formats are thus more common in large-scale scientific computing, as they trade versatility for compactness and performance. SciDB supports two binary formats for data input and output: the `opaque` and `binary` formats [33].

The `opaque` format simply copies each RLE-encoded chunk along with its metadata information directly onto disk. This format is mainly used for database backups. An application that reads or produces data in this format needs to understand how SciDB organizes the chunk data internally. Hence, the `opaque` format is not suitable for scientific applications that desire interoperability.

The `binary` format concatenates the binary representation of every attribute for each cell, and serializes cells in

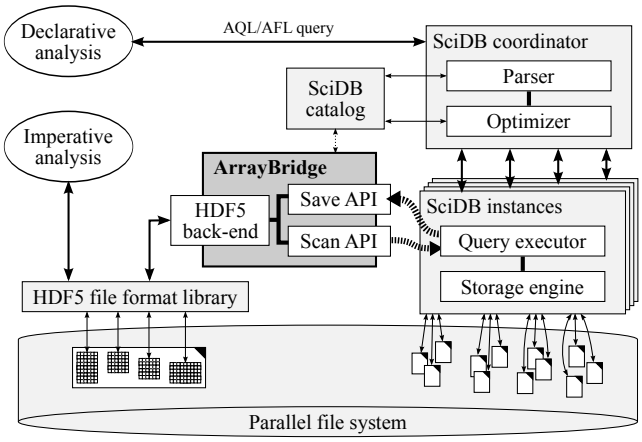


Figure 1: Through ArrayBridge, scientists can intermingle declarative and imperative analyses over the same dataset. The prototype implementation of ArrayBridge is integrated with SciDB and uses HDF5 for data storage.

row-major order. This process is time-consuming as it requires extracting, converting and copying cells from their native RLE representation into the `binary` format.

2.2 The HDF5 file format

The HDF5 format [15] is a prominent scientific data format. Data in HDF5 files is organized using two key objects: datasets and groups. A *dataset* is a multi-dimensional array containing the same type of elements. The data in an HDF5 dataset can be split into multiple *chunks*, each of which stored in a separate contiguous block in the file. HDF5 uses the same chunking method as SciDB. Similar to directories in a file system, *groups* organize data objects such as datasets and other groups into a hierarchical structure.

An HDF5 array is always logically dense. However, applications can register a well-defined value as the *fill value* of a dataset. If no chunk has been created, HDF5 will return the fill value on access. This saves space for arrays with contiguous empty regions, such as a triangular matrix.

The virtual dataset feature

A recent feature of HDF5 is *virtual dataset* support [23, 28] that combines data from multiple source datasets. A virtual dataset can be accessed as an ordinary dataset, but does not store actual data. A virtual dataset defines a list of mappings $m_1, m_2, m_3, ..., m_n$, that describe where the actual data is stored. A mapping is represented as a tuple $< d, src, dst >$; a source dataset d, where the actual data is stored; a source selection src marks the elements in d that are parts of the virtual dataset; and a target selection dst marks the logical positions of the source elements in the virtual dataset.

When a program reads or writes a region in a virtual dataset, the HDF5 library traverses the mapping list to find all the mappings that intersect with the queried region, and then propagates the reading or writing operation to the corresponding source datasets sequentially. As of version 1.10, the HDF5 library does not support removing items from the mapping list, therefore, the only way to modify a virtual dataset is to recreate the list from scratch.

3. SYSTEM OVERVIEW

ArrayBridge allows imperative manipulations and declarative queries to be issued against the same array object on disk. An overview of the system is shown in Figure 1. ArrayBridge exposes two interfaces: a read interface which scans an external array and a write interface that saves into an external array. Our prototype implementation of ArrayBridge integrates with the SciDB query engine and uses the HDF5 library as its storage back-end.

Users query HDF5 data using AQL/AFL queries. We refer to an HDF5-resident array as an *external* SciDB array, to distinguish it from the *native* SciDB arrays. Whenever an external array is queried, control is routed to ArrayBridge which returns the underlying HDF5 data. The new operator `create_array_hdf5()` declares an external array. The user specifies the name and the schema of the array, as well as the HDF5 file and the dataset name, as such:

```
create_array_hdf5(array1, <val1:double>
        [i=0:999,100,0], "data1.hdf5:val1");
```

This statement creates a 1000-element array with one double attribute, stored in HDF5 dataset `val1` in the file `data1.hdf5`. This populates metadata about the array schema in the SciDB catalog (a PostgreSQL database).

One SciDB array can contain multiple attributes (columns), whereas data in HDF5 is organized into single-attribute datasets. Hence, an external array can contain more than one HDF5 datasets with the same shape, each represented as one attribute of the array. Users create multiple attributes in an external array by listing all the attributes and the corresponding datasets in the `create_array_hdf5()` statement.

4. READING EXTERNAL ARRAY OBJECTS

This section presents how ArrayBridge reads HDF5 data and exposes them to SciDB. After an external array is created, a user retrieves the contents of the external array in an AFL query using the `scan_hdf5()` operator. Depending on the query, the `scan_hdf5()` operator either reads the entire array or selectively retrieves specific chunks of the array. This new operator has the same semantics as the `scan()` operator that accesses native SciDB arrays.

4.1 Design of the Scan operator

At the planning stage, SciDB parses an AFL query and generates a query plan for optimization. The query plan is a tree of array operators. During the optimization phase, each operator reports the schema and the estimated size of its output array. (ArrayBridge uses the array metadata in the SciDB catalog for this information.) Each operator in SciDB exposes a chunked-based iterator interface that returns one array chunk of its output upon request.

The interface to the `scan_hdf5()` operator consists of three functions: $Start(obj, attr)$, $Next()$ and $SetPosition(pos)$. The operator is initialized by calling $Start()$ with two parameters that indicate the requested array object obj and the requested attribute $attr$ in that object. After the operator is initialized, calling $Next()$ repeatedly iterates through the array and returns the next chunk assigned to this instance. If the query needs to retrieve specific array chunks, the parent operator will intersperse calls to the $SetPosition()$ function in between calls to $Next()$. $SetPosition()$ accepts a coordinate pos as its parameter for random accesses to

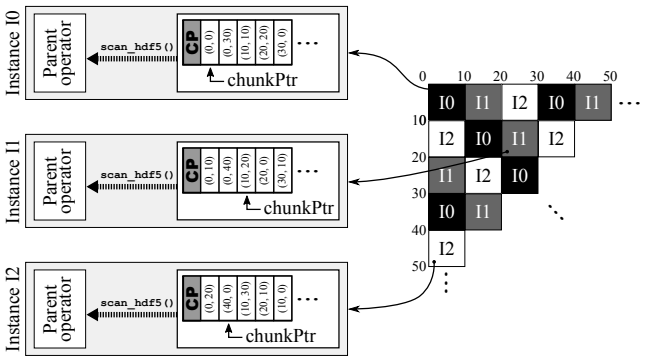


Figure 2: SciDB processes data directly from an external HDF5 array using the `scan_hdf5()` operator of ArrayBridge.

the external array object. If the chunk at *pos* is assigned to this instance, $SetPosition(pos)$ returns true; the next invocation of the $Next()$ function will retrieve the chunk that contains the element at *pos*. $SetPosition(pos)$ returns false if the chunk at *pos* is not assigned to this instance.

Given that ArrayBridge processes data concurrently on multiple nodes, one aspect of the design is how to assign the chunks of an array to different SciDB instances. The native SciDB storage engine assigns chunks to instances when `create_array()` is called, and stores this assignment in the SciDB catalog. This design choice is limiting, however, because imperative applications can change the shape of external array objects through direct calls to the HDF5 API and leave SciDB with stale metadata. ArrayBridge has significant more leeway on when to map chunks to instances because external files on a parallel file system are visible to all instances. ArrayBridge thus assigns chunks to instances at query time. Assigning chunks to instances at query time mitigates load skew and presents an opportunity to update stale metadata to their correct values.

4.2 Implementation of the Scan Operator

We now describe the implementation details of the `scan_hdf5()` operator. The pseudocode is shown in Algorithm 1.

The Start() method: Before the scan operation commences, all instances first determine how to partition the chunks to parallelize the scan operation. We identify each chunk by its coordinates, and use a mapping function $\mu()$ to abstract different chunk assignment algorithms [31]. We use the round-robin assignment in our implementation.

Each instance creates an ordered array CP in memory that stores all the chunks that the mapping function has assigned to this instance. When $Start(obj, attr)$ is called, the catalog is consulted to translate an $(obj, attr)$ reference into the appropriate HDF5 file name f and the dataset d. $Start()$ then opens file f, reads the array shape information for dataset d, and iterates over all chunk positions. Instance i adds chunk c to its local CP if and only if $\mu(c) = i$. Finally, the chunk read pointer *chunkPtr* is initialized to the first chunk.

The Next() method: $Next()$ reads the chunk at the coordinates pointed by *chunkPtr* from the HDF5 dataset, and advances *chunkPtr*. Reading data from an HDF5 file into memory is done by calling `H5DRead()` and passing the source dataset d, the source region r, and the destination buffer c to store the in-memory data. The HDF5 library uses the C array representation for in-memory data.

Algorithm 1: The `scan_hdf5()` operator.

```
1  Function Start(Array obj, Attribute attr)
2      (f, d) ← lookup (obj, attr) in the SciDB catalog
3      open the HDF5 file f and the HDF5 dataset d
4      empty CP
5      foreach chunk p_i in dataset d do
6          if μ(p_i) = this instance then add p_i into CP
7      end
8      chunkPtr ← first element in CP
9  end
10 Function Next()
11     c ← a new RLE chunk with unique elements
12     r ← region at chunkPtr to read from
13     call H5Dread(d, r, c) to read data into c
14     advance chunkPtr
15     return c
16 end
17 Function SetPosition(Coordinate pos)
18     p̂ ← the coordinates of the chunk that contains pos
19     if p̂ ∈ CP then
20         chunkPtr ← p̂
21         return True
22     end
23     return False
24 end
```

The technical challenge is to efficiently convert HDF5 array data from their C representation into the RLE format that the SciDB operators use. We discovered that converting the data into the RLE representation is a serious performance hit for the dense arrays that HDF5 is well-optimized for. We instead opted to masquerade a dense HDF5 chunk as an RLE-compressed chunk c where each element is unique, and request `H5DRead()` to place the data directly into c. This avoids redundant data copying and format conversions.

The SetPosition() method: $SetPosition(pos)$ first computes the coordinates $\widehat{p}$ of the chunk that contains *pos*, and performs binary search in the CP array to find $\widehat{p}$. If $\widehat{p}$ is found, $SetPosition$ changes *chunkPtr* to $\widehat{p}$ and returns true, otherwise it returns false.

5. SAVING ARRAY OBJECTS IN FILES

Many post-processing and visualization workflows are conducted over file-centric APIs. It is thus important to efficiently serialize arrays into popular scientific data formats such as HDF5. This section presents how ArrayBridge materializes SciDB objects into external HDF5 files. The two key features that ArrayBridge supports are (1) parallel writing that bypasses the single-writer limitation of HDF5, and (2) time travel so that imperative kernels can access previous versions of a dataset over the HDF5 API without modifying the code. We describe each in turn.

5.1 Balancing efficiency and interoperability

Although an HDF5 file can be read by multiple SciDB instances concurrently, only one SciDB instance can be writing to a single HDF5 file at a time. This fundamental Single-Writer, Multiple-Readers (SWMR) design constraint of HDF5 is a roadblock to parallel writing and it prevents

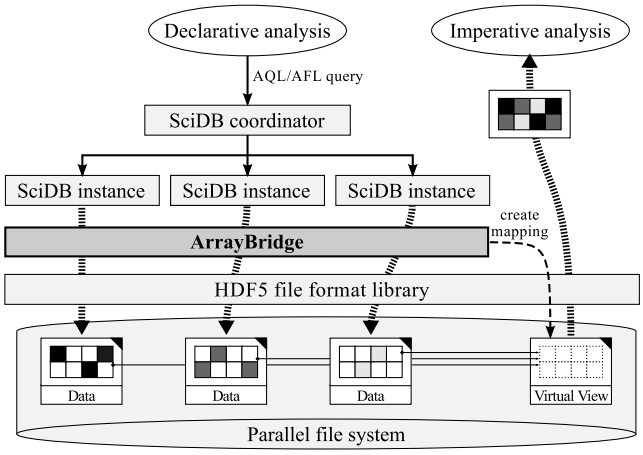


Figure 3: The virtual view mechanism bypasses the single-writer limitation of the HDF5 file format by directing write streams from different SciDB instances into separate objects. Existing imperative analyses access a single object through the view.

SciDB from taking advantage of the highly concurrent I/O subsystems found in modern scientific computing facilities.

Serializing array data into the HDF5 format exposes a dilemma between *writing efficiency* and *interoperability*. To write data in parallel, SciDB must produce multiple HDF5 files. Splitting a single dataset into multiple files, however, makes analysis more cumbersome. Conversely, storing the entire dataset in a single file makes it straightforward to manage and analyze the HDF5 dataset from existing applications, but it limits the write throughput to the I/O capacity of one SciDB instance.

SciDB exposes this trade-off between efficiency and interoperability to the user. The `save()` operator provides two separate writing modes. In the *Serial* mode, the data is shuffled to the SciDB coordinator which writes all the data to a single file. This way, processing and managing the data is straightforward, but the writing throughput is limited to the I/O capacity of one instance. In the *Partitioned* mode, each instance writes the data it stores to a separate file. This way, the writing is parallelized, but the user needs to deal with the complexity of maintaining partition information and adapting existing tools to read partitioned files.

ArrayBridge supports a third writing mode, the *Virtual View* mode, to avoid the artificial dilemma between sacrificing writing efficiency or interoperability. The *Virtual View* mode writes the data on each instance to separate HDF5 files, and then creates a virtual dataset that logically combines these datasets into one object. The *Virtual View* mode utilizes the virtual dataset feature of the HDF5 format, which presents the data in several HDF5 datasets as a single dataset to the application accessing it. The benefit of the *Virtual View* mode is that it combines the efficiency of parallel writing into separate files with the interoperability of producing a single file for the small upfront cost of creating the virtual dataset.

5.2 Serializing SciDB arrays into HDF5

ArrayBridge extends the SciDB `save()` operator to support the HDF5 format. This allows a user to write data back to external HDF5 files. The *Serial* and the *Partitioned* writing modes adopt the SciDB semantics: In the *Serial* mode, data is shuffled to the coordinator, and the coordinator writes the data into an HDF5 file. In the *Partitioned* mode, each instance creates a different file and saves the data assigned to it to a dataset with the same shape as the original array. The chunks that have not been assigned to the local instance are empty: HDF5 logically fills them with a *fill value* but does not store any additional data.

The *Virtual View* mode starts by having every instance write its chunks to a separate file, just like in the *Partitioned* mode. In addition to writing the chunks, each instance also maintains two regions: one is the source region in the local file (src), the other is the target region in the virtual dataset (dst). After the chunks are written, the *Virtual View* mode needs to create the virtual dataset.

ArrayBridge implements two methods to create the virtual dataset. The first method is *parallel* mapping. Each instance appends its mappings into the virtual dataset, using the src and dst objects it created during the writing process. The virtual dataset needs to be recreated, as it cannot be updated directly. Thus, each instance needs to read the mappings in the virtual dataset, append its own mapping to it, and recreate the dataset using the new mapping list. Because only one instance can update the virtual dataset at a time, the parallel mapping technique uses file locking to ensure mutual exclusion. This crude synchronization method allows each instance to update its own mapping without waiting for other instances to exchange mapping data. However, because each update recreates the dataset, $O(n^2)$ mappings will be written for a cluster with n SciDB instances.

The second method of creating the virtual dataset is *coordinator* mapping. Each instance transmits the src and dst regions to the coordinator, which concatenates the per-instance mapping lists and creates the virtual dataset. This requires all SciDB instances to synchronize on a barrier and wait for the coordinator to create the virtual dataset. The coordinator mapping technique writes $O(n)$ mappings for a cluster with n SciDB instances. We evaluate the performance trade-offs of these two techniques in Section 6.4.

5.3 Backward-compatible time travel in HDF5

An oft-requested feature is the ability to query past versions of a dataset to understand how a dataset evolved or to explain the processing steps to the final result. This subsection introduces the time travel capability in ArrayBridge that transparently deduplicates regions that are identical between dataset versions that are stored in the HDF5 file. This reduces the storage footprint of versioned datasets and is interoperable with imperative applications that access former versions through the existing HDF5 API.

Because analyses predominantly access the latest version of a dataset, ArrayBridge always fully materializes the latest version to minimize reconstruction costs. Past versions are stored under a separate group in the HDF5 object hierarchy. All versions can be accessed as ordinary HDF5 datasets by applications via the HDF5 API. A user produces a versioned dataset by passing a parameter in the `save()` operator; ArrayBridge accesses old versions as ordinary HDF5 datasets via an explicit call to the `scan_hdf5()` operator. ArrayBridge supports two techniques for saving a versioned dataset: Full Copy and Chunk Mosaic.

Full Copy: The Full Copy technique incurs the cost of materializing every past version. For example, if a query calls `save()` to update the dataset `speed` to version `V2`, Ar-

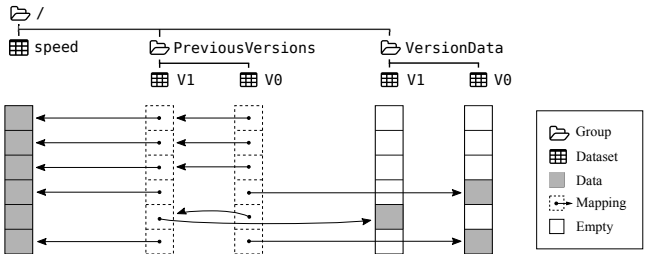


Figure 4: An example of the Chunk Mosaic technique. The latest version is the fully-materialized dataset `speed`. Old versions are accessed through virtual datasets in the `PreviousVersions/` group.

rayBridge renames dataset `speed` as `PreviousVersions/V1`. After this metadata operation is completed, the `save()` operator creates a new dataset named `speed` that stores the latest version `V2`.

Chunk Mosaic: The Chunk Mosaic technique only stores the chunks that are updated and creates a virtual dataset to "stitch together" the past version. An example of the Chunk Mosaic technique is illustrated in Figure 4.

The Chunk Mosaic technique proceeds in two steps. The first step is creating an HDF5 dataset that stores the previous versions of the array chunks that are updated. The unmodified chunks are empty. (Recall that the HDF5 library does not store empty chunks). This dataset has the same shape as the original dataset, and is hidden away in an HDF5 group labeled `VersionData/`. For example, in Figure 4, the HDF5 dataset `VersionData/V0` only stores the chunks of version `V0` of the `speed` dataset that were updated in version `V1`. Because SciDB does not convey which chunks were updated to the `save()` operator, our current implementation discovers which chunks changed by comparing the chunk that is being saved with the chunk that already exists in the latest version of the dataset.

The second step of the ChunkMosaic technique creates a virtual dataset under the `PreviousVersions/` group in the HDF5 file. This dataset combines the unmodified chunks of the latest version and the updated chunks in the `VersionData/` group into a single view. This virtual view will be accessed by applications to retrieve this old version. In the example shown in Figure 4, suppose that the update creates version `V2` which is stored in dataset `speed`. The Chunk Mosaic technique first creates the virtual dataset `PreviousVersions/V1`. If a chunk was modified, the virtual dataset chunk is mapped to the `VersionData/V1` dataset that contains the original chunk in the previous version; otherwise the chunk is unmodified and is mapped to its latest version at `speed`. Finally, the mappings in the dataset `PreviousVersions/V0` that point to `speed` are modified to point to `PreviousVersions/V1`. The Chunk Mosaic technique produces a series of chained virtual datasets that can reconstruct any previous version by either referencing the latest fully-materialized version or a former version of a chunk in the `VersionData/` group.

6. EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the scan, save and time travel functionalities of ArrayBridge. We consider the following questions:

- (§6.2) How does the performance of ArrayBridge compare with native SciDB when reading data from a parallel file system in a typical HPC environment? How much time and space does loading take?
- (§6.3) How does ArrayBridge scale when processing a real multi-TiB simulation dataset that would be infeasible to load in SciDB?
- (§6.4) How efficient is the save mechanism of ArrayBridge, and how does it scale? How effective is the time travel mechanism in deduplicating versioned array objects?

6.1 Configuration and methodology

We evaluate our implementation on the Edison computer of the National Energy Research Scientific Computing (NERSC) facility. Edison is a Cray XC30 supercomputer with 5,586 compute nodes. Each node has two 12-core 2.4Ghz Intel "Ivy Bridge" processors and 64 GiB memory. File storage is provided via the Lustre [24] parallel file system. Lustre distributes the content of a file across multiple I/O servers, called Object Storage Targets (OST), to provide I/O concurrency for highly-parallel applications. Each file is divided into a user-defined *stripe size*, and then these stripes are distributed to the requested OSTs in a round-robin fashion. The number of OSTs a file is distributed to is called the *stripe count*. The file system we use has a total of 248 Lustre Object Storage Targets (OSTs) and more than 30 PiB of total storage. The reported peak I/O throughput exceeds 700 GiB/sec.

We evaluate ArrayBridge on the SciDB 15.12 Community release. According to the SciDB guidelines [32], we configure 8 instances per node and a 16 MiB chunk size. The HDF5 datasets are chunked using the same chunk size (16 MiB). Unless we specify otherwise, we follow the NERSC-recommended I/O configuration of striping each file to 72 Lustre OSTs with a 1 MiB Lustre stripe size.

One challenge in reporting performance results from large-scale HPC facilities such as NERSC is that the computers are always busy with multiple concurrent jobs. This brings high variability in I/O performance. We have sometimes observed variance that exceeds 100% for the same experiment. Many of our results thus report performance in a box plot to convey the effect of variance. We draw a thick horizontal band at the median value, while the bottom and top of the box show the first (25%) and third (75%) quartiles. The whiskers are drawn at the lowest and highest datum within 1.5× of the inter-quantile range (IQR).

Another evaluation challenge is that there is no mechanism to evict cached objects from the parallel file system, because caching happens in multiple layers of the I/O path to cold storage. We thus revert to yielding the computer to other jobs to clear the parallel file system cache. In our experiments, we wait for at least one hour before reading the same data again to ameliorate any caching effects.

6.2 Scan

This subsection evaluates the efficiency of analyzing HDF5 file directly using ArrayBridge through queries on synthetic datasets.

Access method comparison

The first question is how the full scan performance of ArrayBridge compares with the native SciDB storage engine. We choose aggregation as a representative query for the full-scan

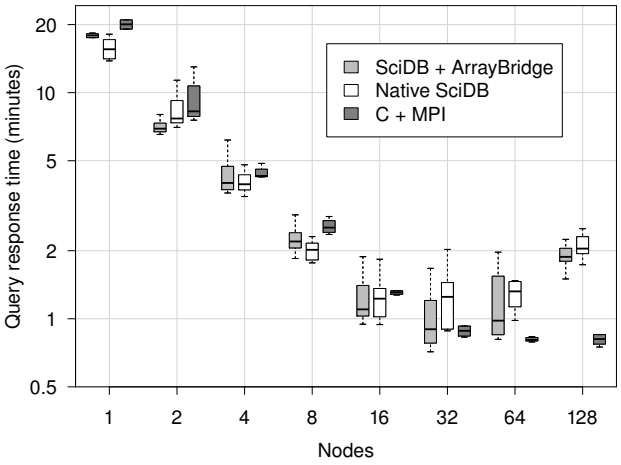


Figure 5: Time to aggregate a 1.5 TiB dataset with different cluster sizes.

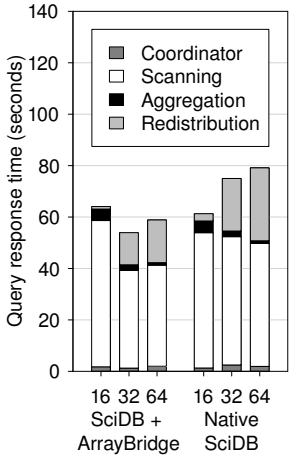


Figure 6: Time breakdown to aggregate the entire 1.5 TiB dataset.

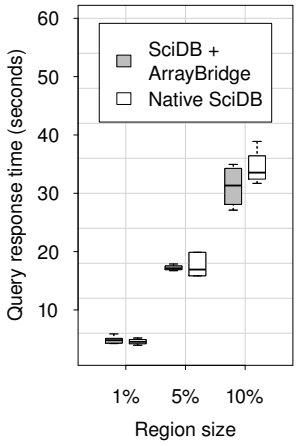


Figure 7: Response time to aggregate contiguous regions in the 1.5 TiB dataset.

pattern, and use an one-dimensional dataset of 192 billion double numbers (approximately 1.5 TiB data). For comparison purposes, we also implemented an imperative aggregation kernel in C that is parallelized through MPI and processes data directly through the HDF5 API. We vary the number of nodes from 1 to 128, and report the query response time in Figure 5.

As the result indicates, the query response time of ArrayBridge is as good as native SciDB: the performance of ArrayBridge is statistically indistinguishable from the performance of the native SciDB engine. All implementations scale near perfectly from 1 to 32 nodes. ArrayBridge achieves its best performance with 32 nodes, aggregating 1.5 TiB of data at a rate of $\sim$ 28 GiB/s. However, while scaling beyond 32 nodes, the response time of the imperative C kernel flattens out, but the response time (and variance) for both ArrayBridge and native SciDB increase.

Figure 6 reports the time breakdown when using 16, 32, and 64 nodes to shed light on why query response time increases in SciDB. We decompose the query response time into four parts: *coordinator time*, the time coordinator uses to parse and optimize the query, broadcast the query plan, and send the result to client; *scanning time*, the average time used to read chunk data from HDF5 library or SciDB storage engine by each instance; *aggregation time*, the average time to aggregate data in local chunks at every instance; and *redistribution time*, the time to redistribute the local aggregation results to the coordinator.

In our experiment, the scanning time for both the ArrayBridge and native SciDB flattens beyond 32 nodes. This mimics the I/O behavior of the imperative C kernel (see Figure 5). The aggregation time does scale perfectly, but it only accounts for a small portion of the query response time, whereas the coordinator time stays constant and only occupies a small portion of the response time. The culprit for the increased query response time is the redistribution cost, which significantly increases with more than 32 nodes. This also explains why the imperative kernel outperforms ArrayBridge and native SciDB with more than 64 nodes. Profiling the imperative kernel revealed that it takes almost the same time to scan the HDF5 file and aggregate locally as for ArrayBridge, but it takes far more time to redistribute partial aggregates in SciDB than to aggregate the result with MPI.

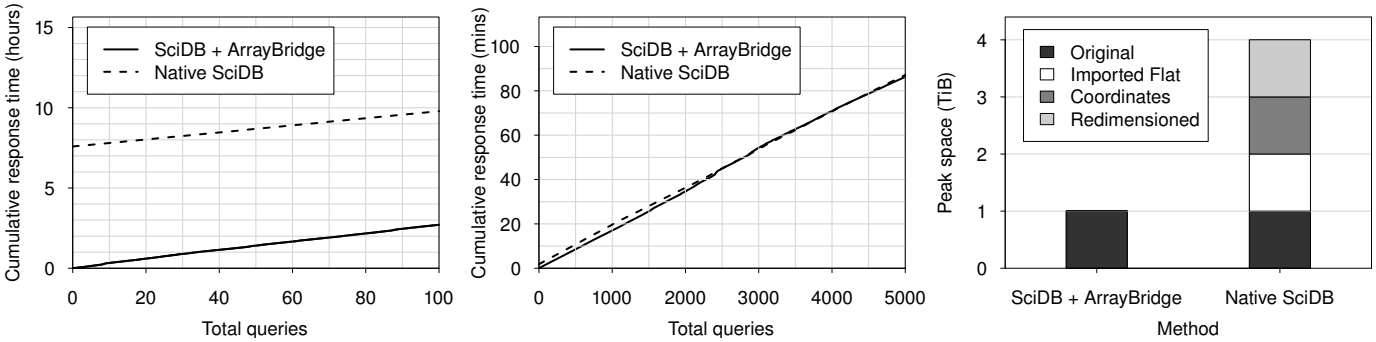
Of course, not all queries will access the entire dataset. To evaluate the effect of implicit indexing in SciDB, we also run a block selection query that randomly aggregates one contiguous region of the 1.5 TiB dataset using 8 nodes. We vary the size of the selected region from 1% to 10% of the dataset. The result is shown in Figure 7. The response time of ArrayBridge and native SciDB is statistically indistinguishable regardless of the selected region size. This shows that the SciDB chunk index does not bring a performance advantage over scanning a chunked HDF5 file directly even when only retrieving a region of a dense array.

Cumulative query response time with loading

Another question is how much extra time and space is needed for loading the data, and whether this upfront time investment accelerates subsequent queries to the same dataset. We evaluate these questions by repeatedly aggregating the same dataset using both ArrayBridge and native SciDB. For ArrayBridge, we query the HDF5 dataset directly; for native SciDB, we load the binary dataset in parallel and redimension it into an one-dimensional array. We fix the cluster size to 8 nodes, and use two synthetic datasets: the first is only 16 GiB and comfortably fits in the local buffer pool (chunk cache) of each SciDB instance; the other is 1 TiB which is nearly twice as big as the local memory in each node.

Figures 8a and 8b report the cumulative query response time of loading and evaluating the first n queries, $n = 1, 2, 3, ...$ The value at $n = 0$ represents the dataset loading time. For the 1 TiB dataset, it takes more than 7.5 hours for the native SciDB to load the data, redimension it, and return the first query result. (The redimension operation alone takes a little more than 7 hours.) In comparison, it only takes ArrayBridge about 1.5 minutes to answer the first query! ArrayBridge provides the answer 300× faster by skipping the time-consuming loading process.

Loading the data into SciDB does not significantly accelerate subsequent queries, either. Surprisingly, this holds even for the 16 GiB dataset which fits entirely in the buffer pool. Recall that the `scan_hdf5()` operator issues I/O requests to the HDF5 file on every call, unlike native SciDB that can redirect I/O requests to the buffer pool. We conclude that caching and prefetching by the Lustre parallel filesystem can



(a) Time to aggregate a 1 TiB dataset that does not fit in memory.

(b) Time to aggregate a 16 GiB dataset that fits in the SciDB buffer pool.

(c) Peak storage requirement when loading a 1 TiB dataset.

Figure 8: Cumulative query response time and peak space consumption to load and complete multiple queries.

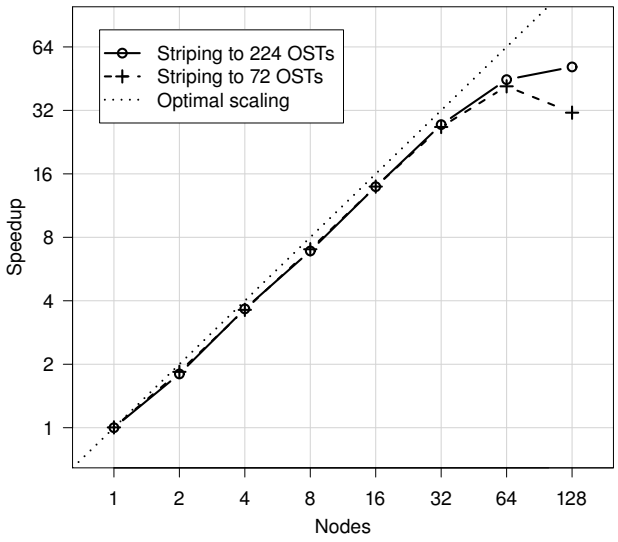


Figure 9: Speedup when analyzing 4.3 TiB of post-processed Particle-In-Cell simulation data with ArrayBridge. Striping to more Lustre OSTs better utilizes the parallel file system at scale.

be as effective as the native SciDB buffer pool in curtailing redundant I/O to cold storage for full scan access pattern.

Another significant advantage of querying HDF5 data directly in its native format is reducing the space overhead of loading. Figure 8c breaks down the peak space consumption during the loading process. As described in Section 2, the steps of importing an one-dimensional array into SciDB includes importing the original data into a flat SciDB array with coordinate information, and then convert it to the redimensioned SciDB format. For a one-dimensional array, this occupies 3× more space than the original data. Datasets of higher dimensionality would need even more space, as the size of the coordinate data increases linearly with the dimensionality of the dataset.

6.3 Analyzing particle-in-cell (PIC) simulation data with ArrayBridge

We now evaluate the performance of ArrayBridge when analyzing a scientific dataset that would be infeasible to load into SciDB due to its massive size. An opportunity for symbiosis between declarative and imperative analyses occurs during post-processing of large-scale simulation data. The dataset is from the plasma physics domain. The scientific application uses VPIC [8], a general purpose particle-in-cell simulation code, for modeling kinetic plasmas with billions of particles. The simulated phenomenon of interest occurs when an intense femtosecond laser pulse is focused on a solid target to create a dense plasma that acts as a so-called "plasma mirror" and specularly reflects incident light [18, 39]. This generates oscillations of the plasma mirror surface, which in turn deforms the reflected field by the Doppler effect and produces intense attosecond pulses. Studying the Doppler harmonic radiation spectrum is challenging, as it requires simulating and analyzing a 3D field with very high resolution. For this problem, one simulation step produces about 50 TiB of data.

As in many other large simulations, the current practice is to perform analyses in phases after the simulation result has been produced. The first phase discards low-energy particles of little interest and reduces the dataset to a 4.3 TiB array in the HDF5 format for further analysis, visualization, sharing and archival storage. This 3D dataset consists of four variables: one variable per component of the 3D velocity vector $\vec{v} = (v_x, v_y, v_z)$, and one variable for the energy E. The query intent is to aggregate the velocity $\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$ and energy E for high-energy particles ($E > 2.0$) over a grid. This mundane analysis can be succinctly conveyed in a SciDB AFL query that directly accesses these HDF5 objects via ArrayBridge.

The scalability of this aggregation query over the 4.3 TiB array dataset is shown in Figure 9. Processing this dataset using a single node returns the answer in 137 minutes. We observe near-optimal speedup up to 32 nodes. With more nodes, the speedup critically depends on the I/O performance. The recommended striping policy of 72 OSTs limits scaling beyond 64 nodes because it limits the available I/O throughput. Striping to 224 OSTs allows for higher I/O utilization, which improves performance for 64 and 128 nodes. Scaling stops at 128 nodes as the internal query processing operations in SciDB become the bottleneck (see Section 6.2 for a detailed discussion).

Overall, ArrayBridge can use 64 nodes to aggregate 4.3 TiB of array data in under 3 minutes for this AFL query. Loading and redimensioning a dataset of this volume into a

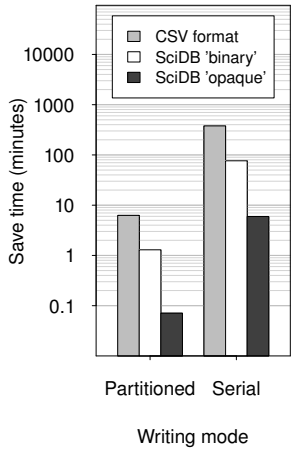


Figure 10: Time to save a 64 GiB dataset in different formats.

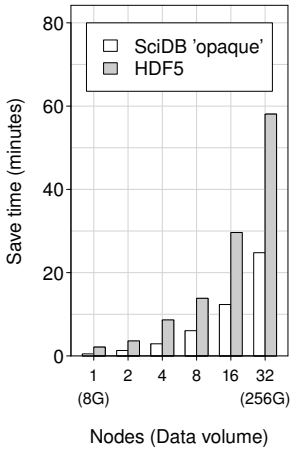


Figure 11: Time to save 8 GiB/node in different formats with the *Serial* writing mode.

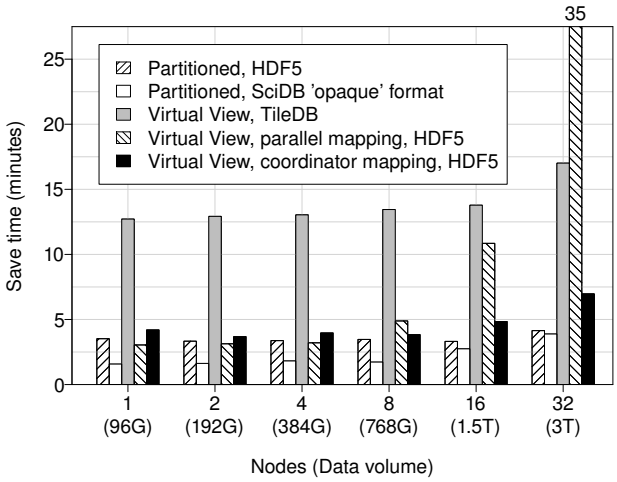


Figure 12: Time to save 96 GiB/node using the *Partitioned* and the *Virtual View* writing modes in HDF5, TileDB and SciDB's proprietary "opaque" formats.

64-node SciDB cluster would have taken more than a day.

6.4 Save

SciDB supports multiple formats for exporting data, including the widely used comma-separated value (CSV) format, the SciDB 'binary' format, and the proprietary 'opaque' format. (See Section 2 for details.) We evaluate the performance of saving in these formats by storing a synthetic two-dimensional 64 GiB dataset of double numbers with 8 nodes, using both the serial and partitioned writing modes.

Figure 10 reports the median query response time of each format/mode combination. (Note that the vertical axis is logarithmic.) As the figure indicates, the CSV and the SciDB 'binary' formats are very slow to write into. Even with 8 nodes writing CSV data in parallel, the writing throughput per node is only $\sim$ 20 MiB/s due to the overhead of converting doubles into a text representation. The SciDB binary format is almost 5$\times$ faster as it avoids text conversions and merely reconstructs the binary on-disk representation from the RLE encoding. The 'opaque' format is 10$\times$ faster than the SciDB 'opaque' format, as it merely dumps the RLE-encoded chunks on disk. Although the 'opaque' format is intended for backups and is not designed to be interoperable across applications, we report save times for the SciDB 'opaque' format in the experiments that follow as it better approximates the peak aggregate writing capability of a SciDB cluster.

Problem of serial writing

We evaluate the performance of the serial writing mode by saving a synthetic two-dimensional dataset with 8 GiB data per node to disk. (Recall that the serial writing mode produces a single output file by redirecting all data to a single SciDB instance.) We vary the number of nodes from 1 to 8, and report the save time for both the HDF5 format and the SciDB 'opaque' formats. Figure 11 shows the median query response time. Although the computational power and I/O capacity increase with the number of nodes, the aggregate writing throughput does not increase accordingly as the single writing instance is the bottleneck; hence, the response time increases as the data volume increases. Profiling the result further shows that both shuffling time and writing time double when the number of nodes doubles. Although materializing arrays using the serial mode produces a single file, its performance does not scale.

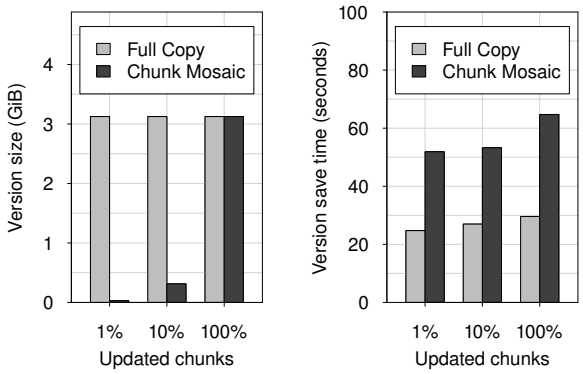
Writing in parallel

We now consider the performance of writing HDF5 files in parallel using the partitioned and virtual view modes. We generate a synthetic two-dimensional dataset with 96 GiB data per node. Figure 12 compares the query response times of saving this dataset using each format/mode combination.

In terms of scaling, partitioned writing performs well. Writing HDF5 in partitioned mode scales perfectly from 1 to 16 nodes, maintaining a writing throughput of about 450 MiB/s per node. Writing into the SciDB 'opaque' format using partitioned mode also scales perfectly from 1 to 8 nodes, which records a per-node writing throughput of about 1 GiB/s. The writing throughput of the 'opaque' format decreases at larger scale and matches the writing throughput of the HDF5 format for 32 nodes. However, the disadvantage of the partitioned writing mode is that it produces one file per SciDB instance. At 32 nodes, the output is contained in 256 files, which can be cumbersome to manage.

The query response time of the virtual view mode with local mapping technique is indistinguishable from the partitioned writing for clusters up to 4 nodes. However, the response time goes up while scaling beyond 8 nodes: at 16 nodes, it takes almost 1.5$\times$ longer to write data in the local mapping mode compared with the partitioned mode; it's almost 10$\times$ as much when using 32 nodes. This is because of the substantial cost of updating virtual datasets, where each instance reads and re-writes all prior mappings. Profiling shows that updating the virtual dataset takes about 90% of the time for 32 nodes.

The coordinator mapping technique significantly ameliorates this overhead, as the coordinator collects all the mappings and creates the virtual dataset once. For 32 nodes, the coordinator mapping technique reduces the virtual dataset creation time to about 30 seconds, or less than 10% of the query time. This allows the user to save a 3 TiB array using virtual view within 7 minutes, and preserves the advantage of allowing imperative analysis kernels to access it as a single



(a) New version size. (b) Version save time.

Figure 13: Version size and save time to update a 3.1 GiB dataset.

dataset.

To evaluate the efficiency of the HDF5 virtual mapping mechanism, we also compare the performance of saving in the HDF5 format with the performance of saving in the TileDB [27] format, which also supports viewing a partitioned array as a single object. TileDB organizes arrays into *fragments* that contain ordered elements in a hyper-rectangular array region. Fragments are stored in separate directories in the underlying file system. This means that different processes can write to different fragments in parallel without conflicts. Akin to the virtual dataset functionality of HDF5, the TileDB library scans all the fragments that intersect with the accessed region, and combines the result at query time.

We implemented a version of the `save()` operator in ArrayBridge that stores data in TileDB. The result shows that saving to the TileDB format takes about $4 \sim 5\times$ more time than saving in the HDF5 format with virtual view. Profiling shows that this slowdown is largely caused by the overhead of creating and closing fragments, which takes nearly 60% of the time for a cluster with 16 nodes. In addition, due to the sheer number of generated directories and files, accessing and managing the array dataset is very time-consuming due to the filesystem metadata management overheads. For example, merely listing the contents of the directory that stores a 1.5 TiB array in the TileDB format using `'ls'` takes more than 30 minutes.

Time Travel

We now evaluate how effectively does the Chunk Mosaic technique deduplicate versioned datasets during time travel queries when compared with the Full Copy technique.[1] We use a synthetic 3.125 GiB dataset of double numbers with 16 MiB chunk size. To create a new version, we first identify a range of chunks in the dataset, and randomly change N elements in these chunks, $N = 1\%$ of the total elements in the array. The number of chunks being updated is varied from 1% of total chunks (2), to 100% of total chunks (200).

Figure 13a compares the version size created by the two time travel techniques. Writing a new version using Full Copy occupies the same space as the original one, since it always duplicates the previous dataset. In contrast, the space usage of the Chunk Mosaic is proportional to the number of chunks being updated. The Chunk Mosaic technique can thus significantly reduce the storage footprint to support time travel analysis.

Figure 13b compares the time to save a new version when the number of updated chunks varies from 1% to 100% of all chunks. The Chunk Mosaic technique takes about $2\times$ longer to write a version compared with Full Copy because it reads and compares the data being saved with the previous version. As the updated elements get dispersed to more chunks, the save time of both techniques increases.

[1]TileDB currently does not support time travel queries, although this feature is in the roadmap for a future release.

7. LESSONS LEARNED

When designing ArrayBridge, we learned the following lessons that are of broader interest to the designers of data management systems for high-end computing facilities:

1. Reading directly from external array formats can be as fast as reading from internal storage formats. We found that a carefully designed and tuned operator can read dense external arrays in the HDF5 format as fast as reading from a native SciDB array, which eliminates the processing and storage overheads that are inherent in the data loading process. This is due to the I/O-bound nature of many AQL/AFL queries and also due to the fact that the HDF5 library does not introduce overheads that slow down disk I/O. Scientific data format libraries like HDF5 are complex pieces of software with decades of engineering and careful tuning based on actual deployment experience. Although there is certainly potential for further improvement [27], existing array storage libraries should not be dismissed as non-competitive without experimental evidence.

2. CPU overheads matter. In high-end computing facilities, CPU overheads can easily encumber the I/O path and impact data processing throughput from cold storage. Our experience contradicts a design tenet for database systems that assumes an abundance of CPU cycles per I/O unit. In computers where the parallel file system offers massive I/O concurrency, this is only true when deploying at a scale of hundreds of nodes. Because job scheduling in an HPC environment often gives higher priority to small jobs that request a few nodes, scientists prefer to run simple, time-sensitive analysis jobs on a handful of nodes to reduce their job's queuing delay.

It is not uncommon to observe the I/O throughput from cold storage reaching 3 GiB/sec per node, if only a few nodes are requested. Such I/O rates require a careful investment of CPU cycles to not make data processing CPU-bound. ArrayBridge eliminates substantial CPU overheads by masquerading the HDF5 data as a special RLE chunk for processing (see Algorithm 1) and enabling *tiled mode* in SciDB which allows more efficient chunk iteration by processing elements in a batch. These optimizations alone improved the performance of the `scan_hdf5()` operator by more than $2\times$.

3. Disaggregating storage and compute requires dispensing with static partitioning of data to instances. Shared-nothing database systems partition data on load under the assumption of a fixed cluster size and hardware configuration. Systems designers need to dispense with the notion of static partitioning in a disaggregated datacenter that separates storage and compute: in a scientific computer, users request some compute capacity and are allocated to whatever nodes happened to be available first. Therefore, the number of nodes the database system runs on changes between jobs. This requires a flexible data partitioning policy that assigns chunks to nodes at query time

instead of during data loading. In addition, the type of the compute node can change between jobs. Hardware specifics like memory/CPU capabilities or MAC addresses should ideally be enumerated at startup without requiring to recreate the database.

4. A non-materialized view mechanism is important for interoperability. The scientific computing software ecosystem is diverse and fragmented. File format libraries are among the few software building blocks that are widely used across scientific domains. The file format library API mediates I/O accesses and is thus a natural integration point to deliver more sophisticated data processing capabilities to scientists. A view mechanism, such as virtual datasets in HDF5, decouples the logical array content and physical data representation. The view mechanism allows ArrayBridge to break the SWMR constraint of the HDF5 library and be backwards compatible with existing applications. I/O middleware such as ArrayBridge that support non-materialized views can transparently convert between file formats and improve the parallelization and update performance over the current HDF5 implementation.

5. The I/O hierarchy is complex in high-end computers, but caching is pervasive. When designing ArrayBridge, we debated the appropriate integration point to the SciDB engine. In particular, we contrasted the modularity of integrating at the query engine layer versus the performance of optimizations such as buffer pool caching and asynchronous I/O if one were to integrate at the storage layer. ArrayBridge integrates at the query engine layer. Our evaluation reveals that repeated accesses on both cache-resident and larger-than-memory datasets are cached by the parallel filesystem (see Section 6.2). In retrospective, the performance benefit from explicitly controlling I/O and caching would have been limited.

8. RELATED WORK

Array database systems

Database systems originally support multi-dimensional datasets by materializing array offsets as attributes of a relational table [3]. Some solutions build new primitives to express array operations within the DBMS [41, 46], while SciQL [48] extends SQL to support the array data model. Also, new query languages have been proposed for querying arrays declaratively. AQL [25] is one of the earliest array query languages. AML [26] expresses array operations as a combination of operators, each of which reads one or more arrays and returns an array. Recent work has introduced techniques to ameliorate skew during array join processing [14] and has proposed similarity joins for array data [49].

Another problem for array database systems is array storage. Of particular research interest is storage of sparse datasets, as such objects cannot be efficiently stored using a dense array representation. Soroush et al. investigate different storage management and chunking strategies for array storage and proposes a two-layer chunking strategy [37]. TileDB proposes using *flexible tiling* to address the problem of accessing sparse data for efficient writes [27]. A related challenge is accessing cells along different dimensions efficiently, to avoid performing very small I/Os or reading unnecessary data. Novel chunking and storage optimizations have been proposed to address this problem [35].

SciDB [9, 38] is an open-source array database system. SciDB uses a similar chunk-based storage as ArrayStore [37], and supports bitmap indices as well as RLE compression within a chunk to accelerate sparse array processing. It supports two query languages, an operator-based language AFL (similar to AML), and a SQL-style query language AQL.

Scientific data formats

HDF5 [16] and NetCDF [29] are two widely used portable scientific data formats. The HDF5 library provides a set of tools to organize different arrays and their metadata, such as particles [10] or biological images [13] in a hierarchical, semi-structured way. Optimizing HDF5 performance on HPC platforms is a well-studied problem; prior solutions have explored how to tune large-scale array processing programs to improve I/O performance [10, 21].

Array versioning

Soroush et al. propose versioned array storage to support both querying a specific array version and extracting the history of an array object [36]. Seering et al. propose a materialization matrix to determine which version to materialize while storing the remaining versions as deltas [34]. DataHub [4, 5] extends the idea of version control to entire relational datasets for data-intensive collaborations. Bhattacherjee et al. investigate the trade-offs between storage size and recreation time for versioning techniques [6].

***In situ* processing**

There is substantial prior research on querying external objects using a relational database system. NoDB [2] proposed to cut the cost of data loading by reading external files directly, and accelerated the querying process by utilizing positional maps and selective parsing. In a MapReduce context, invisible loading incrementally imports the data into a database when parsing data in the map phase [1]. The ScanRAW [11] operator queries the data *in situ*, while transparently loading frequent accessed data in the native storage of the database. ViDa moves beyond the *in situ* data access model and proposes just-in-time query processing [22].

Recent work has investigated *in situ* mechanisms for data in the HDF5 file format. SDS/Q [7, 12] introduces a lightweight relational query engine to directly process data in the HDF5 format, in addition to using bitmap-based indices [19, 47] to accelerate highly-selective queries. Wang et al. proposed running relational select and aggregation queries directly on HDF5 file [43, 44]. In the context of MapReduce, analysis frameworks have proposed querying HDF5 and NetCDF data directly [17, 42].

9. CONCLUSIONS

We have presented ArrayBridge, a bi-directional array view mechanism for scientific file formats that makes declarative array manipulations interoperable with file-centric, imperative analysis kernels. ArrayBridge bypasses the multi-hour load process of SciDB for massive arrays. Scientists can issue SciDB queries over multi-TiB external array datasets in the HDF5 format through ArrayBridge in a few minutes. ArrayBridge scales near-optimally for cluster sizes up to 32 nodes until it encounters internal SciDB scaling bottlenecks. In addition to fast scans, ArrayBridge can materialize SciDB objects in the HDF5 format nearly as efficiently as SciDB serializes its database in its proprietary binary format for backup. For backwards compatibility, existing applications access partitioned HDF5 objects as a single logical dataset through a virtual view mechanism. Finally, ArrayBridge can efficiently store versioned objects by deduplicating un-

modified chunks between versions. Past versions are accessible via the standard HDF5 API, which enables time travel queries from version-oblivious imperative code. The scan component of ArrayBridge is slated to be released in the 2017 community edition of SciDB. We are deploying ArrayBridge in a large scientific computing facility to make it readily available to scientists.

Acknowledgements

This work was partly supported by the National Science Foundation under grants III-1422977, III-1464381, CNS-1513120, by a Google Research Faculty Award, and by the DOE Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231. The evaluation used resources of the National Energy Research Scientific Computing Center (NERSC).

10. REFERENCES

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.

[3] P. Baumann. Management of multidimensional discrete data. *VLDBJ*, 3(4), 1994.

[4] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR*, 2015.

[5] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with DataHub. *VLDB*, 8(12), 2015.

[6] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *VLDB*, volume 8, 2015.

[7] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *SIGMOD*, 2014.

[8] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5), 2008.

[9] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[10] S. Byna, A. Uselton, D. K. Prabhat, and Y. He. Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper. In *Cray User Group meeting*, 2013.

[11] Y. Cheng and F. Rusu. SCANRAW: A database meta-operator for parallel in-situ processing and loading. *TODS*, 40(3), 2015.

[12] B. Dong, S. Byna, and K. Wu. Parallel query evaluation as a scientific data service. In *CLUSTER*, 2014.

[13] M. T. Dougherty, M. J. Folk, E. Zadok, H. J. Bernstein, F. C. Bernstein, K. W. Eliceiri, W. Benger, and C. Best. Unifying biological image formats with HDF5. *Communications of the ACM*, 52(10), 2009.

[14] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker. Skew-aware join optimization for array databases. In *SIGMOD*, 2015.

[15] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *SC*, volume 99, 1999.

[16] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *EDBT/ICDT 2011 Workshop on Array Databases*.

[17] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *Trustcom*, 2013.

[18] A. Gonsalves, K. Nakamura, J. Daniels, H.-S. Mao, C. Benedetti, C. Schroeder, C. Tóth, J. van Tilborg, D. Mittelberger, S. Bulanov, et al. Generation and pointing stabilization of multi-GeV electron beams from a laser plasma accelerator driven in a pre-formed plasma waveguide. *Physics of Plasmas*, 22(5), 2015.

[19] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *SSDBM*, 2006.

[20] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. HACC: Extreme scaling and performance across diverse architectures. In *SC*, 2013.

[21] M. Howison, K. Quincey, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre file systems. In *IASDS'10*.

[22] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*, 2015.

[23] Q. Koziol, E. Pourmal, and N. Fortner. RFC: HDF5 virtual dataset. Documentation, The HDF5 Group, 2014.

[24] The Lustre file system. `http://www.lustre.org`.

[25] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. *SIGMOD*, 25(2), 1996.

[26] A. P. Marathe and K. Salem. Query processing techniques for arrays. *VLDBJ*, 11(1), 2002.

[27] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB array data storage manager. *PVLDB*, 10(4), 2016.

[28] N. Rees, H. Billich, A. Götz, Q. Koziol, E. Pourmal, M. Rissi, and E. Wintersberger. Developing HDF5 for the Synchrotron community. *ICALEPCS*, 2015.

[29] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE CG&A*, 10(4), 1990.

[30] D. Robinson and Q. Koziol. Metadata cache issues under SWMR and their solutions. `http://hdf4.org/HDF5/docNewFeatures/SWMR/Design-Metadata-Cache-Issues-Retries-SWMR-v3.pdf`, 2013.

[31] F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. *arXiv:1302.0103*, 2013.

[32] SciDB hardware guideline. `https://www.paradigm4.com/resources/hardware-guidelines/`, 2016.

[33] SciDB reference guide. `https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Reference+Guide`, 2016.

[34] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *ICDE*, 2012.

[35] T. Shimada, T. Tsuji, and K. Higuchi. A storage scheme for multidimensional data alleviating dimension dependency. In *ICDIM*, 2008.

[36] E. Soroush and M. Balazinska. Time travel in a scientific array database. In *ICDE*, 2013.

[37] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: a storage manager for complex parallel array processing. In *SIGMOD*, 2011.

[38] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and SciDB. *CIDR*, 7, 2009.

[39] C. Thaury, F. Quéré, J.-P. Geindre, A. Levy, T. Ceccotti, P. Monot, M. Bougeard, F. Réau, P. d'Oliveira, P. Audebert, et al. Plasma mirrors for ultrahigh-intensity optics. *Nature Physics*, 3(6), 2007.

[40] The HDF5 Group. Single-Writer/Multiple-Reader (SWMR) documentation. `https://support.hdfgroup.org/HDF5/docNewFeatures/SWMR/HDF5_SWMR_Users_Guide.pdf`, 2015.

[41] A. R. van Ballegooij. RAM: a multidimensional array DBMS. In *EDBT*, 2004.

[42] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A novel MapReduce-like framework for multiple scientific data formats. In *CCGrid*, 2012.

[43] Y. Wang, A. Nandi, and G. Agrawal. SAGA: array storage as a DB with support for structural aggregations. In *SSDBM*, 2014.

[44] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over HDF5. In *CCGrid*, 2013.

[45] WCRP Coupled Model Intercomparison Project, Phase 5. *CLIVAR Exchanges Special Issue, No. 56*, 15(2), 2011.

[46] N. Widmann and P. Baumann. Efficient execution of operations in a DBMS for multidimensional arrays. In *SSDBM*, 1998.

[47] K. Wu. FastBit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16(1), 2005.

[48] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: bridging the gap between science and relational DBMS. In *IDEAS*, 2011.

[49] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity join over array data. In *SIGMOD*, 2016.

[50] Z. Zhao. Automatic library tracking database at NERSC. `https://www.nersc.gov/assets/altdatNERSC.pdf`, 2014.