

ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs

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

[★] *The Ohio State University* [†] *Lawrence Berkeley National Laboratory* [§] *Paradigm4, Inc.*
 {xing.136, floratos.1, blanas.2}@osu.edu {sbyna, prabhat, kwu}@lbl.gov pbrown@paradigm4.com

Abstract—

Scientists are increasingly turning to datacenter-scale computers to analyze massive arrays. Despite decades of database research that extols the virtues of declarative query processing, scientists still write, debug and parallelize imperative HPC programs even for the most mundane queries. This impedance mismatch is due to the cumbersome and costly data format conversions that are needed to use scientific data management tools, such as SciDB, in an HPC setting. Our goal is to make declarative array manipulations from SciDB interoperable with imperative, file-centric analyses from HDF5-based programs.

This paper describes ArrayBridge, a bi-directional array view mechanism for the HDF5 file format, that allows scientists to use SciDB, TensorFlow and HDF5-based analysis code in the same file-centric pipeline without converting between file formats. In addition to fast querying over HDF5 array objects, ArrayBridge produces arrays in the HDF5 file format as easily as it can read from it. ArrayBridge also supports time travel queries from imperative codes through the unmodified HDF5 API, and automatically deduplicates between versions for space efficiency. Our performance evaluation in a large scientific computing facility shows that ArrayBridge exhibits statistically indistinguishable performance and I/O scalability to the native SciDB storage engine and is 3× faster than TileDB.

I. INTRODUCTION

Scientists are increasingly turning to datacenter-scale computers to understand phenomena that would otherwise be impossible or intractable to approach experimentally. Scientific advances in domains as diverse as plasma simulation [1], cosmology [2], and climate modeling [3] require processing many terabytes of array data. Many of these arrays are stored in scientific file formats like HDF5 [4] and netCDF [5]. Usage data from petaflop-scale computing facilities corroborates that HDF5 and netCDF remain among the most popular building blocks for scientific computing today [6].

Scientific data management systems like SciDB [7] have not been widely adopted in large scientific computing facilities. One challenge is that ingesting data into SciDB is slow: parallel loading of modestly-sized (~100 GiB) arrays in SciDB takes hours and uses at least 4× more space than the size of the input for data staging. Furthermore, once the data is loaded in the system it can no longer be accessed by HPC tools, such as ParaView for visualization [8], Catalyst for *in situ* processing [9], FastQuery for indexing [10], nor by custom C or Fortran code that is essential for many problem-specific optimizations. Modern analytics frameworks

such as TensorFlow [11] and Theano [12] are increasingly being used in scientific data processing pipelines along with more traditional HPC software. Prior research corroborates that it is non-trivial to port an entire scientific data processing pipeline to one system [13].

We argue for stronger interoperability between the scientific database system and the tools outside it. This would allow scientists to mix and match storage, processing and analytics software as they see fit for the problem at hand. ArrayBridge is the first step towards realizing this vision. ArrayBridge implements a bi-directional view mechanism for multi-dimensional array objects in the HDF5 file format. ArrayBridge exposes an operator-centric interface, which makes SciDB, TensorFlow, and legacy C or Fortran code *seamlessly interoperable* by producing, accessing and updating versioned array objects directly in the HDF5 file format. By directly interfacing to SciDB and TensorFlow, ArrayBridge does not require scientists to wait for hours for a dataset to be converted to another format before issuing the first query.

Previous research has considered how database systems can bypass data loading [14], [15], [16], [17], [18], but has largely focused on achieving good read performance. Scientists, however, also desire to serialize objects in the database to array formats like HDF5, and do so efficiently. Yet, HDF5 has an inherent single-writer, multiple-readers (SWMR) design constraint [19], [20]. This creates a scalability bottleneck because a single writing process cannot utilize the massive I/O concurrency of the parallel file system in a modern scientific computing facility. ArrayBridge efficiently materializes array objects in the HDF5 file format by writing in parallel through a *virtual view*. The view controls the I/O concurrency by partitioning parallel writes into many independent streams. This 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.

Finally, many scientific processes are iterative and naturally produce versioned array objects. Although SciDB supports time travel queries, scientists lack a principled way of accessing past object versions when the data are stored in an HDF5 file instead. Saving versions as different objects creates bloated files that duplicate unmodified chunks. ArrayBridge implements multi-versioning and discards duplicate array chunks in HDF5 files, which in turn conserves I/O bandwidth and storage

space. ArrayBridge permits time travel queries from existing applications (frequently version-oblivious C or Fortran code) through the unmodified HDF5 file format API. Versions are reconstructed in an opaque manner to applications by carefully using the *virtual dataset* feature of HDF5. Under the hood, ArrayBridge deduplicates common regions between versions and reduces the storage footprint of the HDF5 file.

Our extensive experimental evaluation in NERSC, a large scientific computing facility, shows that ArrayBridge completely bypasses the cumbersome multi-hour data loading process of SciDB for TiB-sized array datasets. ArrayBridge exhibits statistically indistinguishable I/O scalability and performance to the native SciDB storage engine and is about $3\times$ faster than TileDB. In addition, ArrayBridge produces arrays in the HDF5 format as efficiently as SciDB serializes the database in its proprietary format for backup. ArrayBridge is shown to have near-optimal speedup up to 128 nodes and it can evaluate an ad-hoc aggregation query on a 38 TiB HDF5 dataset in minutes.

II. BACKGROUND

A. The SciDB database system

SciDB organizes data into *multi-dimensional arrays*. Every cell in a SciDB array contains one or more attributes (values), similar to a tuple in relational databases. An array is stored and processed in *chunks* using the *regular chunking* strategy [21]. SciDB compresses chunks using Run Length Encoding (RLE).

SciDB adopts a shared-nothing model, and partitions the chunks across all instances in a cluster. Metadata is stored in the SciDB catalog, which is a centralized PostgreSQL database. 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 SciDB coordinator parses an AQL or AFL query to generate a query plan for optimization. The coordinator also orchestrates the evaluation of partial query fragments among instances, and returns the final result to the user. The query plan is a graph of array operators. During the optimization phase, each operator reports the schema of its output array. Every operator exposes a chunked-based iterator interface that returns one chunk in the output upon request.

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. Therefore, loading an r -dimensional array with a attributes requires storage space for a table with $r + a$ attributes. The second step is a *redimension* operation that converts this table to an r -dimensional array.

Supported file formats: Plain text formats like comma-separated value (CSV) files are very versatile, but they are also verbose and 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 [22].

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 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.

Saving SciDB objects to disk: A user saves SciDB objects on disk using the `save()` operator. SciDB 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.

B. The HDF5 file format

The HDF5 format [4] 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], [24] 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 $m_1, m_2, m_3, \dots, m_n$, that describe where the actual data is stored. A mapping is represented as a tuple $\langle d, src, dst \rangle$; 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

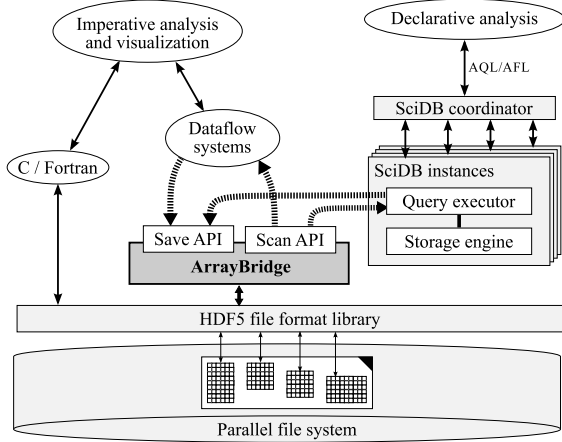


Fig. 1: Through ArrayBridge, scientists intermingle declarative and imperative analyses on the same dataset. ArrayBridge uses the HDF5 library for storage and is integrated with the SciDB query engine.

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.

III. SYSTEM OVERVIEW

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

Users can 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

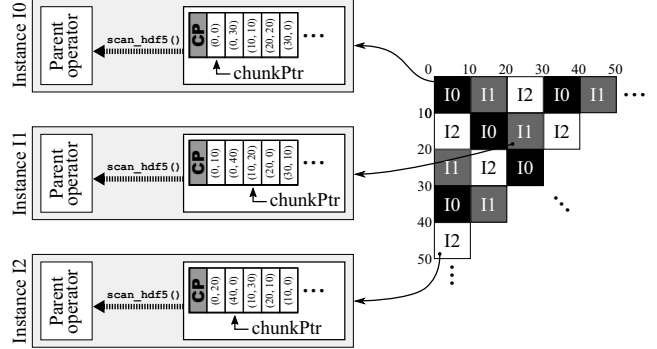


Fig. 2: An operator-centric runtime process data directly from external HDF5 arrays using the `scan_hdf5` operator in ArrayBridge.

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.

IV. READING ARRAY OBJECTS IN THE HDF5 FORMAT

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 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.

A. Design of the Scan operator

The internal interface to the `scan_hdf5()` operator consists of two functions: `Start(obj, attr)` and `Next()`. 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.

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 instances. Database systems (including SciDB) store assignment information in a 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 behind 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.

B. Implementation of the Scan Operator

The pseudocode of the `scan_hdf5()` implementation 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

Algorithm 1: The `scan_hdf5` implementation for SciDB.

```
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 pi in dataset d do
6     | if  $\mu(p_i) = \text{this instance}$  then add pi into CP
7   end
8   chunkPtr ← first element in CP
9   c ← a new RLE chunk with unique elements
10 end
11 Function Next()
12 | r ← region at chunkPtr to read from
13 | call H5Dread(d, r, c) to read data into chunk c
14 | advance chunkPtr
15 | return c
16 end
```

different chunk assignment algorithms [25]. 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 *j* to its local *CP* if and only if $\mu(j) = i$. 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.

SciDB operators exchange data in compressed run-length encoded (RLE) format. We found that compressing HDF5 data into the RLE representation after every read was a performance bottleneck. This predominantly impacts dense arrays that HDF5 is designed for, which do not compress well with RLE. Hence, we decided against compressing data using RLE in ArrayBridge. Each scan operator in ArrayBridge creates a single RLE-compressed chunk *c* where each element is unique, and then reuses this chunk in every call to `H5Dread()`.

V. SAVING ARRAY OBJECTS IN THE HDF5 FORMAT

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 array 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.

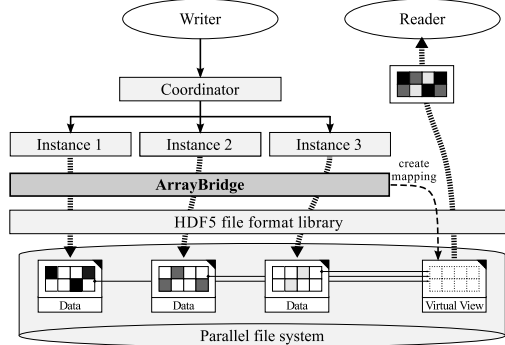


Fig. 3: The virtual view mechanism bypasses the single-writer limitation of the HDF5 file format by directing write streams from different instances into separate objects. Existing imperative analyses access a single object through the view.

A. Balancing efficiency and interoperability

Serializing array data into the HDF5 format exposes a dilemma between *writing efficiency* and *interoperability*. To write data in parallel, one can 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 instance. This fundamental Single-Writer, Multiple-Readers (SWMR) design constraint of HDF5 is a roadblock to parallel writing and it prevents any analytical query engine from taking advantage of the highly concurrent I/O subsystems found in modern scientific computing facilities.

ArrayBridge supports a novel writing mode, the *Virtual View* mode, to allow for explicit control of the I/O concurrency regardless of the number of files being accessed. This avoids the artificial dilemma between writing efficiency or interoperability with the *Serial* and *Partitioned* writing modes of SciDB. The *Virtual View* mode utilizes the virtual dataset feature of the HDF5 format, which presents the data in several HDF5 files as a single dataset to the application accessing it. As shown in Figure 3, in the *Virtual View* mode each instance writes data to separate HDF5 files, and then a virtual HDF5 dataset maps these files into one logical object. 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.

B. Serializing arrays into HDF5

ArrayBridge exposes a `save()` operator to support the HDF5 format. This allows a user to write data back to external HDF5 files. ArrayBridge supports *Serial* and the *Partitioned* writing modes with 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 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 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 instances. We evaluate the performance trade-offs of these two techniques in Section VI-C.

C. 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 three techniques for saving a versioned dataset: Full Copy, Chunk Mosaic and Delta.

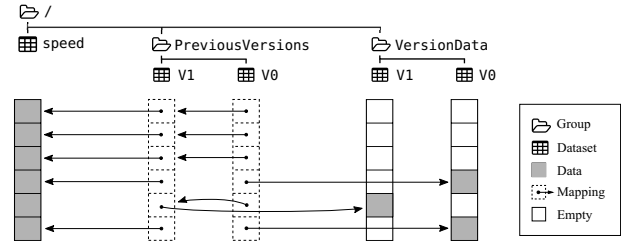


Fig. 4: An example of the Chunk Mosaic technique. The latest version is the fully-materialized dataset `speed`. Old versions are virtual datasets in the `PreviousVersions/` group.

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`, ArrayBridge 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 many analytical engines do not readily 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. We acknowledge that tighter integration with a particular query engine can avoid reading the entire chunk to discover the differences. (For example, SciDB can convey precisely which cells were updated through the `insert()` operator.) We plan to explore opportunities for tighter integration as part of our future work.

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`.

Chunk Mosaic 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.

Delta: A disadvantage of the Chunk Mosaic technique is that it will materialize the entire chunk even when an update has only changed a single cell in this chunk. The Delta technique is a refinement of the Chunk Mosaic technique that only stores the cells that changed (the *delta*) back on disk.

The Delta technique only stores the modified cells by carefully using an HDF5 *chunk filter* to discard cells that were not updated when materializing the previous version. (Recall that the ArrayBridge always materializes the latest version for efficiency reasons.) The HDF5 library supports chunk filters as a mechanism to allow user-defined data manipulations on the I/O path of the HDF5 library. Chunk filters are loaded dynamically and sit between the internal storage manager in HDF5 and the I/O layer. A popular use of the chunk filter functionality has been applying domain-specific storage optimizations and compression. ArrayBridge uses a chunk filter to sequentially store the updated cells and then truncates the buffer that will be forwarded to the I/O layer for storage. This allows ArrayBridge to issue I/O requests that are proportional in size to the number of updated cells in each chunk.

This I/O-oriented optimization comes at the expense of CPU cycles when it comes to creating the mappings that will allow version-oblivious code to reconstruct the historic version of an object. Whereas the Chunk Mosaic technique would create as many mappings as chunks (as shown in Figure 4, under the group `PreviousVersions`), the Delta technique may need one mapping per cell if the updated region cannot be expressed as a contiguous or strided hyper-rectangle. We evaluate this performance trade-off in Section VI-C.

Reconstructing old versions: Old versions are reconstructed automatically by the HDF5 library when one accesses a dataset in the `PreviousVersions` group. When using the Chunk Mosaic technique, the HDF5 library follows the virtual mappings to read the actual chunk. When using the Delta technique, the chunk filter reads the latest materialized chunk and the modified cells from the disk and combines them to the actual chunk using the information in the virtual mapping. We refer interested readers to the virtual dataset design document for more information how the reconstruction process is implemented in HDF5 [23]. The strength of versioning in ArrayBridge is that this reconstruction procedure is opaque to programs that access the HDF5 file: old versions appear as different datasets to programs and can be read without any modification to the program or the HDF5 library.

VI. EXPERIMENTAL EVALUATION

This section evaluates the performance of ArrayBridge. We consider the following questions:

- (§VI-B) How does ArrayBridge perform when reading data from a parallel file system in a typical HPC environment? How much time and space does loading take in SciDB?

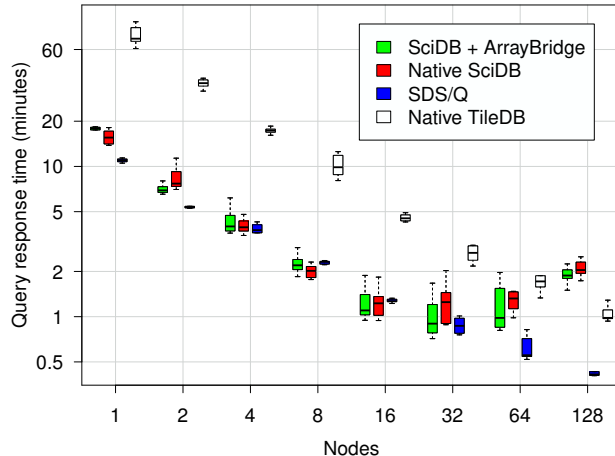


Fig. 5: Response time when aggregating a 1.5 TiB dataset.

- (§VI-C) 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?
- (§VI-D) Does ArrayBridge scale to a real multi-TiB simulation dataset that would be infeasible to load in SciDB?

A. 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” CPUs and 64 GiB of memory. File storage is provided via the Lustre [26] 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 stripes are distributed to the requested OSTs in a round-robin fashion. The *stripe count* is the number of OSTs a file is distributed to. 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 with the SciDB 15.12 Community release. We configure 8 instances per node and a 16 MiB chunk size according to the SciDB guidelines. The HDF5 datasets are chunked using the same chunk size (16 MiB). Unless specified otherwise, we follow the facility-wide I/O tuning recommendations that suggest striping each file to 72 Lustre OSTs with a 1 MiB Lustre stripe size.

One challenge in reporting performance results from large HPC facilities is that the computers are always busy with multiple concurrent jobs. This brings high variability in I/O performance. When the variability is significant, we report performance in a box plot to convey the effect of variance.

B. Scan performance

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

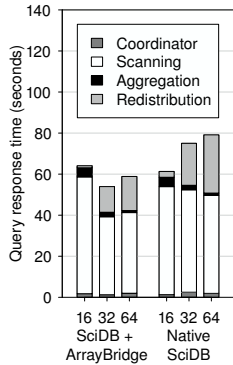


Fig. 6: SciDB time breakdown when aggregating 1.5 TiB.

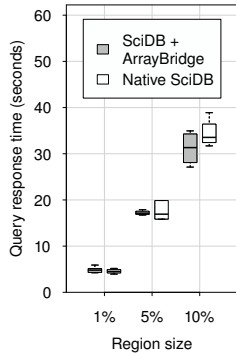


Fig. 7: Impact of selectivity on response time.

Declarative analysis with ArrayBridge: The first question is what is the scan performance of ArrayBridge to scan a one-dimensional dataset of 192 billion double numbers (approximately 1.5 TiB data). For comparison purposes with prior work, we use SciDB [7], the SDS/Q system [16] and TileDB [27]. Figure 5 shows the query response time as the number of nodes varies from 1 to 128.

All implementations scale near perfectly from 1 to 32 nodes. In this region, TileDB takes at least $3\times$ more time than the other systems. TileDB organizes arrays into fragments that are stored in separate directories in the underlying file system. Profiling shows that this slowdown is because of the overhead of creating and closing fragments, which triggers excessive I/O to different Lustre OSTs for metadata operations. SDS/Q outperforms SciDB and ArrayBridge with 1 and 2 nodes because the workload is CPU bound, but the performance equalizes as the workload becomes I/O bound. The ArrayBridge peak I/O performance is about 28 GiB/s with 32 nodes.

ArrayBridge and SciDB stop scaling beyond 32 nodes, while SDS/Q and TileDB continue to do so. Profiling SciDB confirms that this slowdown is not caused by ArrayBridge. Figure 6 shows the time breakdown when using 16, 32, and 64 nodes. The culprit for the increased query response time is the redistribution cost: it takes far more time to merge partial aggregates in SciDB than an `MP_I_Reduce` operation in SDS/Q.

Overall, the performance of ArrayBridge is statistically indistinguishable from the native SciDB engine.

Performance with selective queries: Some queries selectively access data. 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 subset of a dense array.

Imperative analysis with ArrayBridge: An important question is whether the scan performance can be realized for

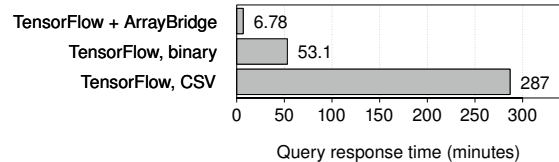
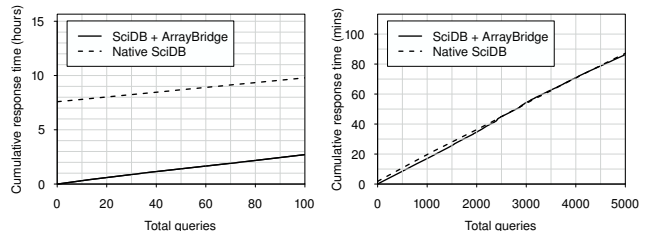


Fig. 8: TensorFlow comparison with an 8 billion element array.



(a) Aggregating a 1 TiB dataset that does not fit in memory. (b) Aggregating a 16 GiB dataset that fits in the SciDB buffer pool.

Fig. 9: Cumulative time to load and complete multiple queries.

imperative programs that read data in the HDF5 format. We turn to TensorFlow [11] to demonstrate this. Users describe computations in TensorFlow as a dataflow graph where they specify both the nodes (operators) of the graph and the edges (data dependencies) between nodes. The unit of data movement between nodes in TensorFlow is a multi-dimensional array (a tensor) which is returned directly by the Scan API of ArrayBridge.

An evaluation of the scan performance of ArrayBridge in TensorFlow reveals that the I/O performance of the native TensorFlow scan kernels can be slower by two orders of magnitude. Figure 8 shows the query response time when aggregating an array with eight billion numbers in 16 MiB chunks from a single thread. Variance in this experiment is less than $\pm 10\%$. Scanning through ArrayBridge completes this query in about 7 minutes. The same query takes $8\times$ longer when reading from a binary file and $42\times$ longer when reading from a CSV file using the TensorFlow operators. This comparison does not include data conversion times from HDF5 into binary or CSV, which would have increased the time to the answer even further. We thus conclude that ArrayBridge provides compelling I/O performance over the native TensorFlow scan which is not optimized for data-intensive analyses.

Time to insight experiment: The evaluation now considers the data loading time in SciDB and how this impacts workloads with multiple queries. We repeatedly aggregate the same dataset using both ArrayBridge and native SciDB in an 8-node cluster. For ArrayBridge, we query the HDF5 dataset directly; for native SciDB, we load the binary dataset in parallel. We 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 9a and 9b report the cumulative query response time of loading and evaluating the first n queries. ($n = 0$ reflects the dataset loading time.)

For the 1 TiB dataset (Figure 9a), it takes more than 7.5 hours for the native SciDB to load the data, redimension it,

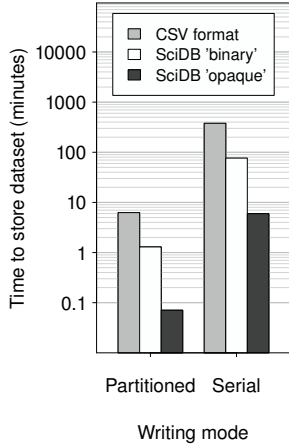


Fig. 10: Time to save a 64 GiB dataset in different formats.

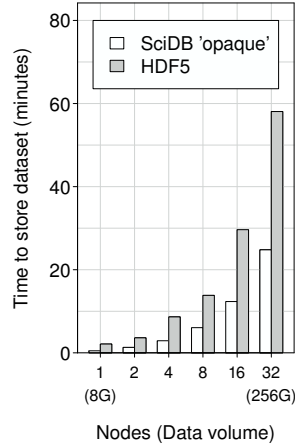


Fig. 11: Time to save 8 GiB/node in different formats with the *Serial* writing mode.

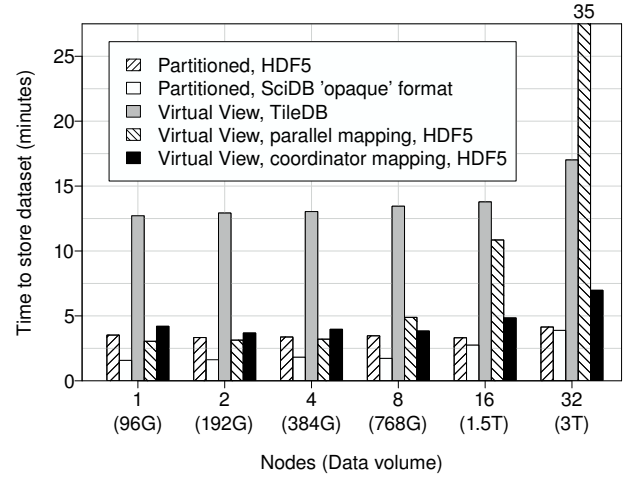


Fig. 12: Time to save 96 GiB/node using the *Partitioned* and the *Virtual View* writing modes in HDF5, TileDB and SciDB’s proprietary “opaque” formats.

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.

This holds even for the 16 GiB dataset (Figure 9b) which fits entirely in the buffer pool. (Recall that the `scan_hdf5()` operator issues I/O requests to the HDF5 file on every call, whereas native SciDB performs all I/O through the buffer pool.) We conclude that caching and prefetching by the parallel filesystem can be as effective as a buffer pool in curtailing redundant I/O to cold storage for a full scan access pattern.

Another advantage of querying HDF5 data directly in its native format is reducing the space overhead of loading. As described in Section II, importing an array into SciDB requires importing the original data into a flat array with coordinate information, and then converting it to the redimensioned SciDB format. For a one-dimensional array, this occupies 3× more space than the original data.

C. Save performance

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 II 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.

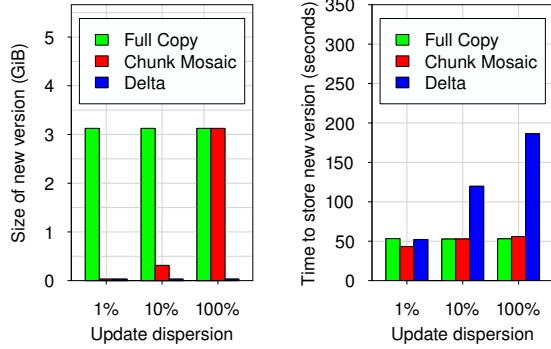
Figure 10 reports the median query response time of each format/mode combination. (Note that the vertical axis is logarithmic.) As the figure indicates, the CSV and the SciDB ‘binary’ formats are very slow to write into. Even with 8 nodes writing CSV data in parallel, the writing throughput per node is only ~ 20 MiB/s due to the overhead of converting doubles into a text representation. The SciDB binary format is almost 5× faster as it avoids text conversions and merely

reconstructs the binary on-disk representation from the RLE encoding. The ‘opaque’ format is 10× 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



(a) Size of new version. (b) Time to store new version.
 Fig. 13: Space and time needed to update a 3.1 GiB dataset.

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 dataset.

To evaluate the efficiency of the HDF5 virtual mapping mechanism, we also compare the performance of saving in the HDF5 format with TileDB [27], which also supports viewing a partitioned array as a single object. In TileDB, different processes can write to different fragments (files) 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. About 60% of the time is spent on metadata operations. This is corroborated by the observation that merely listing the contents of the directory that stores a 1.5 TiB array in the TileDB format using `\ls -l` takes more than 30 minutes as all Lustre OSTs are contacted to retrieve the size of every fragment.

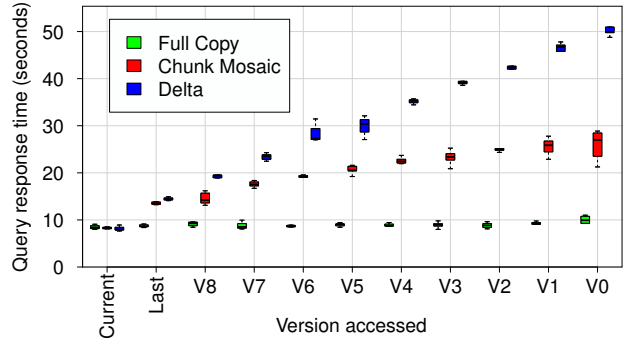


Fig. 14: Time to read an old version.

Time Travel: The evaluation now considers how effective is deduplicating versioned datasets for time travel queries when compared with the Full Copy technique. We use a synthetic 3.1 GiB dataset of double numbers with a 16 MiB chunk size. To create a new version, we update a fixed number of N elements in a strided access pattern, where $N = 1\%$ of the total elements in the array. We change the dispersion of the updates by scattering them to 1% of the chunks, 10% of the chunks, and to the entire dataset (100%).

Figure 13a compares the size of a new version. An update using the Full Copy technique occupies as much space as the original, since it always duplicates the entire dataset. In contrast, the space usage of the Chunk Mosaic is proportional to the number of chunks being updated. The Delta technique is even more storage-efficient as it only materializes the updated cells regardless of the dispersion of the updates. Figure 13b compares the time to store a new version. We note that the Chunk Mosaic technique has comparable performance for a significant space advantage over Full Copy. The Delta technique works well for very few updates, but its performance deteriorates due to the time spent in creating mappings. The Delta technique would take significantly longer if cells were selected uniformly at random.

Figure 14 reports the time to read an old version after 10 updates that change 1% of the elements and scatter the updates to 10% of the chunks. Reading the current version with any technique requires no reconstruction, and neither does reading any version with Full Copy. Delta has higher reconstruction cost over Chunk Mosaic as it operates on individual cells versus entire chunks. Overall, the time to read an old version is proportional to the versions that need to be accessed to perform the reconstruction. The time to reconstruct a version would increase if the updates were more dispersed, as more chunks would need to be accessed. These results suggest that workloads that frequently perform time travel queries should intersperse fully-materialized (“Full Copy”) versions in the version chain. The distance to these fully-materialized versions imposes an upper bound on the number of versions needed for reconstruction and hence limits the worst-case access time.

D. ArrayBridge with particle-in-cell (PIC) simulation data

We evaluate the performance of ArrayBridge when analyzing a scientific dataset that would be infeasible to load into SciDB

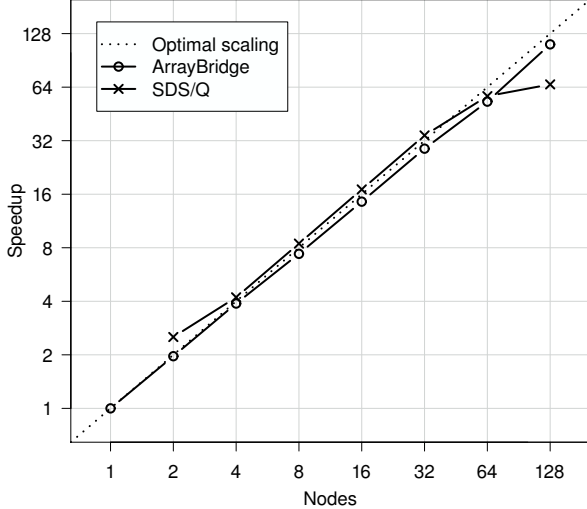


Fig. 15: Speedup when analyzing 38 TiB of Particle-In-Cell simulation data in HDF5 with ArrayBridge and SDS/Q [16].

due to its massive size. In addition, we compare with the SDS/Q system for querying HDF5 data from prior work [16].

The workload comes from the plasma physics domain. The scientific application uses VPIC [1], a general purpose particle-in-cell simulation code, for modeling kinetic plasmas with billions of particles. Studying micro-phenomena requires simulating a 3D field with very high resolution. For this problem, one simulation step produces results that are stored in a single 38 TiB HDF5 file on the parallel file system which is striped to 244 Lustre OSTs.

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 analysis 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 analysis can be succinctly conveyed in a SciDB AFL query that directly accesses the HDF5 data via ArrayBridge.

The scalability of this aggregation query over the output of a single simulation step (a 38 TiB array dataset) is shown in Figure 15 as a function of the single-node performance of ArrayBridge. ArrayBridge on a single node takes more than a day to return the final answer. ArrayBridge scales near-optimally up to 128 nodes, returning the final answer in about 17 minutes. Overall, ArrayBridge aggregates 38 TiB of array data in minutes for this AFL query, whereas loading and redimensioning this dataset using a SciDB cluster of the same size would have taken more than a week.

SDS/Q is on average about 5% faster than ArrayBridge for this query up to 32 nodes. However, SDS/Q stops scaling linearly after 64 nodes and becomes slower than ArrayBridge at 128 nodes. We remark that SDS/Q is a *relational* query engine that is limited to SQL-like queries. Array-centric operations (like matrix multiplication) are not supported in SDS/Q but can easily be expressed as AQL/AFL queries in ArrayBridge.

VII. DISCUSSION

Error handling and fault tolerance: In a nutshell, ArrayBridge propagates errors and does not itself dictate any error handling strategy. It is up to the caller of the interface to determine what to do when a failure happens.

SciDB is designed to tolerate failures as long as a chunk can be accessed by at least one node. If an entire node fails, this is detected by the SciDB coordinator through a heartbeat mechanism and the query will restart. Because `scan_hdf5()` and `save()` retrieve running SciDB instances at query time, they assign chunks only to healthy nodes. The `save()` operator requires manual removal of a partially-materialized version in the HDF5 file on restart. ArrayBridge also exposes I/O errors that may result from the HDF5 library or the parallel file system itself. Currently, such errors are propagated to the SciDB coordinator which in turn aborts the entire query.

In TensorFlow, the user specifies how the operators are distributed among compute nodes. Both node failure and I/O errors would result in a failure of the entire query. The user can then restart the query and possibly alter the distribution of operators to avoid failed compute nodes. Imperative HPC programs are frequently ad-hoc codes with no fault tolerance provisions. Errors are handled by calling `MPI_Abort()` to stop the entire program.

Replacing the SciDB storage engine with ArrayBridge: The experimental results demonstrate that ArrayBridge performs competitively to be considered as a possible storage backend for SciDB. A natural question to ask is what are the limitations of HDF5 compared to the native SciDB storage engine.

The main weakness of using HDF5 as a replacement for the SciDB storage engine is that it does not work in a shared-nothing cluster where nodes do not share a parallel filesystem. Installing the popular Hadoop distributed filesystem is not a solution, as its append-only nature does not permit writes and updates to an HDF5 file. In addition, the HDF5 format does not support features that are primarily geared towards enterprise environments and the cloud. These include user authentication and permission management, transactional storage to avoid data corruption, and data replication for fault-tolerance in commodity clusters. Finally, HDF5 is optimized for dense low-dimensionality arrays but many problems naturally produce sparse arrays of high dimensionality. It is unlikely that HDF5 would perform nearly as well for sparse workloads.

ArrayBridge preserves the flexibility of using either HDF5 or the native SciDB storage format if it is advantageous to the workload. We consider the hybrid approach taken by ArrayBridge more palatable to users compared with exclusively using a single data storage format.

VIII. RELATED WORK

Array database systems

Database systems can store multi-dimensional datasets by materializing array offsets as attributes of a relational table [28]. Some solutions build new primitives to express array operations within the DBMS [29], [30], while SciQL [31] extends SQL to support the array data model. Also, new

query languages have been proposed for querying arrays declaratively. AQL [32] is one of the earliest array query languages. AML [33] expresses array operations as a combination of operators, each of which reads one or more arrays and returns an array. In contrast, SciDB [7] has been designed for arrays: all operators accept and return arrays without additional extensions.

Another research topic for array database systems is storage. Soroush et al. investigate different storage management and chunking strategies for array storage and propose a two-layer chunking strategy [21]. 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 [34]. SciDB uses a similar chunk-based storage as ArrayStore [21], 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 [4] and NetCDF [5] 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 [35] or biological images [36] 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 array processing programs to improve I/O performance [35], [37]. TileDB proposes using *flexible tiling* to address the problem of accessing sparse data for efficient writes [27].

Array versioning

The ability to query earlier versions of a database has been extensively studied in the context of temporal databases that operate on two notions of time, valid time and transaction time [38]. SQL Server supports time-travel queries on the transaction time dimension to access prior states of the database [39]. Prior research has also investigated versioned array storage. Soroush et al. propose versioned array storage to support both querying a specific array version and extracting the history of an array object [40]. Seering et al. use a materialization matrix to determine which version to materialize while storing the remaining versions as deltas [41]. DataHub [42] extends the idea of version control to entire relational datasets for data-intensive collaborations. Bhattacharjee et al. investigate the trade-offs between storage size and recreation time for versioning techniques [43]. These versioning techniques, however, are not integrated with HDF5 and do not reconstruct old versions in an opaque manner for version-oblivious applications.

In situ processing

Prior research has considered how data management systems can bypass the onerous data loading process. A non-exhaustive list includes *in situ* querying of CSV files PostgreSQL [14]; symbiotically querying TIFF data with MonetDB in Data Vaults [15]; transparent loading with ScanRAW [17]; just-

in-time query processing in ViDa [18]; and processing data in the Hadoop file system [44], [45]. This prior work has predominantly focused on read performance, and does not consider how to efficiently materialize data in an array format.

Recent work has investigated *in situ* query mechanisms for data in the HDF5 format. SDS/Q [16], [46] introduces a lightweight relational query engine to directly process data in the HDF5 format, in addition to using bitmap-based indices [10], [47] to accelerate highly-selective queries. In the context of MapReduce, analysis frameworks have proposed querying HDF5 and NetCDF data directly [48], [49]. ParaView Catalyst is an imperative analysis framework for *in situ* processing and visualization of simulation data [9]. ParaView Catalyst operates on data volumes that may be infeasible to materialize on disk. In contrast, a core assumption behind ArrayBridge is that the data of interest are already materialized on disk. ArrayBridge supports analyses that cannot be performed *in situ* because of their complexity, such as array join processing [50] and similarity-based joins [51].

Beyond file-centric interfaces

ArrayBridge uses the HDF5 file format for interoperability between declarative and imperative analyses in a file-centric HPC pipeline. Tighter integrations between imperative and declarative analysis kernels are feasible. Weld proposes a common intermediate representation to optimize across disjoint libraries without materializing data on disk [52].

IX. CONCLUSIONS AND FUTURE WORK

Querying massive binary arrays in HDF5 format from SciDB or TensorFlow requires is a multi-hour format conversion process, even if it is done in parallel. ArrayBridge lets scientists run SciDB and TensorFlow queries over multi-TiB array datasets in the HDF5 format without performing any data conversion. In addition to fast scans, ArrayBridge can materialize 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 the virtual view mechanism in ArrayBridge. Finally, ArrayBridge efficiently stores versioned objects by deduplicating unmodified chunks between versions. Past versions are accessible via the standard HDF5 API, which enables time travel queries from version-oblivious imperative code.

Our future work looks at workloads that use the time travel feature of ArrayBridge extensively. Astronomical surveys such as ASAS-SN [53] and PTF [54] are promising candidates, as time travel queries naturally occur in the detection of supernovae. Another promising direction is considering how to bring the features of ArrayBridge to the cloud and the Hadoop distributed file system. SciDB and TensorFlow can run in a cloud environment, but many array-centric HPC applications are not readily portable. We envision that ArrayBridge will mediate I/O to seamlessly allow these applications to execute in a shared-nothing environment. ArrayBridge is open source and the code is available at: <http://code.osu.edu/arraybridge>

ACKNOWLEDGEMENTS

We acknowledge the contributions of Damodar Yekkuluri in running the VPIC experiment. This work was supported in part by a Google Research Faculty Award; the U.S. National Science Foundation under grants III-1422977, III-1464381, CNS-1513120, SHF-1747447; the Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources and data generated from resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

- [1] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultra-high performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.
- [2] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "HACC: Extreme scaling and performance across diverse architectures," in *SC*, 2013.
- [3] "WCRP Coupled Model Intercomparison Project, Phase 5," *CLIVAR Exchanges Special Issue*, No. 56, vol. 15, no. 2, 2011.
- [4] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *SC*, vol. 99, 1999.
- [5] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE CG&A*, vol. 10, no. 4, 1990.
- [6] Z. Zhao, "Automatic library tracking database at NERSC," <https://www.nersc.gov/assets/alt/NERSC.pdf>, 2014.
- [7] P. G. Brown, "Overview of SciDB: large scale array storage, processing and analysis," in *SIGMOD*, 2010.
- [8] J. Ahrens, B. Geveci, and C. Law, "ParaView: An end-user tool for large data visualization," *Visualization Handbook*, Elsevier, 2005.
- [9] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion *et al.*, "The ParaView coprocessing library: A scalable, general purpose in situ visualization library," in *IEEE LDAV*, 2011, pp. 89–96.
- [10] 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.
- [11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean *et al.*, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016, pp. 265–283.
- [12] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016.
- [13] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska *et al.*, "Comparative evaluation of big-data systems on scientific image analytics workloads," *PVLDB*, vol. 10, no. 11, 2017.
- [14] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "NoDB: efficient query execution on raw data files," in *SIGMOD*, 2012.
- [15] M. Ivanova, M. Kersten, and S. Manegold, *Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 485–494.
- [16] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *SIGMOD*, 2014.
- [17] Y. Cheng and F. Rusu, "SCANRAW: A database meta-operator for parallel in-situ processing and loading," *TODS*, vol. 40, no. 3, 2015.
- [18] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki, "Just-in-time data virtualization: Lightweight data management with ViDa," in *CIDR*, 2015.
- [19] D. Robinson and Q. Koziol, "Metadata cache issues under SWMR and their solutions," 2013.
- [20] The HDF5 Group, "Single-Writer/Multiple-Reader (SWMR) documentation," 2015.
- [21] E. Soroush, M. Balazinska, and D. Wang, "ArrayStore: a storage manager for complex parallel array processing," in *SIGMOD*, 2011.
- [22] "SciDB reference guide," 2016.
- [23] Q. Koziol, E. Pourmal, and N. Fortner, "RFC: HDF5 Virtual Dataset," The HDF Group, Tech. Rep. RFC THG 20130424.v9, December 2014.
- [24] N. Rees, H. Billich, A. Götz, Q. Koziol, E. Pourmal, M. Rissi, and E. Wintersberger, "Developing HDF5 for the Synchrotron community," *ICALEPCS*, 2015.
- [25] F. Rusu and Y. Cheng, "A survey on array storage, query languages, and systems," *arXiv:1302.0103*, 2013.
- [26] "The Lustre file system," <http://www.lustre.org>.
- [27] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The TileDB array data storage manager," *PVLDB*, vol. 10, no. 4, 2016.
- [28] P. Baumann, "Management of multidimensional discrete data," *VLDBJ*, vol. 3, no. 4, 1994.
- [29] A. R. van Ballegooij, "RAM: a multidimensional array DBMS," in *EDBT*, 2004.
- [30] N. Widmann and P. Baumann, "Efficient execution of operations in a DBMS for multidimensional arrays," in *SSDBM*, 1998.
- [31] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes, "SciQL: bridging the gap between science and relational DBMS," in *IDEAS*, 2011.
- [32] L. Libkin, R. Machlin, and L. Wong, "A query language for multidimensional arrays: design, implementation, and optimization techniques," *SIGMOD*, vol. 25, no. 2, 1996.
- [33] A. P. Marathe and K. Salem, "Query processing techniques for arrays," *VLDBJ*, vol. 11, no. 1, 2002.
- [34] T. Shimada, T. Tsuji, and K. Higuchi, "A storage scheme for multidimensional data alleviating dimension dependency," in *ICDIM*, 2008.
- [35] 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.
- [36] M. T. Dougherty, M. J. Folk, E. Zadok, H. J. Bernstein, F. C. Bernstein, K. W. Eliceiri, W. Bengler, and C. Best, "Unifying biological image formats with HDF5," *Comm. of the ACM*, vol. 52, no. 10, 2009.
- [37] M. Howison, K. Quincey, D. Knaak, J. Mainzer, and J. Shalf, "Tuning HDF5 for Lustre file systems," in *IASDS'10*.
- [38] R. T. Snodgrass and I. Ahn, "A taxonomy of time in databases," in *SIGMOD '85*, 1985, pp. 236–246.
- [39] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu, "Immortal DB: Transaction time support for SQL Server," in *SIGMOD '05*. New York, NY, USA: ACM, 2005, pp. 939–941.
- [40] E. Soroush and M. Balazinska, "Time travel in a scientific array database," in *ICDE*, 2013.
- [41] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker, "Efficient versioning for scientific array databases," in *ICDE*, 2012.
- [42] 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.
- [43] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," in *VLDB*, vol. 8, no. 12, 2015.
- [44] A. Abouzied, D. J. Abadi, and A. Silberschatz, "Invisible loading: access-driven data transfer from raw files into database systems," in *EDBT*, 2013.
- [45] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, and J. Gramling, "Split query processing in Polybase," in *SIGMOD*, 2013, pp. 1255–1266.
- [46] B. Dong, S. Byna, and K. Wu, "Parallel query evaluation as a scientific data service," in *CLUSTER*, 2014.
- [47] K. Wu, "FastBit: an efficient indexing technology for accelerating data-intensive science," *Journal of Physics*, vol. 16, no. 1, 2005.
- [48] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang, "SciHive: Array-based query processing with HiveQL," in *Trustcom*, 2013.
- [49] Y. Wang, W. Jiang, and G. Agrawal, "SciMATE: A novel MapReduce-like framework for multiple scientific data formats," in *CCGrid*, 2012.
- [50] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *SIGMOD*, 2015.
- [51] W. Zhao, F. Rusu, B. Dong, and K. Wu, "Similarity join over array data," in *SIGMOD*, 2016.
- [52] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkoft, S. P. Amarasinghe, and M. Zaharia, "A common runtime for high performance data analysis," in *CIDR '17*, 2017.
- [53] T. W.-S. Holloien *et al.*, "The ASAS-SN bright supernova catalogue - III. 2016," *Monthly Notices of the Royal Astronomical Society*, vol. 471, no. 4, pp. 4966–4981, 2017.
- [54] A. Rau *et al.*, "Exploring the optical transient sky with the Palomar Transient Factory," *Publications of the Astronomical Society of the Pacific*, vol. 121, no. 886, pp. 1334–1351, 2009.