6-1-2018

# The Next Generation of EMPRESS: A Metadata Management System For Accelerated Scientific Discovery at Exascale

Margaret R. Lawson
*Dartmouth College*

Dartmouth Computer Science Technical Report TR2018-846

# The Next Generation of EMPRESS

## A Metadata Management System For Accelerated Scientific Discovery At Exascale


**Margaret Lawson**

*Advisor: Professor Charles Palmer*
*External Advisor: Jay Lofstead (Sandia National Labs)*

Department of Computer Science
Dartmouth College

This thesis is submitted in partial fulfillment of the requirement for the degree of
*Bachelor of Arts*

June 2018

## Acknowledgements

I would like to thank several people for their part in helping me complete this thesis and for supporting me throughout my time at Dartmouth. First, I would like to thank my thesis committee. In particular, I would like to thank Professor Charles Palmer for serving as my thesis advisor and guiding me through the thesis process. A special thank you to Professor Cormen for serving as a mentor to me these past four years, and for providing a constant source of encouragement and guidance. Thank you also to Jay Lofstead for overseeing and guiding this research over the past year and for serving as a mentor more generally.

I would additionally like to thank my friends and family for their support and encouragement. I could not have done this without you.

# The Next Generation of EMPRESS - A Metadata Management System For Real-Time Scientific Discovery At Exascale


## ABSTRACT

Scientific data sets have grown rapidly in recent years, outpacing the growth in memory and network bandwidths. This I/O bottleneck has made it increasingly difficult for scientists to read and search outputted datasets in an attempt to find features of interest. In this paper we will present the next generation of EMPRESS, a scalable metadata management service that offers the following solution: users can "tag" features of interest and search these tags without having to read in the associated datasets. EMPRESS provides, in essence, a digital scientific notebook where scientists can write down observations and highlight interesting results, and an efficient way to search these annotations. This kind of service is crucial in light of the current I/O bottleneck and represents a more efficient way to explore high-level dataset contents regardless of I/O trends. EMPRESS also provides storage-system independent metadata and a portable way for users to read both metadata and the associated data. EMPRESS offers scalability through use of a set of dedicated, shared-nothing servers. EMPRESS also provides robust fault tolerance and transaction management, which is crucial to supporting workflows.


## 1 INTRODUCTION

Large-scale scientific simulations are a crucial tool for scientific discovery. These simulations allow scientists to derive insights about phenomena that are impractical or impossible to perform physical experiments on. In recent years, supercomputers have become dramatically more computationally powerful as nodes have been scaled up (with more processors and more powerful processors) and computing clusters have been scaled out (with more nodes per cluster). This has allowed scientists to run simulations at increasingly fine-grained spatio and temporal scales [15, 25]. These simulations often run for extended periods of time (such as 24 hours), and at periodic intervals, output data for analysis [26]. As a result, simulations such as S3D combustion [7], XGC edge plasma fusion [20], and GTS core plasma fusion [51] can easily generate datasets in the terabyte to petabyte range from a single run.

Thus, the ability to use finer-grained simulations presents both an opportunity and a challenge. On the one hand, increased granularity makes it more likely a simulation will capture data, such as a new or rare phenomena or a complex trend, that will lead to discovery. On the other hand, these increasingly large datasets are difficult to store, manage, and explore since memory capacity, network bandwidths, and disk bandwidths are growing at a much slower pace [52]. Data size is also the rate limiting factor for analysis and visualization, the primary means of data exploration. Quite simply, a simulation run produces more data than a scientist can load, scan, or visualize. This challenge is compounded by the fact that most features of interest tend to be relatively small, and can easily be missed if scientists use random sampling or related techniques. The problem can therefore be compared to trying to find needles in an increasingly large haystack.

In recent years, many successful technologies have been developed to reduce I/O and storage system pressure such as compression [23], chunking [30], data layout reorganization [14], data staging [3, 10], and asynchronous or adaptive I/O [31]. However, the HPC community has yet to develop a general solution to the search-and-discovery problem that is scalable, user-friendly, and robust. This paper will seek to argue that the next generation of EMPRESS, offers just such a solution. Namely, EMPRESS provides an easy, light-weight, scalable way for users to tag areas of interest so that, at read time, they can load only the data of interest. A simplified version of the current paradigm is seen in Figure 1 while a simplified version of the paradigm EMPRESS tries to offer is seen in Figure 2. More generally, this paper will argue that rich, application-level metadata management can provide efficient data exploration and facilitate scientific discovery.

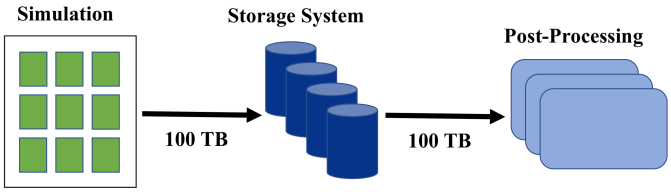


**Figure 1: Typical Write-Read Process (Simplified)**

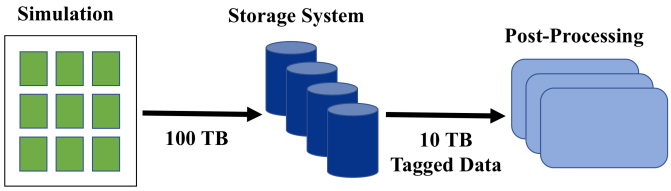


**Figure 2: EMPRESS Write-Read Process (Simplified)**

The rest of the thesis is organized as follows. Section 2 discusses several insights that influenced EMPRESS' design. Section 3 contains an overview of the design elements and decisions. Section 4 presents the Objector, a novel object naming system that EMPRESS uses to provide metadata-to-object mappings. Section 5 presents the implementation that is being evaluated. Section 6 contains the testing and evaluation information for EMPRESS while Section 7 presents the evaluation of the Objector. Next, Section 8 is a discussion of how EMPRESS and the Objector met our goals. Section 9 contains an overview of related work. Finally, Section 10 contains remarks about the next steps for the development of EMPRESS.

## 2 INSIGHTS

Several insights have driven our assertion that rich metadata can reduce time-to-insight, and have influenced EMPRESS's design.

***Scientists have a set of standard practices for annotating data.*** Scientists are trained to extensively document all experiments they perform in a lab notebook. They will often circle, highlight, add a sticky note, or use similar means to note unusual results. They will also write comments that they can refer back to at a later time. However, there are currently no tools for HPC that support integration of these kinds of annotations with simulation outputs. As a result, scientists are forced to rely on ad hoc methods of annotation such as using long, descriptive file names, or continuing to write all comments in lab notebooks [46]. These methods place undue burden on the scientist, are not scalable, and are not viable long-term solutions. Any annotation tool designed for scientific users should facilitate their natural annotation practices.

***Scientific tools should be easy to use.*** Many scientific tools place the burden of remembering on their users. Often, scientific users will be required to remember semantic information such as ids, which mean very little to them [48]. Some tools have improved on this, requiring scientists to remember only "meaningful" identifiers, but having to remember identifiers is still an undue burden. Scientists use many different simulations and tools, each of which may require a different set of unique, "meaningful" identifiers. Instead of requiring users to remember this information, tools should offer a way to catalog what is stored, providing a list of meaningful identifiers that can be used for data access or further exploration.

***Scientists need high level indexing.*** While there are many mature indexing techniques, most of these focus on indexing individual data points. While this can be useful, it is often too fine-grained, and results in indexes that are costly to build, read, and store [57]. In general, what scientists want to identify are interesting application runs, outputs within an application run, or spatial areas within an output. Storing and indexing this "interesting" metadata can therefore facilitate insight and offers space and performance gains over point-by-point indexing techniques.

***Scientists need efficient access to full-context metadata.*** Once scientists have captured high-level metadata about runs, outputs, and areas of interest, they need an efficient way to explore this information. If this metadata is embedded within the files or objects it references, providing a global view of the metadata requires crawling the storage system [45]. Using an external metadata management system can, by contrast, provide efficient access to global, full-context metadata. All of the metadata for one or more application can be explored jointly, allowing complex comparisons to be made and facilitating the search for interesting outputs or runs. Since the metadata is stored independently, it can be pulled to a faster tier and can be explored without having to touch the data, thereby greatly increasing efficiency [48].

***Scientists often have prior knowledge about what constitutes interesting data.*** Scientists perform simulations because they are searching for particular patterns or phenomena. Many of these patterns can be expressed using a combination of variable, geo-spatial, and temporal values [23]. Scientists can then use this information to perform lightweight in-situ analysis to identify "interesting" areas. Performing this analysis in-situ takes advantage of the fact that, at write-time, the data is distributed across a large number of nodes which are equipped with powerful processors. This distribution makes the analysis highly efficient.

***Scientific tools must support workflows.*** Scientists are increasingly using integrated application workflows (IAWs) such as Kepler [32] and Pegasus and DAGMan [9, 33] to pipeline their data through the various steps needed for discovery. Transaction management is crucial to the safety and efficiency of workflows: it ensures that only complete and correct datasets are made visible to workflow components and can notify various components when a dataset becomes available [26]. Therefore, any service that provides data access or is coupled with the data must provide transaction management.

## 3 DESIGN

No existing metadata management tool takes advantage of the insights presented above to offer a light-weight service for exascale that is domain-independent, extensible, scalable, easy to use and offers global metadata views and transaction management. To meet this need, we present the new EMPRESS, which improves upon its predecessor [24] in almost every way. EMPRESS offers the following contributions:

- **A conceptual metadata model** that supports domain independent, extensible, user-defined metadata.
- **A wide range of metadata searching services**. These functions support efficient data exploration, provide "cataloging" services (removing the burden of remembering from the user), and use logically meaningful search parameters (rather than semantic)
- **A client-server model** that offers scalability (for both write and read) through use of dedicated server nodes and a distributed, shared-nothing design.
- **Integration with the underlying storage system**. EMPRESS stores only logical, spatial coordinates from the simulation space. It then uses a special function called the Objector (see Section 4) to map these coordinates to the physical location of the associated data object. This design abstracts away low-level storage details from the user and makes the metadata itself storage system independent (providing portability). This design also enables EMPRESS to support a variety of storage-independent reading capabilities. It can be used to query and retrieve metadata, data, or metadata and the associated data.
- **Transaction management and fault tolerance**. EMPRESS provides a set of policies for transaction management to enable IAWs to work safely and efficiently. EMPRESS also uses a number of strategies to provide fault tolerance against various client and server failures.
- **A rich, simple API** that is exposed to the client as a C++ library.

### 3.1 Metadata Model

EMPRESS uses a domain-independent metadata model to provide a logical view of the metadata independent from the underlying storage details. This model is displayed in Figure 3. This model is composed of two broad categories: basic and custom metadata. Basic metadata is composed of three categories: (application) runs,

timesteps (individual writes or data outputs), and variables (such as temperature or pressure). Each basic metadata object can have an associated set of attributes, and each attribute is associated with a single tag or a named label.

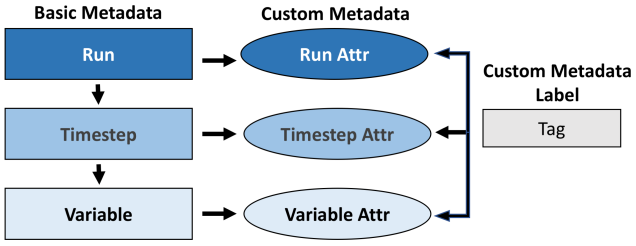


**Figure 3: EMPRESS's Metadata Model**

*3.1.1 Basic Metadata.* Basic metadata captures the structure of simulation outputs and simple metadata about the various components. As mentioned above, the three kinds of basic metadata EMPRESS uses are runs, timesteps, and variables. The basic run structure that EMPRESS assumes is a hierarchical one. A simulation may be run one or more times, and each run is composed of timesteps. Each timestep is composed of variables. The variables, and number of variables, output may differ for each timestep. An example of this structure is shown in Figure 4.

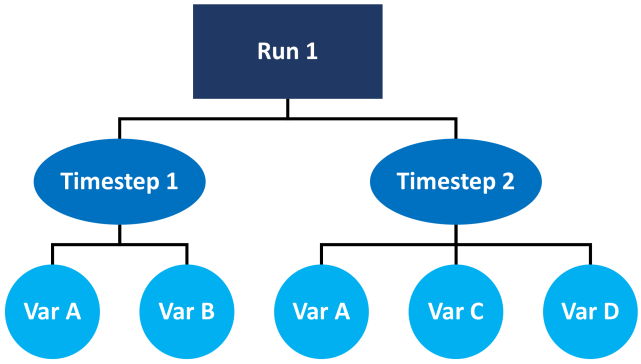


**Figure 4: Basic Metadata Model**

Runs are associated with basic information such as the name of the simulation, and the date and time it was run. Variables are associated with a name, version, and global, logical coordinates. For example, "temperature" version 1 could be associated with $X$:[50,100], $Y$[50,100], $Z$[300,350]. The use of versions allows EMPRESS to maintain meaningful variable names, while allowing for the possibility that physical variable may be captured using different meshes or granularities, or split into different partial-precision variables such as in APLOD level-of-detail encoding [18].

*3.1.2 Custom Metadata.* Custom metadata refers to extensible, user-defined metadata objects, known as attributes, and their associated labels, known as tags. An attribute can be viewed as an instance of a particular tag that is associated with a basic metadata object. Attributes can be associated with an entire run, timestep, or a subset of a variable. They are known as run attributes, timestep attributes, and variable attributes respectively. Each attribute is associated with a user-defined "tag," which provides a logical grouping of attributes. For example, a scientist could add a "combustion" tag to a section of the "temperature" variable for a particular timestep, and then later request all metadata objects with "combustion" tags. Thus, the custom metadata structure can also be viewed hierarchically since a single set of tags is used to refer to run, timestep, and variable attributes and since attributes within each of these categories can then be grouped by associated tag. An example of this structure is shown in Figure 5.

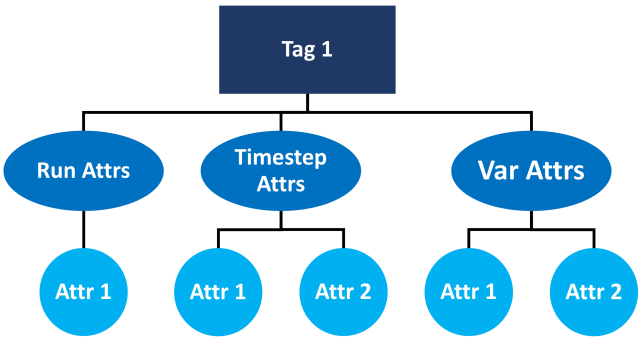


**Figure 5: Custom Metadata Model**

Each attribute is also associated with a value, which can be of any data type. Integers, floats, and strings are stored in their given format and all other types are serialized for storage. Variable attributes also store the global spatial coordinates of the variable subset they refer to. For example, a scientist could add a "maximum" attribute with value 10K onto the "temperature" variable from $X$:[75,80], $Y$[60,80], $Z$[300,325].

*3.1.3 Trade-offs.* Thus, EMPRESS allows scientists to annotate particular experiments or findings, as they are used to doing, and provides substantial flexibility through its extensible metadata and user-defined attributes. While EMPRESS does assume a hierarchical structure for data outputs (runs, timesteps, and variables), this system fits most simulations, and can also be used for data collection (data collection at a particular location, timesteps, variables). This slight reduction in flexibility allows great gains in efficiency, and allows EMPRESS to offer an extensive range of built-in querying functions. What EMPRESS does not currently address are unstructured grids and other non-stencil codes. Instead EMPRESS relies on the fact that most of these coordinate systems can be easily mapped to Cartesian coordinates. For example, XGC [20] outputs its particle list as toroidal coordinates. These coordinates are then converted into a regular Cartesian grid for visualization.

## 3.2 Metadata Searching

EMPRESS allows users to discover application runs, timesteps, variables, and subsets of variables that are of interest "based on the value of descriptive attributes, rather than requiring them to know about specific names or physical locations of data items" [46]. For example, users can perform lightweight analysis in-situ to identify

areas of high turbulence, tag these areas, and then either immediately visualize these areas on staging nodes or use them for future exploration. Thus, the write process, which can be seen in Figure 6, is as follows: lightweight in-situ analysis is performed on the output simulation data to generate custom metadata objects and tags. The custom and basic metadata are then sent to EMPRESS while the data itself is sent to the storage system. Then, as seen in Figure 7, these tags can be used at read time to find data of interest and to allow users to make more informed decisions about what data to read in and what storage tier to place the data in for reading (with higher priority data placed in faster tiers).

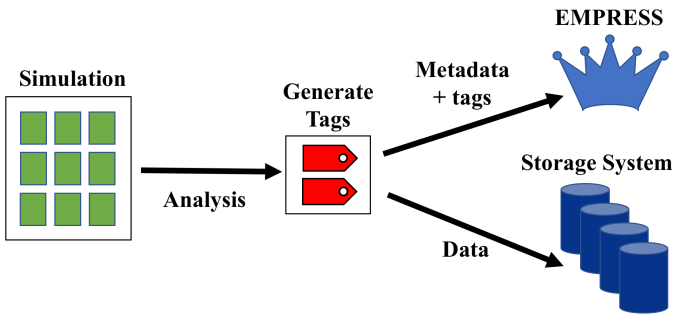


**Figure 6: EMPRESS Write Process (Simplified)**

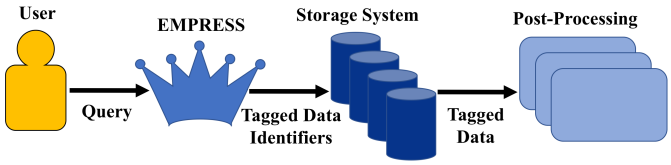


**Figure 7: EMPRESS Read Process (Simplified)**

EMPRESS offers an extensive range of queries to facilitate insight and support a wide range of use-cases. These queries offer a range of views from very specific and detailed to global and broad. In addition, each of the queries below that can be variable constrained can alternately be constrained based on any variable that matches a given substring. This option allows related variables to be queried simultaneously, such as variables with the same name but different versions, or similar names. For example, if a user has "temperature" variables version 1 and 2 or variables "velocity $X$", "velocity $Y$", and "velocity $Z$", attributes related to these variables can be examined simultaneously using the substring functions.

*3.2.1 Variable Attributes.* EMPRESS can return a list of variable attributes for a particular timestep that match any subset of the following constraints:

- Variable or any variable whose name matches a given substring
- (Global, logical) spatial coordinates
- Value (for numeric values)
  - supports less than, greater than, and range queries
- Tag

For example, a user can request attributes in a particular 3-D subspace where the maximum temperature is above 10K (where temperature is the variable, the value is above 10K, and the tag is "maximum").

*3.2.2 Run and Timestep Attributes.* EMPRESS can return a list of run attributes or timesteps attributes (for a particular run) that match any subset of the following constraints:

- Value (for numeric values)
  - supports less than, greater than, and range queries
- Tag

For example, a user could request all observation notes for a given run (where the tag is "observations" or something meaningful to the user).

*3.2.3 Timesteps.* EMPRESS can return a list of timesteps (for a particular run) that match either of the following criteria:

- Contains a variable
- Contains a variable attribute (along with any combination of possible variable attribute constraints listed above)

For example, a user can query "which timesteps contain the 'density' variable?" or "which timesteps contain a 'combustion' attribute for temperature?"

*3.2.4 Tags.* EMPRESS can return a list of tags that are associated with at least one variable attribute (for a particular timestep) matching a set of constraints. These constraints can be any subset of the following:

- (Global, logical) spatial coordinates
- Variable or variable whose name matches a given substring

For example, a user can ask "For timestep 100, what tags appear for Temperature in slice Z?" This can be used for surveying what tags appear (random sampling) or surveying a particular area of interest such as an edge of a plasma fusion reactor.

*3.2.5 Catalog Functions.* Implicit in the above statement but worth emphasizing is that EMPRESS offers means of cataloging all metadata. Users can retrieve a catalog of all runs, timesteps, variables, or tags (or even attributes for that matter). Users do not have to remember what the contents of a particular timestep, run, or even set of runs are, but instead can use a single query to determine this information and use the returned catalog entries to perform subsequent queries.

*3.2.6 Trade-offs.* These rich querying capabilities provide a number of benefits. First, this high-level indexing allows users to quickly identify data of interest with minimal latency and space requirements. Second, this system allows users to capitalize on the fact that many features of interest can be identified at write-time. Users can store metadata in EMPRESS and then query it at any time. Finally, this system provides both narrow and broad views of the metadata (and thus the data it describes). Queries can range from examining a particular spatial area of a variable, to full-context queries comparing timesteps within a run, to global queries comparing entire application runs. Some systems have opted to let users write their own SQL queries instead of providing a set API [47] [19] [49]. We have chosen not to do this as we believe it is an unnecessary burden to require users to be familiar both with the underlying

database storage structure and SQL and require users to do all of the work of creating a set of queries and testing them. We will consider adding a systematic way in the future for users to add additional queries or functionality, but still maintain that offering a foundational set of queries is crucial.

## 3.3 Client-Server Model

EMPRESS uses an in-memory RDBMS for metadata storage and a set of independent, distributed, share-nothing servers to ensure scalability. The servers are connected to the clients using a "server manager." These design choices merit both explanation and discussion as there are important trade-offs to each of these.

*3.3.1 RDBMS.* Each EMPRESS server maintains an in-memory RDBMS as the metadata storage backend.

*Trade-offs.* There are some performance trade-offs to using an RDBMS, which uses B-tree variants both for storage and indexes. In particular, all B-tree operations have a time complexity of $\Theta(\log n)$, whereas most hashing techniques offer $O(1)$ expected performance for write, read, and update. However, hash tables cannot efficiently offer the wide range of queries we support. EMPRESS can offer, for example, efficient queries on variable attributes using any combination of spatial, temporal, variable, tag and value constraints. Hash tables, by contrast, are designed to have a single "key" as a search value. Users are left with two options. One, they can attempt to roll all of the search parameters into a single string key, and then, for any search, crawl the entire hash table and do string matching on each entry, effectively turns the hashtable into a giant list, negating all of its performance benefits. The alternative is to use one parameter as the key (such as the tag or variable), and have all attributes of this category stored as a linked list. While this alternative provides efficient look-ups for attributes constrained by this single parameter, if there are additional constraints, the search must resort to crawling the entire linked list, and examining the values for each entry (again, generally using string matching). Additionally, if the search is not constrained by the parameter used as the key name, it must perform this process (of examining all values) for each key stored in the hash table. Thus, hash tables are not designed for multiple, varying constraints and under these conditions, an RDBMS is much more efficient.

*3.3.2 Independent.* EMPRESS is currently designed to use a set of dedicated server processes running in parallel, each of which maintains an in-memory RDBMs. Following a fairly standard client-server model, the following is the query process. First a client application issues an EMPRESS query using the EMPRESS API. Then the EMPRESS client sends the request to an EMPRESS server. The server responds to this message asynchronously, using message queueing. The EMPRESS server then interacts with the (local, in memory) database to perform the requested service, packages the results and returns them to the EMPRESS client. Finally, the results are returned to the client application. It is also important to note that EMPRESS is currently designed to be used underneath some sort of user interface, such as an I/O system like ADIOS [30] or a visualization system like ParaView [11] or VisIt [21]. This design allows users to take advantage of EMPRESS's capabilities while eliminating the need for them to learn a new interface and manage two separate systems. The evaluation section demonstrates that EMPRESS can be used in tandem with an I/O system, but this usage would not allow the I/O system to take advantage of several of EMPRESS's supported data read functionalities, which will be discussed below (see Section 3.4). This model is depicted in Figure 8.

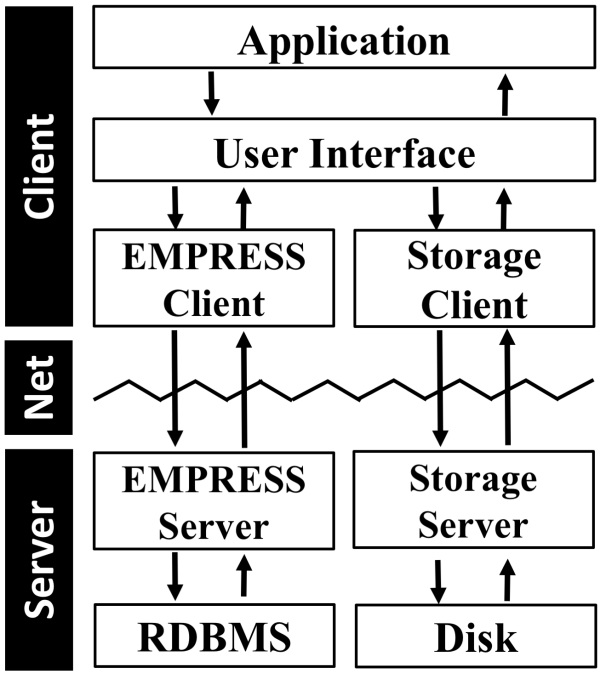


**Figure 8: Client-Server Model**

*Trade-offs.* There are many benefits to using a set of independent servers instead of having clients maintain local databases. There are three such alternative cases to consider: each client maintains a local database (one-to-one), a subset of clients maintains a database (N-to-M) and a single client maintains a database (N-to-1). In the first case, the disadvantages are that there are many small databases that must be either stored (using many small, costly writes), or first aggregated (which is expensive) and then stored. For reading, the data either needs to be aggregated (if it has not already been performed), or each read operation will require an expensive all-to-all communication. If instead only a subset of clients maintains a local database, the clients must use costly global communication to coordinate each metadata write. If only a single client maintains a local database, costly global communication is required and parallelization of metadata access is eliminated. In all three cases, since the metadata is maintained locally, it is not readily available for simultaneous reads by other workflow components. In addition, the metadata will take away from the compute node's available memory, which is generally untenable. Moreover, it is impossible to scale out the metadata operations (in terms of additional processing power and memory). However, using independent servers does require the allocation of nodes that could otherwise be used for computation. If too many nodes are allocated the result may be

idle resources and if an insufficient number are allocated the result may be a metadata bottleneck. EMPRESS tries to mitigate these concerns by offering efficient service with very few resources, such as a 1000:1 client-server processes ratio (see Section 6). In addition, EMPRESS allows servers to be dynamically allocated so that it can respond to server shortages or surpluses. In the future, we will consider offering the option to use EMPRESS locally (see Section 10 for more).

*3.3.3 Distributed, Shared-Nothing.* The metadata database is fully distributed across all EMPRESS servers. Each server maintains a copy of all basic metadata and a horizontal shard of all user-defined attributes, allowing distributed query processing. When EMPRESS is initialized, each client is assigned to a single server by a method that ensures the number of clients per server is balanced. Each server runs completely independently, and never contacts the other servers. The client side manages parallelism by directing client requests to different servers, or sending the same request to each server when necessary.

*Trade-offs.* Maintaining the metadata distributed across the servers allows parallel query processing, and allows the clients to write independently. However, since the servers use a shared-nothing design, for most read queries the clients must either coordinate or send the query to each server. We have found in practice that having one client per server send the query and then distributing the results introduces a much smaller overhead since it results in many fewer searches of the same databases. We have chosen to use a shared-nothing design rather than aggregating queries on the server side since this design offers a higher degree of flexibility. Clients can perform individual or collective reads, depending on resource availability and which clients ultimately need the information. Additionally, this design provides clients with the option of using a more systematic way of distributing metadata across the servers and querying for the results.

*3.3.4 Server Manager.* When the servers are initialized, they register their contact information with a single process called the "server manager." When the clients initialize, they then contact the server manager to obtain a list of available servers, and connect to one. All subsequent EMPRESS operations are carried out using this server. This server manager can help the system recover from server failure. If an operation takes longer than a certain period of time, the client can contact the directory manager to determine whether the server is still available and if not, connect to a new one. The server manager can either be a separate node (independent mode), or can itself be a server (shared mode).

*Trade-offs.* Using a server manager introduces single point of failure into the system. However, it provides a portable mechanism of connecting the client and server nodes, thereby facilitating EMPRESS's independent design. It also facilitates EMPRESS's shared-nothing design since no server coordination is required to connect the clients to servers, detect server failure, or reassign clients to servers in the case of failure or dynamic allocation of servers. Each server manager mode comes with its own trade-offs. Using independent nodes requires additional resource allocation, but also allows servers to be shared across multiple jobs. Each job uses the server manager's address to connect the clients to the servers, providing support for workflows by allowing them to access metadata updates as soon as they are available.

## 3.4 Storage System Relationship

EMPRESS has a somewhat complex relationship to the data storage system in that it manages the metadata externally, provides integration with the storage system, and also offers users abstraction from the storage system. It is perhaps obvious, but since EMPRESS is a separate system for metadata management the metadata is not necessarily embedded in the data objects it describes. Having this metadata external to the data offers several advantages, which were outlined in the insight sections, such as offering efficient access to global and full-context metadata. EMPRESS's metadata is also independent of the storage backend used for the data since it stores only logical, spatial coordinates from the simulation space and not physical data locations (see Section 4.5 for more). However, providing a storage-independent mapping between logical and physical locations a challenge. To do so, EMPRESS uses the Objector. The Objector will be discussed in more detail later (see Section 4). This ability allows great flexibility in storage backend and allows users to perform data reads using an EMPRESS function that requires only the associated variable metadata and the logical subsets desired. This design abstracts away low-level storage details from the user and removes from them the burden of remembering file or object names to read in files. This design also means that a system can use EMPRESS to provide storage-independent read functionalities for just data, just metadata, or metadata and its associated data. Inspired by ADIOS [30], EMPRESS offers an additional functionality: it uses the simulation's domain decomposition function to allow clients to read the data for a particular writing process if they so desire.

*Trade-offs.* As discussed above, there are many advantages to maintaining metadata external to the data it references. However, the client must make calls to both the metadata and data management systems to ensure consistency. In addition, since the metadata is completely opaque to the storage system, the storage system cannot leverage this information to make more informed decisions. Programmable storage systems [43] [42] [34] are being investigated at UCSC that would allow this metadata to be shared across the layers of the stack. This technology could largely resolve the "middleware vs. storage system layer" metadata debate.

## 3.5 Transaction Management and Fault Tolerance

*3.5.1 Transaction Management.* As discussed above, transaction management is crucial to supporting IAWs, which are important tools for scientific discovery. EMPRESS ensures consistency and availability by using transaction management that is largely based on the $D^2T$ system [27]. In essence, each client independently writes the metadata for its assigned subsection of the simulation space, and if no client encounters an error, all of these writes are committed as a single transaction. This management provides a slightly relaxed version of ACID (atomicity, consistency, isolation, durability), a set of properties that are intended to ensure the validity of a database even in the case of different kinds of failure. Transactions are atomic, meaning they are either committed in their entirety or aborted, and

consistent, meaning that they guarantee the database is left in a valid state. However, the isolation requirement is somewhat relaxed: uncommitted transactions are not visible to any other process but are stored in the database in an "inactive" state. In addition, the durability requirement is somewhat relaxed because transactions are check-pointed to disk regularly rather than immediately.

*Trade-offs.* Since transactions are not immediately stored to disk, in the case of server failure, there either needs to be a mechanism for recovering the updates since the last transaction, or just acknowledgement of slightly reduced fault tolerance. The best method of addressing this issue will be investigated in the future. However, it is important to note that this design provides large performance gains by reducing the frequency of costly saves to disk. Managing the visibility of transactions within the database itself imposes a slight overhead, but also increases concurrency since none of the metadata tables are locked when a transaction is in process. Since uncommitted transactions are invisible to read processes and the metadata tables are not locked during writing, the metadata database can be safely accessed for both writing and reading at the same time. As mentioned above, this facilitates workflows since analysis and visualization components are given safe and immediate access to metadata updates. This also means that if, during post-processing and analysis, an additional feature of interest is discovered, it can be added to the metadata database without affecting the stored data. This independence is important to application scientists since any modifications made to the data introduce the possibility of data corruption.

*3.5.2 Fault Tolerance.* EMPRESS provides return values for all operations to indicate whether it succeeded. Through use of the server manager, and regular check-pointing of the databases to disk, it can also recover from server failure. However, for the most part, EMPRESS leaves particular failure protocols up to the user. If a particular operation fails, the user can decide whether they want to abort the operation, try again, or respond in some other way. This flexibility recognizes that many users and scenarios will require a different set of responses.

*Trade-offs.* EMPRESS provides a balance between fault-tolerance and efficiency. For example, after clients send a message to the servers, they wait for a response. Doing so produces some idleness, but allows clients to immediately retry the operation if it fails, or respond is some other situation dependent way. EMPRESS has a few additional fault-tolerance policies to address in the future (see Section 10 for more).

### 3.6 API

All of the functionality discussed above is exposed to the client as a rich, simple C++ API library. It offers six categories of functions:

- **Write:** write functions for each of the seven metadata types, which can insert metadata objects one at a time, or in a batch:
  - Basic metadata: application run, timestep, variable
  - Custom metadata: run attribute, timestep attribute, variable attributes, tag
- **Delete:** supported for runs, timesteps, variables and tags. Deleting any of these entities also deletes any related metadata. For example, deleting a run deletes all associated timesteps, variables, tags, and attributes. Deleting a tag deletes all associated attributes.
- **Transaction:** make visible/invisible, supported for all seven metadata types (meaning that users can safely write additional metadata of any type at any time).
- **Search:** the wide range of metadata query operations discussed above.
- **Data Access:** as discussed above, EMPRESS offers functions that allow a client to map from logical spatial dimensions to matching data object names, and from a writing process's MPI rank to matching spatial dimensions.
- **Initialize/finalize:** the EMPRESS client is initialized with a simple initialize function. There is also a simple finalize function to tell the EMPRESS clients to shut down, and a function that tells the EMPRESS servers to shut down. As discussed above, EMPRESS allows the same set of servers to be shared across jobs, meaning the clients and servers will not always be shut down at the same time.

## 4 THE OBJECTOR

An important piece of related work that we have developed is the *Objector*. As will be discussed below, the Objector is both a novel object name generator and is used by EMPRESS to provide mappings from logical spatial coordinates from the simulation space to physical data locations. A full discussion of the Objector is outside the scope of this paper, but more details can be found in a paper by Nasirigerdeh [34].

### 4.1 Motivation

Object-based storage systems are widely used in the cloud and in HPC. Several of the most widely-used parallel file systems use object-based storage, such as Lustre [41] and PanFS (Panasas) [55]. The production exascale systems that EMPRESS is designed to support are expected to have around 10 trillion objects. At this scale, it is infeasible for EMPRESS to store and search a list of all object names in order to map from the logical space of metadata to the physical space of the associated data. Therefore, as discussed above (see Section 3.1), EMPRESS does not store any metadata about the physical location of data objects and instead stores only logical spatial coordinates. For I/O systems that support hyperslab reads (subspace selections using offsets and counts for each dimension), logical spatial coordinates are sufficient. Using a variable's logical coordinates, any set of coordinates can easily be converted into global offsets (e.g., if a variable's logical coordinates start at (10,10,10), the coordinate (15,15,15) has global offsets of (5,5,5) ). However, for EMPRESS to provide direct access to data objects in storage systems, it needs a mechanism to map from the logical space to the physical space. This mapping is provided by the Objector. The Objector is an object name generator that provides mapping from basic metadata (such as run name, timestep, and variable name and version) and logical spatial coordinates to matching object names and byte offsets within these objects. The Objector can be used for any storage system that supports named data objects (such as file systems), but is especially beneficial for object storage systems because of their large, flat namespaces.

## 4.2 Introduction

The Objector is an object name generator that provides mappings from user-meaningful metadata and logical spatial coordinates to object names and data offsets. As mentioned above, the Objector is a piece of code that can be stored with $O(1)$ storage space and eliminates the need to store any object names. The Objector is inspired by the following insight: for every simulation, scientists write a small piece of code that assigns a portion of the logical space to each process. Therefore, each simulation already has, in essence, a function to map from the physical space to the logical space. If this mapping can be captured, it can be used in an object name generator.

## 4.3 Design

The Objector has a large number of parameters. First, the Objector needs descriptive identifiers for the run (jobID, name), timestep, and variable (name and version). Second, it needs the object size and data size (how many bytes per data point). Finally, it needs information about the data decomposition so that it can determine where chunk boundaries are, which is in turn can be used to determine where object boundaries are.

Since simulations use the same domain decomposition throughout an application run, the Objector only needs to be stored one per run by EMPRESS. EMPRESS databases themselves can later be serialized and stored into objects, using the descriptive run identifiers as the key. Then, when a user wants to read from a particular run, the associated metadata (EMPRESS databases) can be loaded into memory.

## 4.4 Algorithm

With this initial prototype, a regular Cartesian grid is assumed, meaning that each process is assigned a uniformly sized "chunk" of the simulation space, illustrated in Figure 9. The Objector then uses a 1-D chunk domain decomposition, meaning each chunk is sliced along the $X$-axis and entire $YZ$ planes are used to compose the objects.

A detailed explanation of the Objector's algorithm is included in Appendix A along with an example. A simplified version of the algorithm is as follows:

(1) Convert the coordinates to global offsets
(2) Find the first chunk that overlaps with the coordinates (by dividing the starting coordinate by the uniform chunk size)
(3) Find the first object within this chunk that overlaps with the coordinates (using the uniform object size)
(4) Iterate through all overlapping object coordinates by incrementing $Y$ and $Z$ by the fixed chunk $Y$ and width, and incrementing $X$ by the uniform object $X$-width.
(5) For each object, generate a name using the unique run, timestep and var identifiers, along with the starting coordinates

## 4.5 Contributions

- *Storage system independence.* The Objector allows EMPRESS to store storage-system independent metadata and map any attribute to the associated data. Therefore, if the naming scheme or backend storage system changes, only the Objector (no stored metadata) needs to be updated.
- *Increased querying efficiency.* The time complexity for the Objector is determined by the number of object names that match the given coordinates rather than the total number of object names, making it much more efficient to retrieve matching object names.
- *O(1) storage.* The Objector is a piece of code that can be stored as a string using $O(1)$ storage space. This allows an unlimited number of object names to effectively be stored with virtually no storage space requirements.
- *Faster data reads.* The Objector contributes to faster read times in several ways. First, mapping data directly to objects allows logical alignment between the data and object boundaries, increasing locality. This approach is in contrast to typical data striping, which uses fixed byte offsets that generally do not align to the dataset's logical structure. Using smaller data blocks also allows more precise reads, resulting in less "uninteresting" data being read and thereby increasing performance. Finally, some object storage systems, such as Ceph [54], offer a way to slice the data on the storage side [53], eliminating the need to send "uninteresting" data over the network and thereby further increasing performance.

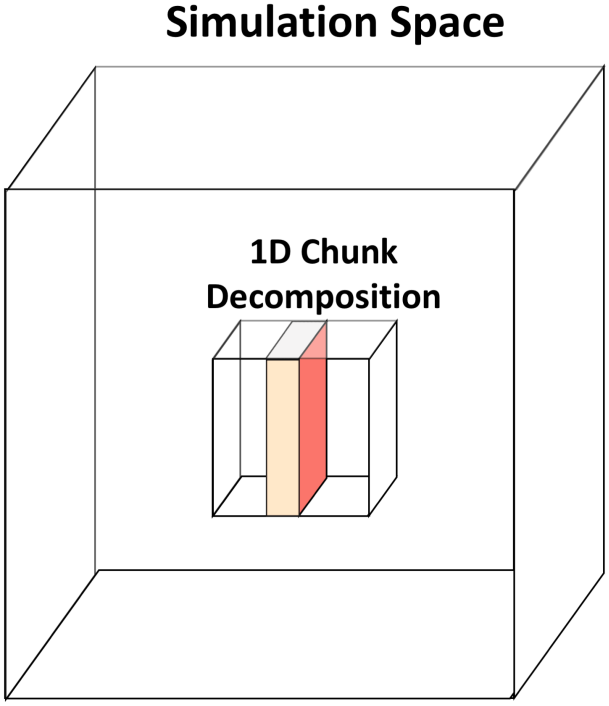


**Figure 9: The Objector's Data Decomposition**

## 4.6 Related Work

There has been a lot of work investigating ways of managing metadata growth rates. PnetCDF [25] files and the default mode of HDF5 [12] both use logically contiguous data organization, which

minimizes the amount of metadata that needs to be stored for reading particular byte offsets. However, while this approach is successful in keeping the metadata load small, the penalties for writing at scale [31] and reading at scale [28] are largely not worthwhile except in the most space constrained environments or situations. ADIOS [30] and the new chunking mode supported by HDF5 and netCDF-4 allow data to be written in non-contiguous chunks by maintaining basic metadata for each chunk. This approach allows much more efficient writing and good read performance, but it is well documented that the overhead of writing this metadata per-chunk and having to linearly search this metadata to find a matching chunk presents scalability limitations that will preclude use of this approach at exascale [29]. DeltaFS [59] offers a similarly modular approach to namespace management by externally storing a separate namespace for each set of file outputs. It has also recently been extended to offer indexing on these files [58]. However, DeltaFS has not yet examined how to provide more fine-grained indexing, or how to use this namespace system to support rich metadata writing and searching.

### 4.7 Future work

This initial version of the Objector is designed for regular, rectangular Cartesian grids. However, we are in the process of expanding this technology to other topologies. In particular, we are investigating using the Objector for fusion simulations, high energy physics (event-based data collection), and comparative genomics [34].

## 5 IMPLEMENTATION

The current EMPRESS implementation relies on existing embeddable RDBMS libraries and other support tools. Each of these is explored below.

### 5.1 RDBMS

The implementation tested uses SQLite [1] [36] version 3.22 as the RDBMS. While other relational database systems could have been used, SQLite offers many desirable properties: it uses a serverless model and a dynamic type system and is very light-weight. Since SQLite is serverless EMPRESS can institute its own client-server model using a different networking service that uses asynchronous communication (discussed below) and allows a single client function call to result in many SQL command executions (more or less message bundling). Since SQLite uses a dynamic type system, EMPRESS can use a single attribute table to support attributes of various types (integer, real, text, blob), which results in more efficient querying, and a smaller storage overhead. Some of the drawbacks of SQLite, such as the fact that it locks the entire database during transactions, are avoided through EMPRESS's shared-nothing model and custom service interface.

### 5.2 Database Relational Schema

The current implementation uses a separate table per class of metadata. There are seven in all: one each for runs, timesteps, variables, run attributes, timestep attributes, variable attributes, and tags. This implementation uses a highly normalized form, which eliminates redundancy and inconsistent dependency. The implementation also uses a number of indices to speed queries. Almost all of these strictly index EMPRESS's internal identifiers for key search parameters (i.e., a combination of run, timestep, variable or tag ids, which are unsigned 64-bit integers).

*5.2.1 Trade-offs.* Normalization reduces metadata redundancy and increases data integrity, but requires more costly join operations.

### 5.3 Networking Library

For portability, EMPRESS is built using the Faodel [50] infrastructure. Faodel offers data management services initially conceived to support Asynchronous Many-Tasks (AMT) applications, but has since been generalized to support broader independent service and work flow integration tasks. For this project, the data management services are being augmented with rich metadata functionality. The base Faodel metadata is based on key-value stores driving everything through keys. Faodel is built upon the long stable and performant NNTI RDMA communication layer from the Nessie [35] RPC library from Sandia. All of the layers above the communication management have been replaced, offering better C++ integration and richer functionality. EMPRESS uses the Boost serialization library to serialize the data passed as messages between the client and servers and to store non-native types in SQLite.

### 5.4 The Objector

The Objector is currently implemented using Lua [17] to gain several advantages over alternatives. First, Lua functions can be turned into a string (easily permitting storage in EMPRESS) and then loaded and executed like normal code. Lua is also designed to be used as embedded code, offering easy access from C++ code with near native performance. Lua is also user-friendly in that it is very similar to Python, a language many users are familiar with. This is important because, as the Objector expands to more complex cases, it will be critical for users to write their data decomposition functions in Lua so that it can be used by the Objector. Finally, Lua also has a low footprint, adding minimal overhead.

## 6 EMPRESS EVALUATION

### 6.1 Goals

The overall goal for EMPRESS is to provide a useful tool for extreme-scale scientific simulations. To do so, EMPRESS must be scalable, introduce minimal overhead, be useful for accelerating data exploration, and provide advantages over alternatives. Therefore, we had the following goals for the evaluation of EMPRESS:

- *Scalability:* To determine EMPRESS's performance with different client-server ratios (strong scaling) and with a fixed client-server ratio but varying numbers of clients (weak scaling). In particular, we wanted to evaluate whether EMPRESS can efficiently support metadata operations with limited hardware resources.
- *Overhead:* To determine how much of an overhead EMPRESS presents in terms of memory usage, and metadata write and read times.
- *Accelerating Data Exploration:* To determine if EMPRESS can be used to accelerate data exploration by returned tagged

areas of interest that limit the reading scope and thus limit the read time.

- *Comparison to Alternatives:* To compare the performance of EMPRESS's metadata writing and reading operations to commonly used metadata management services.

## 6.2 Testing Configurations

*6.2.1 Testing Environment.* Testing was performed on the Skybridge capacity cluster at Sandia, a supercomputer designed to run multiple mid-sized jobs at once. Skybridge has 1848 nodes with 16 cores/node (2 sockets each with 8 cores running 2.6 Ghz Intel Sandy Bridge processors). It has an Infiniband interconnect and 4 GB RAM per process. The software environment is RHEL7 as the OS, the GNU C++ compiler version 4.9.2, OpenMPI 1.10, and HDF5 1.10 as the I/O system (choice discussed below). All experiments utilized the Lustre parallel file system, and the default stripe size and count parameters were set by HDF5. Additional tests were also run on the Chama and Serrano capacity clusters at Sandia. Those show similar results and are omitted for space considerations.

*6.2.2 Types of Experiments.* To evaluate EMPRESS, we ran two series of tests. One set uses HDF5 [12] for data management and EMPRESS for metadata management and the second set uses HDF5 for both data and metadata management. The choice of HDF5 as the comparison system merits some discussion. One reason HDF5 was chosen is that, by most measurements, it is the most commonly used I/O library for HPC science applications [6]. It thus presents a realistic representation of the metadata management used by scientists today. In addition, while none of these I/O libraries offers metadata tagging or searching (see Section 9.3 for more), HDF5 offers superior metadata management to the other alternatives. In particular, HDF5 offers scoped attribute namespaces, meaning that it can naturally support run, timestep and variables attributes and can limit searches to the relevant set of attributes. HDF5 also supports packet tables, which provides a means of creating an attribute table with variable length attributes providing a compact and global view of the attributes, and an easy way to iterate over the relevant set of attributes [12]. It is important to note, however, that HDF5 does not offer any metadata indexing. HDF5 deploys B-trees for searching data structures (such as variables, or tables within a file), but offers no other indexing. In addition, the only attribute "querying" functionality that HDF5 offers is a way to iterate through all of the attributes associated with a particular data object. Therefore, to provide a comparison of EMPRESS and HDF5, we first had to extend HDF5 using available data structures (the packet table) so that it could offer the same set of functionality that EMPRESS offers. In this way, almost all EMPRESS functionality was duplicated with this HDF5 packet-table-equivalent with a few exceptions. First, since HDF5 maintains metadata internally (it is embedded in the same structure as the data it describes), we have omitted transaction management. In addition, we rely on the natural HDF5 data structures for offering the (simplified) basic metadata such as timestep values, and variable names and versions.

*6.2.3 Experiment Scales.* For both the HDF5 and joint EMPRESS-HDF5 tests, experiments were performed using 1000, 2000, and 4000 processes for writing. These scales were chosen to facilitate in-depth testing, but future evaluation will use larger scales. For the joint EMPRESS-HDF5 tests, for each testing scale, experiments were run using ratios of 100:1 and 1000:1 write processes to EMPRESS server process. These configurations are summarized in Table 1. Each of these configurations was performed a minimum of five times, and results were averaged across these runs.

**Table 1: Testing Configurations**

| Test Type | Number of Write Processes | Number of Metadata Servers |
|---|---|---|
| **EMPRESS + HDF5** | **1000** | **1** |
| **EMPRESS + HDF5** | **1000** | **10** |
| **EMPRESS + HDF5** | **2000** | **2** |
| **EMPRESS + HDF5** | **2000** | **20** |
| **EMPRESS + HDF5** | **4000** | **4** |
| **EMPRESS + HDF5** | **4000** | **40** |
| **HDF5** | **1000** | **N/A** |
| **HDF5** | **2000** | **N/A** |
| **HDF5** | **4000** | **N/A** |

## 6.3 Writing

The following outlines the testing setup used in evaluation. The setup attempts to provide a small example of typical simulation runs. Each test writes a single application run with 3 timesteps. Each timestep is composed of a set of 10 3D variables. These represent the categories of data measured in the simulation. Variables used in this evaluation include temperature, pressure, and density among others. Each of these variables is distributed across the processes using a 3D domain decomposition, so that each process writes a regular hyper-rectangle (a "chunk") that is 125x160x250 data points. Each data point is an 8 byte double. It is important to note that this data size per chunk is held constant across all runs meaning that as the total number of clients increases, so too does the total amount of data written. This chunk size amounts to 0.4GB per process per timestep (0.04GB per variable, with 10 variables). This data size was chosen because it constitutes 10% of a process's total RAM, which is a reasonable estimate for how much data a process may write per output. The processes also write a set of metadata through a set of EMPRESS functions, which will be discussed in more detail below (see Section 6.5). There are 10 tags, each of which has a set frequency that determines what percentage of chunks it is associated with. These metadata tags and frequencies were chosen to estimate normal use. More tags and higher frequencies can easily be supported. These tags include "blobs" (a scientific name for spatial phenomena), annotations and ranges at varying frequencies, and maximum and minimum. The blobs have a Boolean value (indicating presence or absence of a particular feature), the maximum and minimum have a double value (like the associated data), the notes have text values, and the ranges have values that are a pair of integers. These tags are summarized in the Table 2. In addition, for each timestep, the processes determine their local maximum and minimum for the "temperature" variable and then coordinate to determine the global maximum and minimum. One process then writes these values as timestep attributes. At the end of the run,

**Table 2: Tags used in Testing**

| Tag | Data Type | Frequency (% of all chunks ) |
|---|---|---|
| **Frequent Blob** | **Boolean** | **25%** |
| **Infrequent Blob** | **Boolean** | **5%** |
| **Rare Blob** | **Boolean** | **.1%** |
| **Maximum** | **Double** | **100%** |
| **Minimum** | **Double** | **100%** |
| **Frequent Note** | **String** | **20%** |
| **Infrequent Note** | **String** | **2.5%** |
| **Rare Note** | **String** | **.5%** |
| **Infrequent Range** | **2 Integers** | **10%** |
| **Rare Range** | **2 Integers** | **1%** |

this same process calculates the maximum and minimum for temperature across the entire run, and writes this as run attributes into EMPRESS.

## 6.4 Reading

Each testing configuration uses 10% of the number of write processes for reading meaning that 100, 200 and 400 read processes are used. For the EMPRESS-HDF5 tests, the same number of servers are used for both reading and writing, meaning that the reading process-server ratios are 10:1 and 100:1. Reading consists of 3 stages. The first stage performs six read patterns that are identified by the Six Degrees of Scientific Data[28] as typical for analysis codes. These six patterns are, for a given timestep:

(1) Read all data
(2) Read all data for a variable
(3) Read all data for 3 variables
(4) Read a plane in each dimension
(5) Read a 3D subspace
(6) Read a partial plane in each dimension

The second stage examines how rich metadata can be used to accelerate these read patterns. Patterns 2 and 3 (reading a single variable and reading three variables) are performed using three different read selectivities. This process involves querying a particular feature of interest (tag) that appears with the given selectivity (found on 25%, 5% and .1% of the data chunk), and then reading in only the data that matches this query. Finally, the third stage performs a wide variety of metadata queries and subsequent data reads to test the performance of the full suite of functionality that EMPRESS offers and to test the performance of the HDF5 equivalent.

## 6.5 Write and Read Examples

The basic write and read query process has been outlined above (see Section 3.3.2), but here we will demonstrate how these can be combined to write an entire application run, and perform various read queries. These are the write and read processes used in the testing harness.

*6.5.1 Writing Example.* Algorithm 1 demonstrates the basic write process for an application run. At the start of writing, each compute process initializes the EMPRESS client, which then connects to a single EMPRESS server with which it can pass requests as messages. It is important to note that only the functions that start with an * are performed by each client. The remaining functions are called by one client per server, ensuring that each server is given a copy of each basic metadata object (runs, timesteps, variables) and tags, and that the attributes are written to a single server (this is one of the * functions). Also, each write function supports both an individual and batch version, for writing one or multiple pieces of metadata. The algorithm outlined below is the single-client "batch" insertion algorithm. All testing runs use this "batch" insertion since initial tests revealed that at the evaluated scales, the individual metadata writes take five to ten times longer than the batched writes. This performance difference is understandable since the many, small write requests overload the servers with metadata that can easily be bundled, sent to the server and inserted in a single batched transaction. More extensive batching is possible, using client aggregators, and this possibility will be explored in the future.

**Algorithm 1** Batch Writing algorithm

```
1:  procedure WriteRun
2:      empress_init (...)
3:      metadata_create_run (...)
4:      metadata_create_tags_batch (...)
5:      metadata_activate_run (...)
6:      metadata_activate_tags (...)
7:      for all timesteps do
8:          metadata_create_timestep (...)
9:          metadata_create_vars_batch (...)
10:         for all variables do
11:             *write_chunk_data (...)
12:         end for
13:         metadata_insert_timestep_attributes_batch (...)
14:         *metadata_insert_var_attributes_batch (...)
15:         MPI_Barrier (...)
16:         metadata_activate_timestep (...)
17:         metadata_activate_var (...)
18:         metadata_activate_timestep_attributes (...)
19:         metadata_activate_var_attributes (...)
20:     end for
21:     metadata_insert_run_attributes_batch (...)
22:     metadata_activate_run_attributes (...)
23: end procedure
```

One interesting thing to note is that runs are activated as soon as they are inserted into the database, ensuring that as soon as data is available for a timestep in the run (as soon as a timestep within the run is activated), it will be visible and therefore usable by downstream processing.

*6.5.2 Reading Example.* An example metadata exploration and read procedure is presented in Algorithm 2. This algorithm assumes that the user does not remember what runs, timesteps, or variables they have stored (if any of these assumptions are false, the associated catalog functions could be omitted). Again, the first step for

reading is to initialize the EMPRESS client, which then connects to a single EMPRESS server. Since each server maintains a copy of all basic metadata and tags, a client needs to query only a single server. To provide each client with the catalogs, the catalog functions can be called once per client or can be called by a subset of clients that broadcast the results to the remaining clients. In practice, we have found that having a subset query and broadcast is more scalable. In the testing harness, we just have one client catalog and then broadcast to all clients. However, as mentioned previously, all attribute functions require either having each client query each server, or having one client per server issue the query and share the results. We have found that the latter is much more efficient. The basic read example, outlined below involves cataloging the set of runs to determine which to explore, and then, for this run, cataloging the stored timesteps. The user then catalogs the variables that were output for a timestep of interest, and the tags that are associated with the run. This is then used to catalog all attributes associated with this variable, a tag of interest, and a logical spatial location (such as "temperature", "maximum", $X$:[50,100], $Y$:[100,200], $Z$:[100,200]).

**Algorithm 2** Metadata Reading algorithm

```
1: procedure ReadVarAttrs
2:     empress_init (...)
3:     metadata_catalog_runs (...)
4:     metadata_catalog_timesteps (...)                ▷ For Run X
5:     metadata_catalog_vars (...)        ▷ For Timestep Y of Run X
6:     metadata_catalog_tags (...)                     ▷ For Run X
7:     metadata_catalog_all_var_attrs_with_tag_var_dims (...)
8:     MPI_Allgather (...)              ▷ Each client receives all attrs
9: end procedure
```

*6.5.3 HDF5 Attribute and Data Reading Example.* Algorithm 3 presents an example of how the HDF5 testing harness uses HDF5 to read attributes and the data associated with these attributes. First, a single process opens the timestep file and the variable attribute packet table. It then traverses the packet table to look for attributes that match the given tag and variable name. It then closes the file and table. These attributes are then serialized and broadcast to the remaining processes. Each process then iterates over the list of attributes and, if any overlap with its assigned portion of the simulation space, it reads in the data associated with this overlapping portion. Reading only the overlapping portion ensures that no portion of the attribute's associated data will be read multiple times. The attribute is then read as follows. First, the overlapping, logical, spatial coordinates are converted into $X$, $Y$, and $Z$ offsets and counts (where offsets begin at 0). The remaining procedure is the standard HDF5 read process: the dataset (variable) of interest is opened; the dataspace for the dataset (variable) and hyperslab of interest (overlapping area) are then retrieved; these dataspaces are used along with the offsets and counts of interest to read the data. The HDF5 resources are then closed.

*6.5.4 HDF5 and EMPRESS Attribute and Data Reading Example.* When EMPRESS is used in conjunction with HDF5, with EMPRESS managing all metadata attributes, the process is nearly identical. This process is outlined in Algorithm 4. The only procedure that changes is the first. The query is performed using one client per server, allowing parallel query processing. The internal search mechanism is naturally quite different since it uses EMPRESS's RDBMS backend instead of an HDF5 packet table. The attributes are then allgathered (instead of broadcast) since multiple clients have retrieved attributes.

**Algorithm 3** HDF5 Attribute and Data Reading algorithm

```
1: procedure HDF5Read
2:     if rank == 0 then
3:         open_timestep_file_and_var_attr_table (...)
4:         md_catalog_var_attr_w_tag_var(tagname, varname)
5:         serialize_attrs (...)
6:         close_timestep_file_and_var_attr_table (...)
7:     end if
8:     MPI_Bcast ( attrs )
9:     read_data_for_attrs ( attrs )
10: end procedure
11:
12: procedure read_data_for_attrs
13:     open_timestep_file_collectively (...)
14:     for all attrs do
15:         if dims_overlap(proc_dims, attr_dims)  then
16:             get_overlapping_dims (proc_dims, attr_dims)
17:             read_attr(attr, overlapping_dims)
18:         end if
19:     end for
20:     close_timestep_file (...)
21: end procedure
22:
23: procedure read_attr_abbreviated
24:     convert_spatial_coords_to_offset_and_count (attr)
25:     HDF5_open_variable_and_dataspace (...)
26:     HDF5_select_hyperslab (...)
27:     HDF5_open_attr_dataspace(...)
28:     data = HDF5_read ( var_id, attr_dataspace, var_datasapce)
29:     close_HDF5_resources (...)
30: end procedure
```

**Algorithm 4** HDF5 Read with EMPRESS

```
1: procedure HDF5ReadWithEMPRESS
2:     if rank < num_empress_servers then
3:         md_catalog_var_attr_w_tag_var(tagname, varname)
4:         serialize_attrs (...)
5:     end if
6:     MPI_Allgather ( attrs )
7:     read_data_for_attrs ( attrs )
8: end procedure
9:
```

## 6.6 Results

*6.6.1 Scalability.* Figure 10 demonstrates how the EMPRESS write performance varies with a varying client-server ratio and

a fixed number of clients, and how it varies with a fixed client-server ratio with a varying number of clients. As expected, the number of servers dramatically affects metadata write performance, since clients wait to receive a response from the servers indicating whether the writes succeeded. However, even though there is a factor of 10 difference in the number of servers, the performance varies by approximately factors of three for 1000 write processes and six for 2000 write processes. This result is partially by the requirement that the basic metadata (about the application run, timesteps, variables, and tags) be written to each server. Also, with the additional servers, there are fewer clients per server and thus a greater chance of load imbalance. While clients are equally distributed across the servers, it is randomly determined which clients will write an attribute for a given variable and tag (subject to the tag's frequency). Looking at the fixed client-server ratios, we can see that EMPRESS scales well. For the 100:1 client-server ratio case, the total metadata write time actually decreases from 1000 write clients to 2000 write clients. For the 1000 client case, most operations performed better on average. However, a small fraction of operations took significantly longer. This finding can largely be understood as a result of bottlenecks due to load imbalancing and random fluctuations in performance. For the 1000:1 client-server case, the 2000 write client performance is slightly worse than the 1000 write client performance. This finding is likely a result of slight load imbalancing.

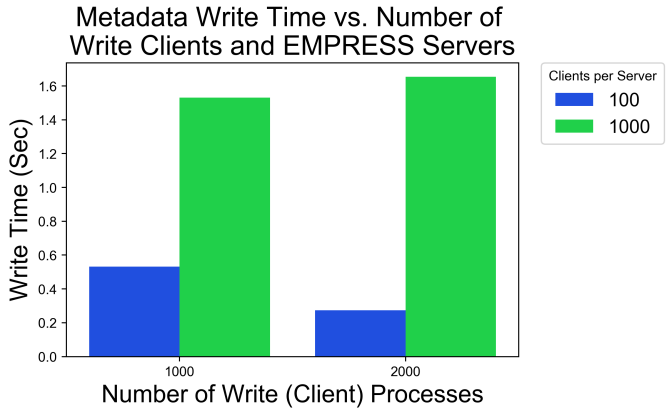


**Figure 10**

### 6.6.2 *Overhead.*

*Storage Overhead.* Table 3 demonstrates the metadata storage requirement at the evaluated scales. The results show that EMPRESS can support basic and rich metadata and can provide mapping from logical spatial dimensions to physical storage locations with a trivial amount of required storage space.

*Write and Read Overheads.* Figure 11 demonstrates the performance of the EMPRESS's metadata write functions with averages represented using the green lines, 25 and 75 percentiles demonstrated using the boxes and the maxima and minima demonstrated by the whiskers (the header and footer lines). All metadata operations used in writing are demonstrated above. These include writing each of the seven types of metadata (runs, timesteps, variables, tags, run attributes, timestep attributes and variable attributes) and

**Table 3: Data and Metadata Sizes at Different Scales**

| # Write Processes | # EMPRESS Servers | Data Size | Metadata Size | Metadata % of Data Size |
|---|---|---|---|---|
| 1000 | 1 | 1.2 TB | 8.1 MB | .0007% |
| 2000 | 2 | 2.4 TB | 16.2 MB | .0007% |
| 4000 | 4 | 4.8 TB | 32.6 MB | .0007% |

"activating" the metadata for use in other workflow components (transaction management). The graph demonstrate that EMPRESS can support basic metadata write operations with relatively good efficiency. The slowest operations are writing a batch of 10 variables, which takes around .16 seconds, and writing a batch of 10 attributes, which takes around .5 seconds. The variable attributes thus constitute both the largest performance cost and the largest fraction of metadata written, using around 90% of the total storage size, because variable attributes are the only kind of metadata that is inserted by each process. Runs and tags and run attributes are inserted once per run and once per server, and variables, timesteps and timestep attributes are inserted once per timestep per server. Over 95% of the variable attribute write time is spent with the message waiting in the server's queue for it to begin executing the request. This time thus reflects the bottleneck effect of having only one server per thousand clients. Nevertheless, even with this very small dedicated resource requirement, the metadata introduces only a small performance penalty. As indicated in Table 4, the metadata writing constitutes only a small fraction of the total write time. Figure 12 demonstrates the performance for a small subset of EMPRESS's supported read functions. These perform more quickly than the write functions on average since reading utilizes 1/10th as many read clients but the same number of EMPRESS servers. These read functions thus allow users to query the high-level data contents of various datasets with very fast performance. These results show that EMPRESS introduces negligible storage, write and read overhead.

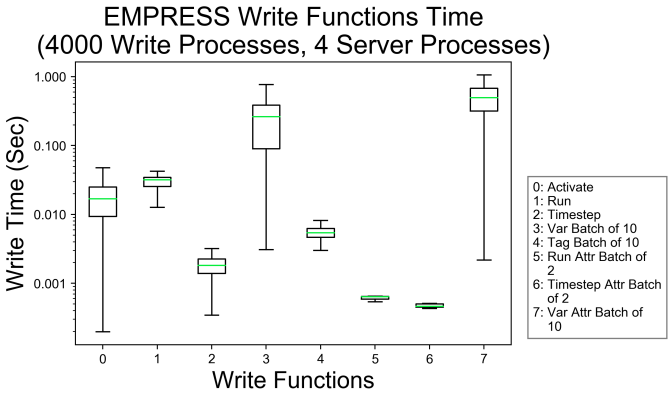


**Figure 11**

## 6.7 Accelerating Data Exploration

To evaluate how EMPRESS can be used to accelerate data exploration, we perform variations of read patterns 2 and 3 (reading an entire variable and reading three variables) using four different read

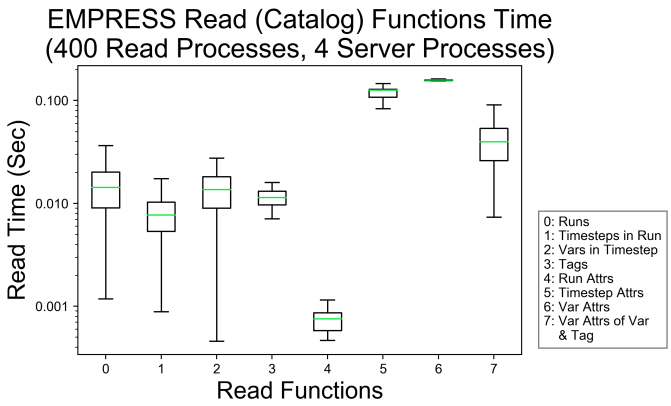


**Figure 12**

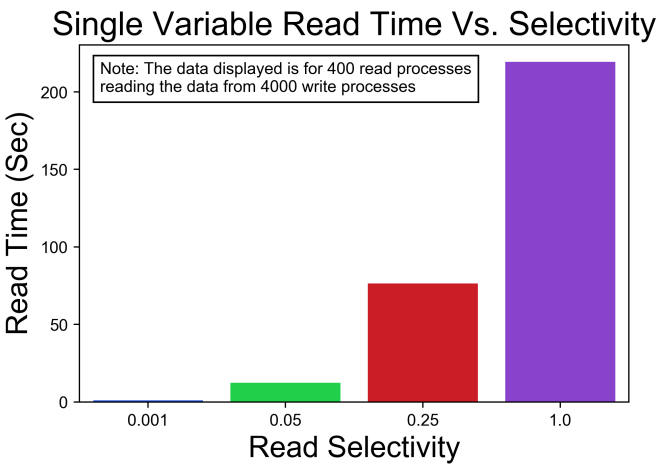


**Figure 13**

**Table 4: Data and Metadata Write Times at Different Scales**

| # Write Processes | # EMPRESS Servers | Data Write Time (Sec) | Md. Write Time (Sec) | Md. % of Write Time |
|---|---|---|---|---|
| **1000** | **1** | **1834.89** | **1.53** | **.09%** |
| **2000** | **2** | **3938.24** | **1.65** | **.04%** |
| **4000** | **4** | **8460.29** | **1.46** | **.02%** |

selectivities. First, the entire read patterns are performed without using EMPRESS. Then, EMPRESS is used to retrieve all attributes associated with a given tag for each pattern variable. HDF5 then reads in the data associated with these attributes. This process is performed for three different tags, which correspond to 25%, 5% and .1% of the data chunks. Figure 13 demonstrates the resulting performance for the 400 read client case (where the data and metadata were written by 4000 write processes). An important takeaway from the diagram is that reducing the read volume does not produce a 1-to-1 reduction in the total read time. This finding is due largely to the performance impact of HDF5 having to linearly search its stored metadata to identify the correct chunks to read, and due to the fact that since chunks are distributed randomly, the read could result in storage system contention. However, the read times are still significantly accelerated. The results demonstrate that EMPRESS can be used to store and query areas of interests leading to accelerated read performance for subsequent data analysis.

## 6.8 Comparison to Alternatives

*Writing.* Figure 14 demonstrates the performance comparison for writing metadata with EMPRESS and HDF5. At small scales, HDF5 actually out performs EMPRESS. This performance is the result of three different effects. First, in HDF5, basic metadata for runs, timesteps and variables are written as part of creating the associated files and data structures, making it impossible to disentangle the metadata write time from the other performance requirements. It is therefore not included in the numbers presented in the table. Second, as mentioned above, we have omitted transaction management from the HDF5 example since all metadata is managed internally (it is embedded in the same structure as the data it describes). Third, and most importantly, since only a single process can write metadata, the clients first coordinate to send their variable attributes to a single process. At small scales, due to the smaller number of coordinating processes and the small total metadata volume, this global coordination actually takes less time than performing thousands of smaller writes to the EMPRESS servers. While it is possible to perform client-side aggregation of attributes with EMPRESS, this was not performed in the testing harness. However, we see that as the number of processes increases, the HDF5 metadata write performance degrades while the EMPRESS performance remains relatively constant if the client-server ratio is held constant. Thus, ultimately, HDF5 suffers from the fact that it has no means of scaling out and parallelizing its metadata operations.

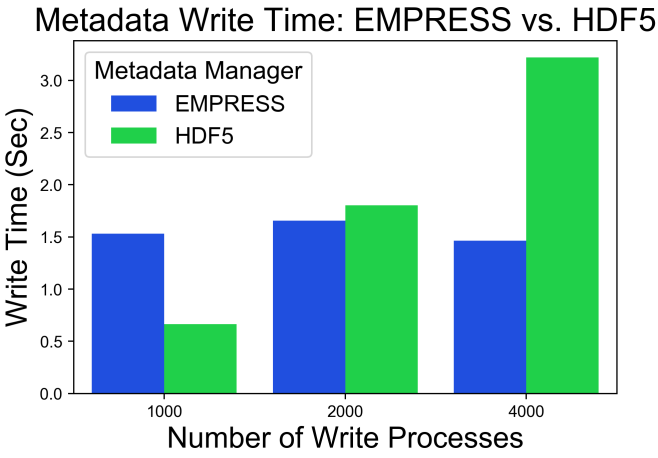


**Figure 14**

*Reading.* Figure 15 demonstrates the summed metadata query time that each system uses to perform the three metadata queries for pattern 2 (retrieving all attributes for a given variable that match a tag of 25%, 5% and .1% selectivity). The demonstrated results are for the 100:1 read client to EMPRESS server ratios (corresponding to the 1000:1 write process to server ratios). The results demonstrate that while EMPRESS can perform the queries very efficiently, HDF5 cannot. This performance is due to two different effects. One, HDF5 has no means of parallelizing attribute querying, meaning that a single process must crawl the entire attribute table. Two, HDF5

does not offer metadata indexing, and thus must perform a linear search. As a result of these two effects, it is understandable that the search performance for HDF5 varies linearly with the amount of metadata written (and thus searched). This performance stands in large contrast to the fast, relatively constant performance that EMPRESS can offer by using a small number of dedicated resources.

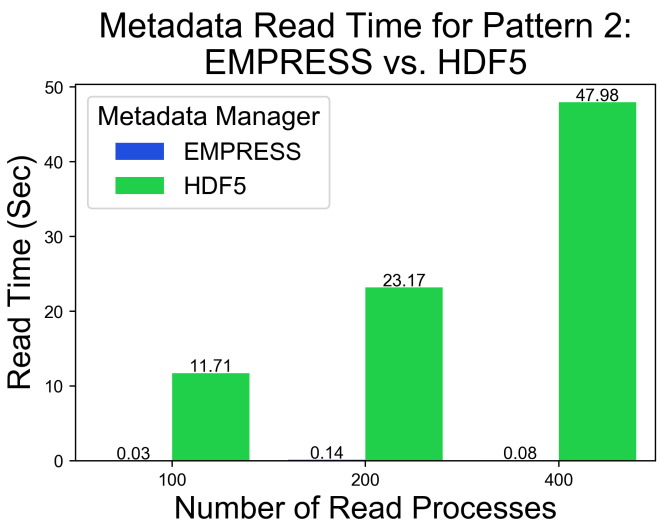


**Figure 15**

# 7 OBJECTOR EVALUATION

## 7.1 Goals

To evaluate the Objector, we compare it to the alternative of storing object names directly. To this end, we had three specific goals for evaluation:

- *Storage:* Compare the storage requirements for storing the Objector function with the storage requirements for storing object names directly. Since the Objector can generate all object names, the only storage cost is the size of the text representation of the Objector function.
- *Write Performance:* The Objector can be used when a simulation is writing to turn an array of data into a set of objects. It does this by providing a list of object names and a list of where to slice the data for each object. The alternative is to generate a data decomposition scheme and set of object names using some other means, and to store the object names directly. We want to compare the performance of the Objector to the performance of directly writing the object names to storage.
- *Read Performance:* The Objector can be used during reading to convert a set of spatial coordinates into a list of object names and the matching data offsets within the associated objects. The alternative is to search all stored object names or associated metadata for query matches. We want to compare the performance of using the Objector to querying a stored list of object names.

## 7.2 Testing Configurations

The evaluation attempts to simulate how the Objector would be used when a set of processes are writing a simulation output to storage or are reading part of a simulation output for analysis. This is then compared to the alternative of storing object names directly by storing and querying object names in a local SQLite[36] in-memory database. As discussed above (see Section 5.1), SQLite is a light-weight database with fast performance and minimal storage and performance overheads and thus represents as fair as possible a comparison. The Objector testing harness consists of the same simulation space used in the EMPRESS-HDF5 harness, meaning that the simulated application run is composed of three timesteps, each of which are composed of 10 3D variables that are decomposed using a 3D domain decomposition into chunks of 125x160x250 data points. The testing runs simulate the processes needed to write a simulation space with these characteristics as a set of objects for 1000, 2000, 4000, 8000, and 16000 write processes. However, unlike in the EMPRESS-HDF5 testing harness, no data is written. Instead only the generation, storage, and retrieval of the object names associated with the data is considered. The object size is set to 8MB since this is the ideal size used by Ceph [54], the targeted storage backend for integration with EMPRESS. In addition, only a single process is used to actually generate or write these object names and subsequently to generate or query them for reading. This is to provide as generous as possible a comparison to SQLite since using multiple tables or having multiple processes share a table introduces additional overheads either in storage, performance, or both for SQLite.

## 7.3 Write Process

Both versions of the testing harness simulate the steps needed to convert a chunk of array data into a set of objects with unique object names. Thus, for each timestep, each (simulated) process generates the object names associated with its assigned section of the simulation space for each variable. In both cases, this is done using the Objector. For each of these chunks, the Objector returns the list of matching offset names and where the process should slice its data array to divide its data into these objects. For the SQLite version of the harness, there is an additional step where each (simulated) process writes each of these object names into a local, in-memory database along with a few other identifiers such as the associated timestep and variable identifiers and the logical spatial dimensions (used to speed querying later). Indices are created on the timestep and variable identifiers to additionally speed querying. It merits discussion that the SQLite version of the testing harness uses the Objector. Although some system would be needed to generate a set of object names and the data decomposition, there could be a more efficient system for doing this. Therefore, we will not penalize the SQLite version for the time it spends running the Objector and instead will compare the time needed to run with Objector as a standalone process with just the time needed to write the names to SQLite.

## 7.4 Read Process

Both versions of the testing harness simulate the process needed to read a subsection of the simulation space for analysis by turning

a set of logical spatial dimensions of interest into a list of matching object names. The reading section utilizes the same six read patterns discussed above (see Section 6). Only a single process is used to read. Unlike in the writing section, this process does not simulate hundreds of read processes and instead queries the entire read pattern at once. This again provides the most generous possible comparison to SQLite, since for each simulated read process SQLite would have to traverse the entire object name table to answer the query. In the Objector version of the testing harness, for each of the six read patterns the process passes the logical dimensions to the Objector to generate the matching object names and the data ranges for each of the associated objects that match the query. In the SQLite version of the harness, the logical dimensions along with the timestep and variable identifiers are used to query the object name table and retrieve matching names.

## 7.5 Objector Write and Read Examples

*7.5.1 Objector Writing Example.* An example of how the Objector is used to write data is demonstrated in Algorithm 5. A process passes its assigned chunk coordinates to the Objector and receives a list of object names, offsets, and counts in return. The offsets and counts indicate the range of $X$, $Y$, and $Z$ data index values associated with each object. Since, as discussed above (see Section 4), the Objector uses a 1D domain decomposition, each object is associated with a single, contiguous slice of the data. For each object name, this slice of data is extracted, and then the object is written.

**Algorithm 5** Objector Writing algorithm

```
1: procedure ObjectorWrite
2:     retrieve_obj_names (chunk_coordinates)   ▷ Use Objector
3:     for all objNames do
4:         objData = slice_chunk_data (...)
5:         obj_storage_write (objName, objData)
6:     end for
7: end procedure
```

*7.5.2 Objector Reading Example.* An example for how the objector is used to read data is demonstrated in Algorithm 6. First, the users identifies a set of logical, spatial coordinates of interest, for example, by retrieving a set of variable attributes of interest using EMPRESS. Then, one at a time, the set of coordinates are given to the Objector along with the associated variable, timestep, and run identifiers. For each set of coordinates, the Objector returns the matching object names, offsets, and counts. For each matching object name, the offsets and counts are used to slice the data (filtering out the data points that do not match the spatial coordinates). This data is placed into the appropriate location in an array so that, once all of the objects for a set of coordinates have been read, the array will contain all of the requested data in the correct order.

## 7.6 Results

Figure 16 demonstrates the evaluation of storage sizes for the Objector and SQLite tests. As explained previously, one of the strengths of the Objector is that it is an $O(1)$ storage solution. Since it is a function that is stored as a string, it requires a fixed number of bytes

**Algorithm 6** Objector Reading algorithm

```
1: procedure ObjectorRead
2:     find_spatial_coordinates_of_interest (...) ▷ Using EMPRESS
3:     for all coordinates do
4:         retrieve_obj_names (coordinates)        ▷ Use Objector
5:         for all names do
6:             data = obj_storage_read (name)
7:             slice_data (...)
8:         end for
9:         patch_data (...)
10:    end for
11: end procedure
```

(around 500) independent of the size of the simulation space and the number of associated object names. The storage size for SQLite, by contrast, grows linearly with the number of object names. While the 0.2GB required in the 16000 write process case is a relatively trivial size, it is important to keep in mind that this test represents only a small fraction of what on object storage system on an exascale machine will have to support. While the 16000 write process case writes 2.4 million object names, the systems we are targeting are expected to manage around 10 trillion object names. At this scale, object names would use around 1 PB of storage.

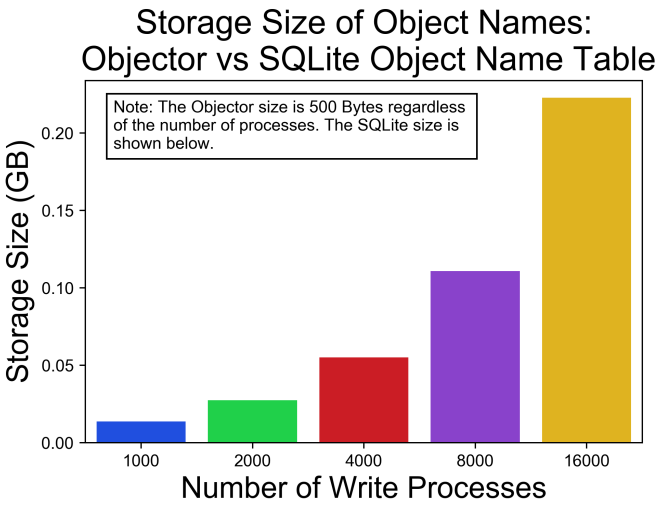


**Figure 16**

Figure 17 demonstrates how performance for the Objector (generating object names) and the SQLite comparison (writing object names to an in-memory database) vary with the number of write processes simulated. As expected, the Objector performs much faster than the SQLite alternative since the object name generation time is not dependent on the size of the simulation space or the number of corresponding object names. This time could be dramatically reduced through parallelization, but still presents a dramatic performance difference that will be exacerbated at larger scales.

Figure 18 demonstrates the Objector and SQLite performance for the 16000 client case on the six read patterns. Both provide roughly equivalent performance on the first three patterns, each of which involves reading one or more variables in their entirety.

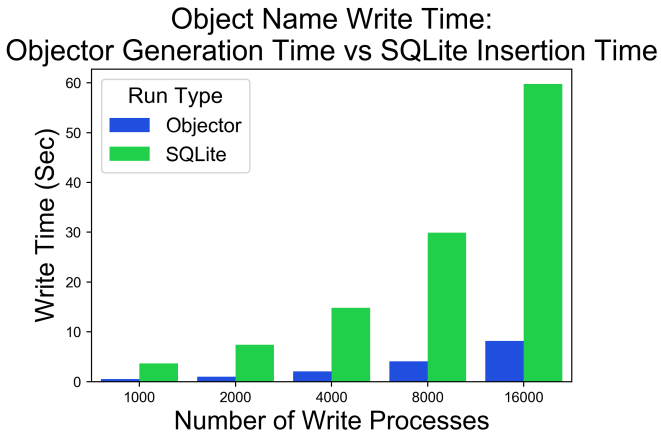


**Figure 17**

This represents the worst-case scenario for a single contiguous read using the Objector since it must generate all object names for the requested variables. However, patterns 4, 5, and 6 demonstrate the strength of the Objector. When reading subsets of a variable, the Objector has to generate only the matching object names, whereas SQLite has to search the entire table. Thus, the Objector provides as-good performance at the tested scales for producing the matching object names for entire variable reads, but performs significantly faster for reading subsets of variables. In addition, although SQLite performs quite well at the tested scales, it was not designed to operate at the petascale or exascale level. Therefore, it is quite likely that as the process count continues to increase, the performance will start to degrade.

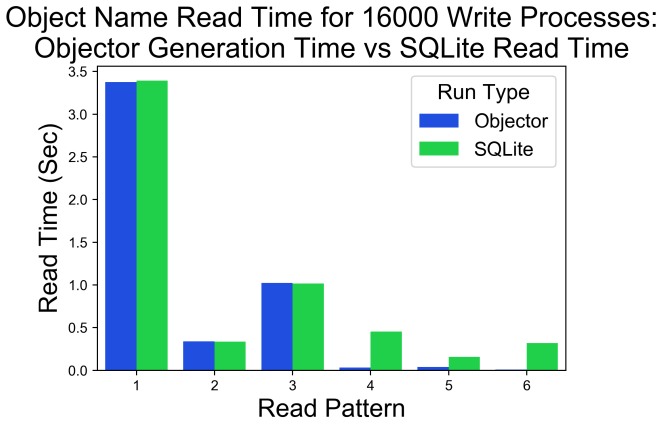


**Figure 18**

## 8 DISCUSSION

Testing shows that EMPRESS can efficiently support a wide variety of basic and custom metadata operations using minimal dedicated resources. Testing further shows that these metadata services can be used to identify regions of interest within a dataset and thereby reduce the reading scope and required read time. This demonstrates that EMPRESS can be used to accelerate scientific discovery, analysis and visualization and serves as an argument more generally for the importance of robust metadata services in scientific data management. Testing also demonstrates that there is still room for improvement. Using minimal dedicated resources such as one server process per thousand client processes results in the clients spending the bulk of the metadata writing time waiting for the servers. Means of reducing this time will be explored in the future with possible solutions including client-side aggregation and the possibility of using local EMPRESS servers. Although EMPRESS's design and this paper have focused on its use for scientific simulations, EMPRESS has implications for other domains. In particular, it can easily be used for scientific data collection, such as astronomy photographs or weather-related sensor data.

The Objector is an example solution to the exascale object naming problem. In particular, when coupled with EMPRESS, it provides a picture of what rich metadata can offer: the ability to externally manage and query metadata and the ability to map this metadata to the associated data without having to store the physical locations of data objects. This provides users with a highly efficient means to explore high level data contents and then determine which subsets are of sufficient interest to read in.

## 9 RELATED WORK

Related work falls into four general categories: storage systems, metadata management and indexing systems at the storage level, I/O systems, and metadata management and indexing systems at the I/O level.

### 9.1 Storage Systems

There are many widely used large-scale parallel and distributed storage systems such as Lustre [41], GPFS [40], GoogleFS [13], Ceph [54], PVFS [39], and HDFS [44]. However, these are largely focused on offering scalable storage and fault-tolerance. As a result, they generally only maintain system metadata, such as the names and access permissions of data objects, which is static and not extensible. While some of these systems offer extended or user-defined attributes, these are not scalable. They also do not offer "tagging" and "searching" capabilities, which are crucial for discovery [48]. In addition, many file systems already suffer from metadata bottlenecks because they are limited to one or very few metadata management servers [29].

One storage system that provides more extensive metadata capabilities is HP StoreAll with ExpressQuery [19]. HP StoreAll ExpressQuery is a production archival storage system that uses a distributed database for metadata management. It supports file tags and tag searches, but suffers a number of limitations. The tags are stored as string key-value pairs, which, as discussed before (see Section 3.3.1) and with SoMeta below, comes with a number of disadvantages. Because of this design, all queries must be phrased as equals, range, or substring queries. ExpressQuery puts the burden on the user to formulate the queries and remember key names. In addition, these tags are associated with entire files. This cannot offer sufficiently fine-grained indexing for simulation outputs, which tend to put an entire variable or even entire timestep in a file. This system also requires the users to keep track of the file system structure for queries instead of abstracting these low-level details away from the user.

## 9.2 Metadata Management and Indexing at the Storage Level

Recently, some work has been done to extend storage systems to offer "tag and search" capabilities. A notable recent example is TagIt [45]. Hosting the metadata management system inside the storage system allows metadata updates to be automatically triggered when data objects are written or updated. This provides increased consistency. However, there are costs associated with operating the metadata system at such a low level. First, scientific users generally use I/O systems and do not interact with the storage system directly. Therefore, unless the I/O system is extended to interface with the metadata management system, users will not be able to leverage the capabilities. Second, these systems tend to be dependent on the specific storage back-end, limiting portability. Third, as discussed above, many file systems already experience a severe metadata bottleneck, and have no means of dynamically increasing the number of metadata servers to meet demand. TagIt points out that it can only be used with very specific storage backends and its evaluation demonstrates severe scalability limits. Fourth, managing metadata on compute nodes offers greater processing power and faster interconnects than storage nodes can offer. A few TagIt specific problems are that it does not support sub-file indexing, is reliant on the file system's extended attributes, and supports very few search queries.

## 9.3 I/O Systems

The four most common I/O systems used by scientific simulations are ADIOS [30], PnetCDF [25], HDF5 [12], and netCDF-4 [38]. While each of these offer user-defined metadata attributes, they do not support scalable metadata services such as tagging, searching, or indexing. Furthermore, for each of these, metadata is embedded in associated data files, which results in several of the issues discussed above (see Section 2 bullet 2). In addition, each of these only uses a single client to write and read all metadata, adding a severe scalability bottleneck. These systems are simply not designed for large-scale metadata access or to use rich metadata as a tool for finding interesting data.

SciDB [5] is a popular I/O system in the astronomy community. Unlike the others, it is an array-based system with a database backend. However, it does not offer any kind of user-defined attributes and concentrates instead on basic metadata.

## 9.4 Metadata Management and Indexing at the I/O Level

Many systems use indexes to offer point-by-point data searches with a combination of variable, spatial (index), and temporal constraints. Examples include FastBit [56], FastQuery [16], and ISABELA-QA [15] (a block-index hybrid). However, none of these have been extended to support these kinds of rich querying services for metadata. In addition, bitmap based indexes (such as FastBit and FastQuery) also suffer from very large index sizes, with indexes typically ranging from 30%-200% of the original dataset size.

Property graphs have recently been suggested as a flexible means of offering metadata management with tight data coupling. However, efficient searching (graph traversal) has yet to be addressed, and many of these systems are still in the early development stage [8].

SoMeta [48], an object-based metadata management service, offers many similar capabilities to EMPRESS. Like EMPRESS, it can provide a wide range of tag-based searches. However, the two have a fundamentally different design. While EMPRESS uses an RDBMS backend with indexes to speed querying, SoMeta uses a distributed hash table backend (without indexes). Since SoMeta relies on a hash table, it is subject to many of the tradeoffs described above (see Section 3.3.1). In particular, SoMeta is optimized for performing queries for a specific metadata object name, but must resort to crawling the entire hash table if only parts of the name are provided. If the object name is not a search constraint, it must crawl all metadata objects. Quite simply, there is no good way to optimize a single key for many kinds of constraints (such as spatial, variable, temporal, and value). Since all metadata tags and values are stored as a string, it must also perform extensive string searching and matching for partial name, tag or value based constraints. In addition, SoMeta suffers from a usability problem: users are required to give each metadata object a unique name and to remember this name. SoMeta does not offer any mechanism to catalog the metadata names, tags, or timesteps stored in the hash table. SoMeta also does not support transactions at this time, limiting its use in workflows. Nevertheless, SoMeta offers some useful services such as using servers locally that EMPRESS will explore in the future (see Section 10).

Several related tools have been developed for a specific domain or application such as the Catalog Archive Server (CAS) [49] for the Sloan Digital Sky Survey (SDSS), ATLAS [4] for the Large Hadron Collider, and the Atmospheric Data Discovery System (ADDS) [37]. The Catalog Archive Server offers a very extensive range of queries and has been very carefully tuned. However, it is entirely domain-specific, with all of its metadata and queries designed for 2D astronomy photos. ATLAS offers a much less domain specific metadata model and supports a wide range of queries, but does not support user-defined attributes or tags. Its metadata model includes an "events" category for tagged phenomena, but the tags supported and the conditions under which they are added are entirely controlled by the system. The Atmospheric Data Discovery System (ADDS) supports rich metadata storage and querying, but also does not support user-defined metadata. Instead, all of its metadata is automatically extracted from existing datasets. This could be a useful capability, but does not replace the need for user-defined metadata.

Another category of related tools uses an external system to manage information about the storage locations of data objects. An example is the Scientific Data Manager (SDM). The SDM uses a database to store metadata about the physical locations of data objects and abstracts away low-level storage details from the user. It also supports some basic attribute capabilities, but these are very limited.

## 9.5 SIRIUS

In addition to these four general categories of related work, it merits discussion that EMPRESS is part of a larger work, the DOE Office of Science ASCR SIRIUS project. SIRIUS is designed to provide a system for scientific data management at exascale. It therefore seeks to

address all of the challenges outlined in the introduction including the I/O bottleneck, the need for efficient search and discovery, and severe storage space constraints. Specific contributions of SIRIUS include predictable storage performance (provided by Ceph [54]), extensible custom metadata (provided by EMPRESS), and data reduction techniques. Predictable storage performance allows users to make informed decisions about how frequently to output data. This is especially important since most users want to ensure that only a small, fixed percentage of time is devoted to I/O (instead of computation). As has been discussed in this paper, EMPRESS can be used to accelerate data exploration. SIRIUS also offers data reduction techniques that can provide high compression with small error bounds (reducing storage volume) and different level-of-precision read options, speeding data access for lower precision reads.

## 10 FUTURE WORK

EMPRESS is a work in a progress. While the new version of EMPRESS has greatly improved upon its predecessor, there are a few issues for the next generation of EMPRESS to address. To support exascale scientific simulations with rich annotations, EMPRESS will need to establish when a server should start writing to a new database (dividing the databases into multiple data objects for storage) and how to track these various pieces. This is not an issue at moderate scales since a single process typically has 1 to 4GB of RAM and possibly other memory at its disposal but will be important for exascale. It will also be important for EMPRESS to continue to improve fault-tolerance, such as addressing the best method to recover metadata operations that occur between checkpoints to disk and identify serve failure. EMPRESS will also continue to examine the best strategy for load balancing when the number of servers does not equal the number of databases.

We will also explore extending EMPRESS to offer additional functionality. We have recently extended EMPRESS to offer a preliminary local-server version. This was not the original design point due to the performance penalties associated with storing many small database files or joining many small databases into a large database. However, we will be further exploring and extending this this new version of EMPRESS in the future. Recent work has already begun to look at expanding the Objector to new domain areas [34]. Similarly, we will investigate the possibility of offering direct support for coordinate systems other than Cartesian and for supporting non-uniform meshes and Adaptive Mesh Refinement codes. We will also look into the possibility of offering more robust provenance tracking. Another interesting area to explore is point-by-point indexing. As demonstrated in the evaluation section, clients can store maximum and minimum values to create a sort of block index on the data. An I/O system could use this functionality to narrow the search space and then perform candidate checks on matching chunks. In the future, we will consider offering this as an explicit service. We will also consider making EMPRESS further extensible by offering direct support for users to generate their own queries. Other functionalities we will consider adding are support for multi-variable and multi-tag queries, more robust queries on entire runs, update functions, and delete functions for just attributes.

There are also a number of design or implementation decisions that we need to explore more fully. Virtually all design decisions come with inherent trade-offs. One decision to explore is under what conditions metadata is better suited for NoSQL databases such as MongoDB [2] and Cassandra [22] rather than relational databases. Another decision to explore is if there is a better way of distributing metadata across the servers.

Finally, there is work to be done to integrate EMPRESS with other related tools. For example, since EMPRESS is reliant on the ability of users to identify and store areas of interest, we can work with domain scientists and machine learning experts to ensure that we are fully supporting their needs. In addition, we have plans to integrate EMPRESS with ParaView [11] or [21] to allow EMPRESS's metadata to be used for interactive, visual exploration of datasets.

## ACKNOWLEDGEMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This work was supported under the U.S. Department of Energy National Nuclear Security Agency ATDM project funding. This work was also supported by the U.S. Department of Energy Office of Science, under the SSIO grant series, SIRIUS project and the Data Management grant series, Decaf project, program manager Lucy Nowell.

## REFERENCES

[1] [n. d.]. ([n. d.]). http://www.sqlite.org/
[2] [n. d.]. MongoDB. ([n. d.]). http://www.mongodb.com/
[3] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. 2010. Datastager: scalable data staging services for petascale applications. *Cluster Computing* 13, 3 (2010), 277–290.
[4] Solveig Albrand, Thomas Doherty, Jerome Fulachier, and Fabian Lambert. 2008. The ATLAS metadata interface. In *Journal of Physics: Conference Series*, Vol. 119. IOP Publishing, 072003.
[5] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 963–968.
[6] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. 2017. Tuning HDF5 subfiling performance on parallel file systems. (2017).
[7] Jacqueline H Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. 2009. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery* 2, 1 (2009), 015001.
[8] Dong Dai, Robert B Ross, Philip Carns, Dries Kimpe, and Yong Chen. 2014. Using property graphs for rich metadata management in hpc systems. In *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE, 7–12.
[9] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.
[10] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (2012), 163–181.
[11] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Gevecik, Michel Rasquin, and Kenneth E Jansen. 2011. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 89–96.
[12] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 36–47.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, 29–43. http://doi.acm.org/10.1145/945445.945450

[14] Zhenhuan Gong, David A Boyuka II, Xiaocheng Zou, Qing Liu, Norbert Podhorszki, Scott Klasky, Xiaosong Ma, and Nagiza F Samatova. 2013. Parlo: Parallel run-time layout optimization for scientific data explorations with heterogeneous access patterns. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 343–351.

[15] Zhenhuan Gong, Sriram Lakshminarasimhan, John Jenkins, Hemanth Kolla, Stephane Ethier, Jackie Chen, Robert Ross, Scott Klasky, and Nagiza F Samatova. 2012. Multi-level layout optimization for efficient spatio-temporal queries on isabela-compressed data. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 873–884.

[16] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, and Wes Bethel. 2006. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *Scientific and Statistical Database Management, 2006. 18th International Conference on*. IEEE, 149–158.

[17] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 2–1–2–26. https://doi.org/10.1145/1238844.1238846

[18] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. Boyuka, T. Rogers, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova. 2012. Byte-precision level of detail processing for variable precision analytics. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 1–11. https://doi.org/10.1109/SC.2012.26

[19] Charles Johnson, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, Alistair C Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J Doyle, et al. 2014. From research to practice: experiences engineering a production metadata database for a scale out file system.. In *FAST*. 191–198.

[20] S Ku, CS Chang, and PH Diamond. 2009. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion* 49, 11 (2009), 115021.

[21] T Kuhlen, R Pajarola, and K Zhou. 2011. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*.

[22] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 5–5.

[23] Sriram Lakshminarasimhan, John Jenkins, Isha Arkatkar, Zhenhuan Gong, Hemanth Kolla, Seung-Hoe Ku, Stephane Ethier, Jackie Chen, Choong-Seock Chang, Scott Klasky, et al. 2011. ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 31.

[24] Margaret Lawson, Craig Ulmer, Shyamali Mukherjee, Gary Templet, Jay Lofstead, Scott Levy, Patrick Widener, and Todd Kordenbrock. 2017. Empress: Extensible Metadata Provider for Extreme-scale Scientific Simulations. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS '17)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145\/3149393.3149403

[25] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Supercomputing, 2003 ACM/IEEE Conference*. 39–39. https://doi.org/10.1109/SC.2003.10053

[26] Jay Lofstead and Jai Dayal. 2012. Transactional parallel metadata services for integrated application workflows. *HPCDB'12* (2012).

[27] Jay Lofstead, Jai Dayal, Karsten Schwan, and Ron Oldfield. 2012. D2t: Doubly distributed transactions for high performance and distributed computing. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 90–98.

[28] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. 2011. Six degrees of scientific data: reading patterns for extreme scale science IO. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC '11)*. ACM, 49–60. http://doi.acm.org/10.1145/1996130.1996139

[29] Jay Lofstead and Robert Ross. 2013. Insights for Exascale IO APIs from Building a Petascale IO API. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 87, 12 pages. https://doi.org/10.1145/2503210.2503238

[30] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. 2009. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*.

[31] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing variability in the IO performance of petascale storage systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 1–12.

[32] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1039–1065.

[33] Grzegorz Malewicz, Ian Foster, Arnold L Rosenberg, and Michael Wilde. 2007. A tool for prioritizing DAGMan jobs and its evaluation. *Journal of Grid Computing* 5, 2 (2007), 197–212.

[34] Reza Nasirigerdeh, Michael A Sevilla, Margaret Lawson, Noah Watkins, Latchesar Ionkov, Jeff LeFevre, Jay Lofstead, Carlos Maltzahn, Joel Armstrong, and Mark Diekhans. 2018. Dataset Striping: Mapping Scientific Datasets to Programmable Storage Systems. In *Proceedings of the 9th ACM Symposium on Cloud Computing 2018 (SoCC '18)* in submission.

[35] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. 2006. Efficient Data-Movement for Lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*. Barcelona, Spain.

[36] Michael Owens. 2003. Embedding an SQL database with SQLite. *Linux Journal* 2003, 110 (2003), 2.

[37] Sangmi Lee Pallickara, Shrideep Pallickara, and Milija Zupanski. 2012. Towards efficient data search and subsetting of large-scale atmospheric datasets. *Future Generation Computer Systems* 28, 1 (2012), 112–118.

[38] R Rew, E Hartnett, J Caron, et al. 2006. NetCDF-4: Software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*.

[39] Robert B Ross, Rajeev Thakur, et al. 2000. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*. 391–430.

[40] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *FAST*, Vol. 2.

[41] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[42] Michael A Sevilla, Reza Nasirigerdeh, Carlos Maltzahn, Jeff LeFevre, Noah Watkins, Peter Alvaro, Margaret Lawson, Jay Lofstead, and Jim Pivarski. 2018. Tintenfisch: File System Namespace Schemas and Generators. In *The 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2018)*.

[43] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A programmable storage system. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 175–190.

[44] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 1–10.

[45] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Geoffroy R Vallée, Seung-Hwan Lim, and Ali R Butt. 2017. Tagit: an integrated indexing and search service for file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 5.

[46] Gurmeet Singh, Shishir Bharathi, Ann Chervenak, Ewa Deelman, Carl Kesselman, Mary Manohar, Sonal Patil, and Laura Pearlman. 2003. A metadata catalog service for data intensive applications. In *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 33–33.

[47] Yu Su and Gagan Agrawal. 2012. Supporting user-defined subsetting and aggregation over parallel netcdf datasets. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 212–219.

[48] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. 2017. SoMeta: Scalable Object-centric Metadata Management for High Performance Computing. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 359–369.

[49] Ani R Thakar, Alex Szalay, George Fekete, and Jim Gray. 2008. The catalog archive server database management system. *Computing in Science & Engineering* 10, 1 (2008).

[50] Craig Ulmer, Shyamali Mukherjee, Gary Templet, Scott Levy, Jay Lofstead, Patrick Widener, Todd Kordenbrock, and Margaret Lawson. 2018. Faodel: Data Management for Next-Generation Application Workflows. In *Proceedings of Workshop on Infrastructure for Workflows and Application Composition (IWAC), 2018*.

[51] WX Wang, ZTWM Lin, WM Tang, WW Lee, S Ethier, JLV Lewandowski, G Rewoldt, TS Hahm, and J Manickam. 2006. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas* 13, 9 (2006), 092505.

[52] Yi Wang, Yu Su, and Gagan Agrawal. 2013. Supporting a light-weight data management layer over hdf5. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 335–342.

[53] Noah Watkins. 2013. Dynamic Object Interfaces with Lua. (Oct 2013). https://ceph.com/geen-categorie/dynamic-object-interfaces-with-lua/

[54] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[55] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.
[56] K Wu, S Ahern, E W Bethel, J Chen, H Childs, C Geddes, J Gu, H Hagen, B Hamann, J Lauret, J Meredith, P Messmer, E Otoo, A Poskanzer, O RÃijbel, A Shoshani, A Sim, K Stockinger, G Weber, W m Zhang, and et al. 2009. FastBit: Interactively Searching Massive Data. In *PROC. OF SCIDAC 2009*.
[57] Tzuhsien Wu, Jerry Chou, Shyng Hao, Bin Dong, Scott Klasky, and Kesheng Wu. 2017. Optimizing the query performance of block index through data analysis and I/O modeling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 12.
[58] Qing Zheng, George Amvrosiadis, Saurabh Kadekodi, Garth A Gibson, Charles D Cranor, Bradley W Settlemyer, Gary Grider, and Fan Guo. 2017. Software-defined storage for fast trajectory queries using a deltaFS indexed massive directory. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. ACM, 7–12.
[59] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2834976.2834984

# A APPENDIX

To illustrate an example of the Objector's algorithm, say that we are interested in the variable "temperature" from the timestep 0 of a run of "XGC" with the job scheduler identifier of 123456. These pieces of metadata are given to the Objector so that it can create a unique identifier (name) for this variable. Say that the "temperature" variable goes from $[1, 1, 1]$ to $[100, 100, 100]$, that each chunk has size $[10, 10, 10]$ data points and the ideal object size is 100 data points. Say furthermore that our identified coordinates of interest within the simulation space are $[51, 51, 51]$ to $[60, 60, 60]$. The following is the process that the Objector would use for generating the matching object names (for either writing or reading).

(1) Convert the coordinates to global offsets by subtracting the starting variable coordinate from the starting coordinate of interest
   - $[51, 51, 51] - [1, 1, 1] = [50, 50, 50]$
(2) Find the first chunk number that overlaps with the coordinates of interest (by dividing the global offset for the starting coordinate of interest by the uniform chunk size). Then, determine the chunk's starting coordinate using the uniform chunk size.
   - Chunk number: $[50, 50, 50]/[10, 10, 10] = 5$
   - Starting chunk coordinate: $5 * [10, 10, 10] = [50, 50, 50]$
(3) Find the first object number within this chunk that overlaps with the global offset for the starting coordinate of interest (using the uniform object size, divide the $X$-offset from the first overlapping chunk by the uniform object $X$-width). Then, determine the object's starting coordinate using the uniform chunk size.
   - Uniform object size: $[1, 10, 10]$ (since the Objector always uses the whole $Y$ and $Z$ chunk widths for an object)
   - Object number: $(50 - 50)/1 = 0$
   - Starting object coordinate: $0 * [1, 10, 10] + [50, 50, 50] = [50, 50, 50]$
(4) Iterate through all overlapping object coordinates by incrementing $X$ by the uniform object $X$-width and incrementing $Y$ and $Z$ by the fixed chunk $Y$ and $Z$ widths.
   - Second matching object's starting coordinates: $[51, 50, 50]$
   - ...
   - Ninth matching object's starting coordinates: $[59, 50, 50]$
(5) For each object, generate a name using the unique run, timestep and var identifiers, along with the starting coordinates.
   - First matching object name: XGC/123456/0/50/50/50
   - ...
   - Ninth matching object name: XGC/123456/0/59/50/50