MIQS: Metadata Indexing and Querying Service for Self-Describing File Formats

Wei Zhang
Texas Tech University
Lubbock, Texas
X-Spirit.zhang@ttu.edu

Suren Byna
Lawrence Berkeley National Laboratory
Berkeley, California
sbyna@lbl.gov

Houjun Tang
Lawrence Berkeley National Laboratory
Berkeley, California
htang4@lbl.gov

Brody Williams
Texas Tech University
Lubbock, Texas
Brody.Williams@ttu.edu

Yong Chen
Texas Tech University
Lubbock, Texas
yong.chen@ttu.edu

ABSTRACT

Scientific applications often store datasets in self-describing data file formats, such as HDF5 and netCDF. Regrettably, to efficiently search the metadata within these files remains challenging due to the sheer size of the datasets. Existing solutions extract the metadata and store it in external database management systems (DBMS) to locate desired data. However, this practice introduces significant overhead and complexity in extraction and querying. In this research, we propose a novel Metadata Indexing and Querying Service (MIQS), which removes the external DBMS and utilizes in-memory index to achieve efficient metadata searching. MIQS follows the self-contained data management paradigm and provides portable and schema-free metadata indexing and querying functionalities for self-describing file formats. We have evaluated MIQS with the state-of-the-art MongoDB-based metadata indexing solution. MIQS achieved up to 99% time reduction in index construction and up to 172k× search performance improvement with up to 75% reduction in memory footprint.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

Metadata Search, HDF5 Metadata Management

ACM Reference Format:
Wei Zhang, Suren Byna, Houjun Tang, Brody Williams, and Yong Chen. 2019. MIQS: Metadata Indexing and Querying Service for Self-Describing File Formats. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356146

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356146

1 INTRODUCTION

Large-scale scientific applications, including experiments, observations, and simulations, generate enormous amounts of data. [3, 4, 8, 11, 24, 25, 27, 31]. Due to upcoming construction of even larger scientific infrastructures and the increasing demand on the fine nature of the data, the volume of scientific data is expected to grow rapidly. Such growth imposes a substantial challenge to scientists - to survive in the ocean of data, and particularly, to efficiently find the data they require [38, 40, 43, 46, 48].

Self-describing data and file formats, such as HDF5 [13], netCDF [33], ADIOS-BP [26], and ASDF [14], are extensively used in scientific applications. In these file formats, the metadata of each data object is stored alongside the data object. This practice makes these formats both self-describing and self-contained. For self-describing and self-contained data management paradigm, it is ideal that the metadata search functionality is included as part of the self-describing and self-contained data management solution. However, most self-describing and self-contained data management solutions do not provide such internal metadata search functionality. Instead, finding interesting data in multiple files is achieved by iterating through all the files and performing pattern matching on the metadata one by one. Such a procedure is easy to implement and works well when there are a limited number of files and the amount of metadata is small. However, when the number of data objects in a file is very large, sifting through the metadata objects within one file becomes time-consuming. Many applications further compound this issue by generating a large number of data files. For instance, cosmology observations store the images of the sky either hourly or daily and stores all the datasets into multiple self-describing data files [24].

While metadata management and indexing have been well-studied in the context of file systems [23, 28, 38, 41, 47, 48], metadata indexing for self-describing data formats remains underdeveloped. Due to the lack of effective metadata search capabilities provided by the runtime libraries and tool-chains packaged with these self-describing data formats, scientists often use a separate database management system (DBMS) to facilitate the process of finding the required data. Typical examples can be seen from BIMM [20], EMPRESS [21], the SPOT Suite [9], and JAMO [18], where relational databases (e.g. SQLite [39], PostgreSQL [32]) and NoSQL databases

(e.g. MongoDB [29]) are used for maintaining the metadata and providing the metadata search functionality needed for locating the required data from the self-describing data files.

However, a database solution disjoint from the self-describing data files is incompatible with the self-describing and self-contained data management paradigm. This approach requires extra overhead in deployment as well as constant maintenance in order to keep the database updated. In addition, initial deployment of the database system for metadata search necessitates reading the metadata from the self-describing files and then loading it into the database. This duplicates the metadata at two places and hence leads to storage redundancy. Moreover, the database has to be either made available or migrated when the self-describing files are transferred to a different site, which is a complicated process, not to mention that the entire database has to be updated with the new location of file paths.

Toward the goal of providing a self-contained metadata search service for self-describing file formats, we propose *MIQS*, a novel Metadata Indexing and Querying Service. MIQS is designed to be a library that can be integrated into the existing self-describing and self-contained data management solutions. It provides metadata indexing and querying functionality for self-describing data formats in a way that complies with the self-describing and self-contained data management paradigm. By introducing an in-memory index structure, along with an on-disk persistence mechanism of the index, MIQS provides the capability to build portable indexes and an efficient metadata query service. Applications that utilize MIQS can perform metadata search without consulting an external DBMS. This philosophy complies with the self-describing and self-contained data management paradigm.

We have developed a prototype implementation of MIQS to support the HDF5 library and to study its effectiveness. We have conducted evaluation tests on the Edison supercomputer hosted at the National Energy Research Scientific Computing Center (NERSC). Our experiments against over 144 million attributes from 1.5 million objects in 100 astrophysics data files [24] show that, compared to a state-of-the-art MongoDB powered indexing solution, MIQS achieved up to 99% time reduction in index construction and searched metadata about 172k× faster. Moreover, MIQS used only up to 25% of the overall storage footprint that the MongoDB-based solution used for these datasets.

The contributions of this research are summarized as follows:

- We identify the drawbacks of existing DBMS-powered metadata indexing and querying solutions for self-describing file formats.
- Following the principle of self-describing and self-contained data management paradigm, we introduce an integrated, lightweight metadata indexing and querying service for self-describing data files. The in-memory indexing data structure achieves significantly better query performance without transforming or duplicating metadata, as compared to existing database-powered solutions. The index persistence mechanism also makes the index reconstruction easier and ensures the portability of the metadata search service.
- We develop a prototype implementation of MIQS to support the HDF5 library and conduct extensive evaluations to validate the design. The evaluations confirm that MIQS is efficient in building metadata indexes and offers better query performance than database-powered solutions, with less storage consumption. Additionally, MIQS is both portable and transparent in nature, which promotes the design philosophy of the self-describing and self-contained data management solution.

The rest of this paper is organized as follows. In Section 2, we review self-describing data formats such as HDF5 and existing solutions for metadata indexing and searching. We introduce the design of MIQS in detail in Section 3. In Section 4, we present the experimental evaluation results of MIQS. We conclude this research and discuss future work in Section 5.

2 BACKGROUND AND RELATED WORK

In this section, we briefly review self-describing data formats, using HDF5 as an example. We also discuss existing research and solutions that provide metadata search functionality for self-describing data files.

2.1 Self-Describing Data Formats and HDF5

Self-describing data formats, such as HDF5, netCDF, ADIOS-BP and ASDF, are designed to provide one-stop data management solution with no dependency on other data management solutions. In these data formats, the metadata is stored alongside the data itself, providing description, interpretation, and even definition of the data objects. This fusion between data and its metadata allows the self-describing formats to provide users with programming interfaces and tools that facilitate portable one-stop data management solutions. Applications that adopt these APIs and toolchains can be expected to behave in the same way regardless of where the applications are deployed. Data analysis performed on these self-describing files need not consult other data management solutions as the descriptive information is already contained in the data formats themselves.

HDF5 (Hierarchical Data Format version 5), a typical example of the self-describing data format, is one of the most frequently used data formats in scientific fields, which a large number of users rely on for scientific data management and exchanges [16]. When considering the case where we have a set of HDF5 files managed by the file system, there is a deep hierarchy across both the file system and also the HDF5 data format, as shown in Figure 1.

In a file system (e.g. GPFS [34] or Lustre [35]), each directory may contain sub-directories and also the HDF5 files. Within each HDF5 file, there are two types of objects - groups and datasets. The groups function as internal nodes within the hierarchy while the datasets play the role of leaf nodes. Attached alongside each of these data objects (either a group or dataset) are metadata attributes. Different objects may share the same set of attributes, but the value of these shared attributes can be different from one another. The hierarchical organization of these data objects introduces attribute inheritance between parental data objects and child data objects. In other words, the metadata attribute values of a group also apply to all child groups and datasets underneath.

Not all self-describing data formats follow the hierarchical organization. For instance, netCDF has a flat organization of all variables.

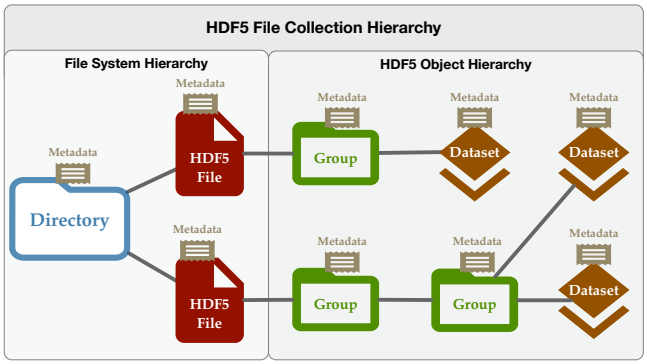


Figure 1: HDF5 file collection hierarchy

However, flat organization is a special case of hierarchical organization, where there is only one level in the hierarchy. As such, we consider metadata indexing and querying using a hierarchical organization to be generalizable to all self-describing data formats. In the rest of this paper, our discussion of metadata indexing and querying will focus on hierarchical organization formats such as HDF5.

2.2 Metadata Indexing and Querying over Self-Describing Files

The metadata in self-describing data formats provides users with descriptive information about the underlying dataset and therefore is prevalently used for finding required data. In self-describing data files, the metadata can be seen as a collection of attributes. Each attribute can be represented as a tuple $< k, v >$, where k represents the attribute name and v represents the attribute value. Finding required data can then be accomplished by issuing metadata queries that utilize these key-value pairs. These queries may search for the identifiers of data files or data objects that match given query. Each query contains query target $\mathcal{T}$ and query condition $\mathcal{Q} =< q_k, q_v >$, where q_k and q_v are the attribute name and attribute value, respectively, in the query condition.

Table 1: Structure of metadata queries with a query target and a query condition

$\mathcal{T}$	$\mathcal{Q} =< q_k, q_v >$
Data files	<BESTEXP, 113919>
Data objects	<AUTHOR, John>

According to the Digital Curation Conference (DCC) [10], the major data types of metadata attributes are strings and numbers. In particular, the attribute names are of strings and attribute values are either strings or numbers. Therefore, in this study, we focus on metadata queries with q_k to be a string and q_v to be a string or a number. For example, as shown in Table 1, one metadata query may ask for the identifiers of data files that contain attribute key-value pair <BESTEXP, 113919>, where attribute name "BESTEXP" is a string and its value "113919" is a number. Another metadata query may look up the identifiers of the data objects with attribute key-value pair <AUTHOR, John>, where "AUTHOR" is a string and "John" is also a string.

Most self-describing data formats come with libraries and tools that enable access to the metadata of each data object in a single file. However, many contemporary applications tend to store data into multiple self-describing files instead of a single file [24] in order to avoid performance issues and to be able to better cope with different file systems. In such scenarios, even with the libraries and tools provided by the self-describing data formats, finding the required data over a large number of self-describing files remains a challenging task. Due to the lack of libraries to provide metadata search capabilities over a collection of multiple self-describing files, scientists have to build external metadata search system on their own with the help of a database management system (DBMS).

Examples of such DBMS-powered approaches are BIMM [20], EMPRESS 2.0 [21], JAMO [18], and SPOT Suite [9]. In BIMM [20] and EMPRESS 2.0 [21], the metadata is stored in relational database management systems (RDBMS), such as PostgreSQL[32] and SQLite [39]. It is difficult, however, to accurately reflect the hierarchical organization of metadata for self-describing files using the tabular model of RDBMS. Some other solutions, such as JAMO [18] and SPOT Suite [9], use document database (MongoDB [29] in particular) to avoid the complexity and effort of transforming metadata into tabular format. In these MongoDB-powered solutions, metadata is transformed into BSON format [7] (the binary format of JSON [19]) first, and the BSON representation of the metadata is then stored in MongoDB for efficient metadata search. From the BSON documents in MongoDB, MongoDB indexes have to be created on specified attributes to accelerate queries against these attributes.

However, all database-powered solutions share common drawbacks. In order to explain these drawbacks, Figure 2 provides an abstract view of these existing DBMS-powered solutions. As shown in this figure, the self-describing data files are stored on a shared file system, and applications running on the HPC system can access the data and metadata in these files concurrently. Apart from the HPC system, an external, dedicated database system is built to facilitate metadata search. The metadata of the self-describing data files is first transformed into another format that complies with the data model of the database system, and is then imported into the database system. This operation duplicates the metadata of the self-describing data files and stores them into the external DBMS. To search metadata, applications on the HPC system have to issue queries through the client-side APIs of the DBMS. In this case, metadata queries and the querying results are transmitted over the network between the HPC system and the database. Such a solution disregards the self-containing property of self-describing data formats by introducing external data source for metadata management and hence suffers from the following drawbacks:

- **Performance issue:** Since applications need to communicate with the external database over the network, the communication overhead can be significant. Particularly, when applications have a high degree of concurrency, metadata search queries can overwhelm or throttle the network.
- **Storage redundancy:** Since the self-describing data files already contain the metadata, duplicating the metadata in

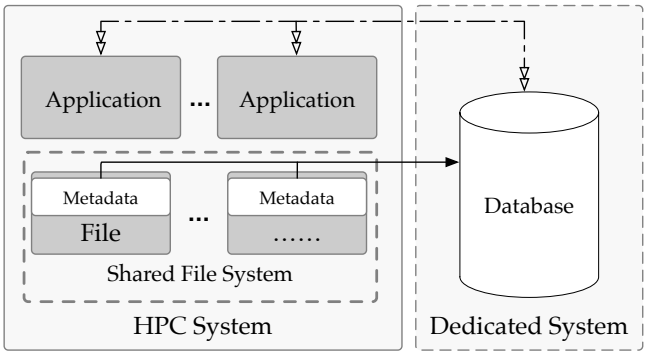


Figure 2: A high-level view of database-powered metadata search solution

any form in DBMS can introduce storage redundancy for storing an extra copy of the metadata.

- **Data model adaption:** The data model in a database is different from that of the metadata in self-describing files. In the aforementioned DBMS-powered solutions, the metadata in self-describing files has to be reformatted into either tables in RDBMS or BSON documents in MongoDB, which requires extra efforts and time in transforming data formats.
- **Schema constraints:** The data model in RDBMS or document-based databases require the user to know about the data schema used in these systems. For example, in a RDBMS, users have to know on which table an index should be built and from which table the required data records can be retrieved from. In MongoDB, the metadata is represented as BSON document where BSON objects follow the same hierarchical organization found in the original metadata. Therefore, when building index for the attributes, the users must have knowledge on the data schema of the BSON document and they have to know the paths to the data objects where the attributes belong to. As such, when issuing a metadata query in MongoDB, it is necessary for the users to specify the path to the attribute in the query condition. For example, if attribute "BESTEXP" belongs to both object "/3540/55127" and object "/3540/55127/100/coadd", users have to explicitly indicate two BSON paths for building the index on "BESTEXP":"3540.55127.BESTEXP" and "3540.55127.100.coadd.BES-TEXP". Accordingly, when performing a query for "BESTEXP", users still have to specify the path of the object containing "BESTEXP" [30]. However, when building the metadata index and issuing metadata queries, users are typically aware of only the attribute name (e.g. "BESTEXP") and attribute value (e.g. 113919). The hierarchical schema of the metadata is hidden from the users and the only way to know the path to the owner object of the attribute is to scan the entire dataset.
- **Maintenance demand:** The database system runs on top of the operating system. Consequently, any update to the database system or operating system can cause compatibility issues, which require non-trivial efforts in rectifying them.
- **Portability and mobility:** The external database and the metadata search solution are not seamlessly integrated into the self-describing data formats, which introduces complexity in deployment, and significantly affects the mobility of the data files and the portability of the metadata search solution.

3 METHODOLOGY

In this section, we introduce MIQS - a novel metadata indexing and querying service for self-describing data formats. We first present an overview of the MIQS design, and then introduce the index construction procedures of MIQS. Afterwards, we introduce how MIQS serves metadata queries.

3.1 Overview of MIQS Design

MIQS is designed to be a software library providing metadata indexing and querying service that complies with the self-describing and self-contained data management paradigm and can be integrated into the querying APIs or tools of existing self-describing data formats such as HDF5 querying module [15]. Without duplicating the metadata that is already included in the self-describing data files, it only maintains the index of the metadata to facilitate efficient metadata query. As shown in Figure 3, given a collection of self-describing files, MIQS allows each process to create and to maintain a copy of the in-memory index on specified attributes after scanning the data files. Since the index will maintain the relationship between attribute key-value pairs and file/object paths, even a full copy of the index will not consume a significant amount of memory. Once the in-memory index is built, processes can access in-memory index via the metadata search service provided by MIQS. In other words, each process is independent and self-contained. Different from some existing distributed metadata management solutions [17, 40, 42, 45, 47], MIQS follows a shared-nothing parallel architecture which ensures efficient and independent direct memory access for metadata search without network operations.

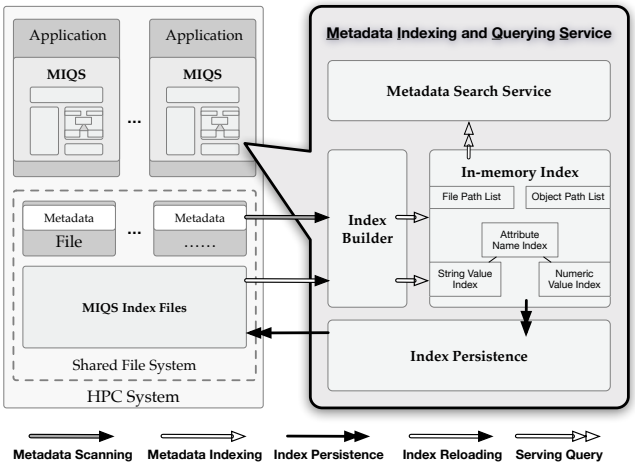


Figure 3: Overview of the architecture of MIQS-based solution

Processes with MIQS can also store the in-memory index to on-disk index files via the index persistence mechanism of MIQS. The MIQS on-disk file can be included as part of the self-describing file collection and can be transferred to any place alongside the self-describing data files. In addition, when transferring a self-describing dataset to a different HPC site, MIQS can quickly reload its on-disk index files into memory to reconstruct in-memory index.

This design eliminates the need for a dedicated external database system. Instead, metadata index files are maintained alongside the datasets while the in-memory index is integrated into existing solutions, which are completely self-describing and self-contained. As a consequence, improved portability and mobility are achieved. Also, in this design, there is no need to maintain databases or to know about data schema in order to use the metadata search service. This design also avoids the effort in adapting data models. Moreover, it does not duplicate the metadata in self-describing data files into other data sources like DBMS, which avoids storage redundancy. The search performance can also be expected to improve since there is no network communication between applications and an external database system.

As the datasets in most scientific applications are generated once and will be read many times for analysis without change [6, 12, 37], our current design does not yet support index updates if a self-describing dataset is modified while being searched. In such a scenario, MIQS needs to reconstruct the indexes to continue performing metadata searches against a modified dataset.

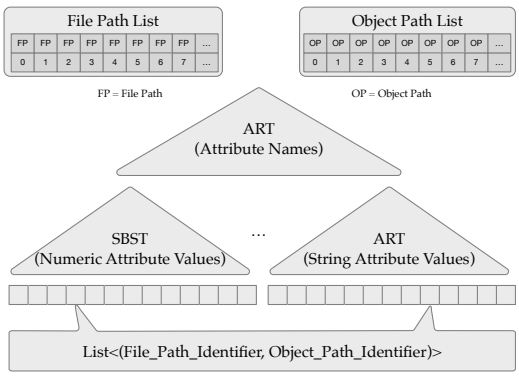


Figure 4: MIQS index data structures

3.1.1 In-memory Index. As shown in Figure 4, in MIQS, the in-memory index contains:

(1) Two path lists serving as mapping tables between path strings and integer identifiers. We use this data structure to store both global file path and global object path mapping tables. In this study, we use the tagged linked list from libhl [2] to implement this functionality.
(2) A root-level adaptive radix tree (ART) [22] for the attribute keys (name strings).
(3) A secondary tree-based index at each leaf node of the root-level ART. The secondary indexes are designed for indexing attribute values using a self-balancing search tree (SBST) [36] for numeric values and an ART for string values. (In MIQS, we use the red-black tree in libhl [2] for the implementation of SBST.)
(4) A list containing multiple tuples at each leaf node of each secondary index where each tuple contains the file path identifier and the object path identifier. These identifiers correspond to those found in the global file path lists and global object path list which helps avoid storing space-consuming strings, i.e. the repetitive file and object paths.

This combination of ARTs and SBSTs is able to support exact queries on attribute names, string attribute values and numerical attribute values. Also, it opens the possibility for prefix/suffix queries on attribute names and string attribute values as well as range query on numbers. In this study, our major contribution is to eliminate 3rd-party data sources and maintain the property of the self-contained data management paradigm. Therefore, we only discuss exact queries on strings and numbers in order to showcase the benefit of the self-contained and portable design of MIQS. Furthermore, as a result of the direct access to the local in-memory index, both index construction and metadata search are expected to be much more efficient than database-powered solutions.

3.1.2 Index Persistence Mechanism. The MIQS index files are designed to permanently store indexes, alongside the corresponding self-describing datasets. An index file can then be loaded into memory quickly for metadata searches when an application uses the dataset. It can also be used to help with the recovery of the metadata querying service if the application process is terminated. These index files can be transferred along with the self-describing data files, so that applications integrated with MIQS can perform efficient metadata queries with minimal effort devoted to setting up the metadata search service.

3.1.3 Schema-Free Metadata Index Model. As mentioned in Section 2.2, in the DBMS-powered solutions, the indexing and querying procedures always require information about the data schema. In MIQS, when building the index, users only have to specify the name of the attributes without knowing the host of the attributes. MIQS will match every attribute with the specified name, regardless of the hierarchical schema of the metadata. Also, for metadata search, users only need to be aware of the type of query target, the query attribute name q_k and the query attribute value q_v in the query condition. Users do not need to know the specific structures of the metadata. There is also no need to support a full-fledged SQL-like query language.

3.1.4 Portability and Mobility. As discussed in 3.1.2, MIQS stores the index files alongside the self-describing files. Consequently, whenever the combined self-describing files and index files arrive at a scientific computing facility, and the metadata indexing and querying service can be immediately put to use once the dataset is prepared and the application is deployed. This practice conforms to the self-contained data management paradigm. Therefore, we consider the portability and mobility of the MIQS solution for self-describing files to be favorable when compared to DBMS-powered metadata search solutions.

3.2 Index Construction

In MIQS, there are two types of index construction procedures. The first is to build the index from scratch, which involves metadata scanning (optional), metadata indexing and index persistence. We call this procedure "initial indexing". The other is called "index reloading". As shown in Figure 3, for initial indexing, the MIQS index builder has to retrieve metadata attributes from the self-describing data files. This is done by scanning the metadata in an existing set of files or by receiving the attributes in an in-situ fashion when the data files are generated. When an attribute is encountered, the index builder retrieves the attribute name and value, along with the corresponding object and file path and creates an index in the memory. After the metadata scanning and indexing are finished, MIQS stores the in-memory index into the MIQS index files using its index persistence mechanism. For index reloading, the MIQS index builder can load all the MIQS index files to rebuild the in-memory index. Our compact index file format is designed to support rapid index recovery. Next, we will describe the initial indexing procedure in detail.

3.3 Initial Indexing

When MIQS index files do not exist yet, we perform initial indexing procedure. The initial indexing procedure includes three sub-procedures, which are metadata scanning (optional), metadata indexing, and index persistence.

3.3.1 Metadata Scanning. When building metadata index on an existing set of data files for the first time, MIQS will perform metadata scanning procedure. Since MIQS is designed for applications running in a parallel environment on HPC systems, MIQS leverages parallelism starting from the index construction phase. Consider a parallel application with n processes where process rank r ranges from 0 to $n-1$. In this scenario, each process maintains a file counter, given as $file\_counter$, and that is incremented each time a file is encountered during metadata scanning. The following equation can be used to determine whether or not a given file should be scanned:

$$p = file\_counter\%n \tag{1}$$

When $p = r$, process r will perform metadata scanning on the encountered file; otherwise, it will skip this file and continue. By utilizing this mechanism, MIQS is able to leverage data parallelism across different data files and to avoid the I/O contention occurred when multiple processes are accessing the same data file concurrently.

During the metadata scanning procedure, the MIQS index builder scans four elements from a self-describing file collection (as shown in Figure 5). These elements include :

- The directory that contains the self-describing files. We call this directory the root directory of the data file collection (we use root directory for this meaning in the rest of this paper).
- The self-describing data files.
- The objects in the self-describing files, including groups and datasets.
- The metadata attributes attached to each data object in the self-describing data file.

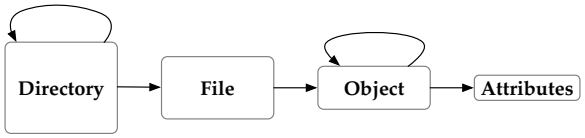


Figure 5: Four elements in MIQS metadata scanning. The arrow shows inclusive relationship.

For scanning metadata, MIQS defines a series of operations on the aforementioned four elements. All these procedures take a global memory pointer called *index_anchor* as a parameter. This memory pointer references the address of the MIQS index root. As part of each procedure, MIQS recursively scans the files of each sub-directory within the root directory.

For each file scanned, MIQS invokes the OnFile procedure. For every object within a file, the OnObject is called with the file path and the object path as parameters.

In the OnObject procedure, MIQS extracts the metadata attributes for the encountered data object and prepares the index record for each attribute. Since multiple attributes may share the same set of file and object paths, if MIQS stores the path strings alongside each attribute key-value pair, these path strings will be repeated among index records of different attributes for many times. The repetition of path strings will lead to significant memory overhead. Thus, for each attribute to be indexed, MIQS takes the related file and object paths, puts them into the file path list and object path list, respectively, and records their integer identifiers. Afterwards, both identifiers are combined into a tuple $t = \{file\_path\_identifier, object\_path\_identifier\}$. This design replaces space-consuming path strings that are of variable lengths with fixed-size and space-saving integers. Based on the key and value of the attributes, MIQS is then able to build an index for the attribute with tuple t by calling the CreateIndex procedure. When finished, MIQS recursively scans and invokes the OnObject procedure on any existing child objects.

3.3.2 Metadata Indexing. Given the metadata attribute name *key* and attribute value *value*, along with the tuple t storing the path identifiers of the file and the data object which the metadata attribute comes from, MIQS performs the metadata indexing procedure. As shown in Algorithm 1, in the CreateIndex procedure, MIQS first extracts the root ART from the index anchor (which is a global pointer to the in-memory index) and then inserts the attribute name into it. By doing so, MIQS is able to locate the corresponding leaf node of the attribute name at the root-level adaptive radix tree (a.k.a ART) where a self-balancing search tree (a.k.a SBST) or an ART places its root. If it is a numeric value, MIQS inserts the value into the SBST and inserts the tuple t into the list which is linked to the corresponding leaf node. Likewise, if it is a string, MIQS inserts it into the ART and links the tuple with the corresponding leaf node of the ART.

After the metadata indexing procedure, each process of the parallel application maintains partial indexes for approximately $1/n$ of the metadata in the entire self-describing file collection. The index persistence mechanism ensure that each process gets a full copy of the index for the entire file collection.

Algorithm 1 Create Index

```
1: procedure CreateIndex(key, value, t, index_anchor)
2:     ART_root ← index_anchor
3:     leaf_k ← insert_into_ART(key, ART_root)
4:     if value is a string then
5:         ART_value ← get_ART(leaf_k).
6:         leaf_v ← insert_into_ART(value, ART_value).
7:         link(leaf_v, t).
8:     else if value is a number then
9:         SBST_value ← get_SBST(leaf_k).
10:        leaf_v ← insert_into_SBST(value, SBST_value)
11:        link(leaf_v, t).
12:    end if
13: end procedure
```

3.3.3 Index Persistence. Once the MIQS index builder finishes metadata indexing for the metadata in all data files, the MIQS index persistence mechanism is invoked and the in-memory metadata index is stored onto disk.

Figure 6 shows the layout of MIQS compact index file format and how the in-memory index is stored in such a format. When

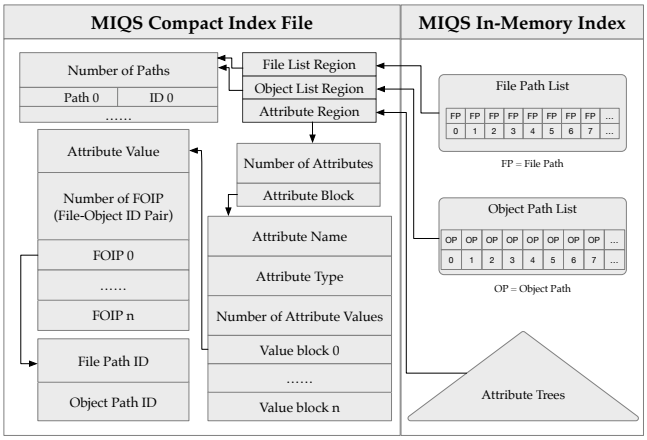


Figure 6: MIQS compact index file format

persisting the in-memory index, MIQS first writes the file path list and object path list in succession. Each path list is persisted as a frame that starts with a frame header. The frame header contains an integer number that represents the number of items in the path list. After the frame header, the items in the path list are stored as a series of item blocks. Each block contains the path string and also the numeric identifier.

After file list frame and object list frame, there is the attribute frame. The attribute frame also starts with a frame header containing the total number of attributes that are maintained in the in-memory index. After the frame header, there are a series of attribute blocks.

The attribute block starts with the attribute header containing the attribute name, the data type of the attribute and the number of unique attribute values. We differentiate three data types supported in MIQS: an integer value type (including all sizes of integers), a floating-point value type, and a string type. Since each attribute may have different attribute values of the same data type, MIQS records the number of unique attribute values at the end of the attribute header and then write a series of attribute value blocks (or value blocks for short) after it.

Each value block starts with the actual attribute value, the number of matched File-Object ID pairs (FOIPs) - the tuples created in OnObject procedure, and then the actual FOIP containing the file path identifier and the object path identifier.

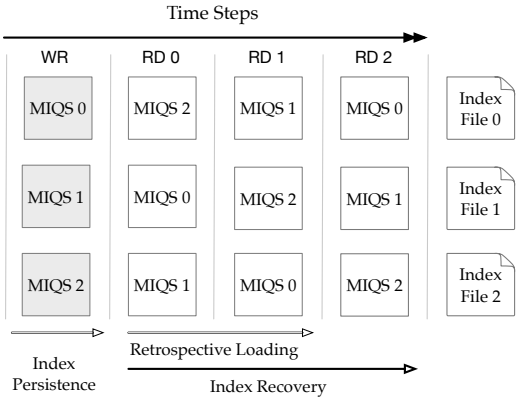


Figure 7: MIQS parallel index file access. "WR" stands for write operation, "RD" stands for read operation.

Following the above procedure, each process will write its own in-memory index into a separate index file with the process rank in the index file name (shown as "WR" step in Figure 7). Thus far, the index persistence procedure is finished.

3.4 Index Loading

3.4.1 Retrospective Loading. After the index persistence procedure, each process only maintains the index of the files it has scanned. In order to have a full copy of the index of the entire file collection, MIQS performs "**retrospective loading**" procedure. For an application with n processes, after these processes finish their own index persistence procedure, each process $r (0 \leq r < n)$ will take $n-1$ "RD" steps (as shown in Figure 7) to read the $n-1$ index files generated by other processes. At each "RD" step $s (0 \leq s < n-1)$, process r reads index file $f = (n + r - s - 1)\%n$ and loads the index record into its own in-memory index. After s "RD" steps, each process will have a full copy of the entire metadata index. As shown in Figure 7, at RD step 0 and 1, process 0 reads index file 2 and 1 respectively, process 1 reads index file 0 and 2 respectively, and process 2 reads index file 1 and 0 respectively. In this way, we can guarantee that no index files will be accessed by multiple processes simultaneously. After the retrospective loading procedure, each process will have a full copy of the metadata index on the entire self-describing file collection.

3.4.2 Index Reloading. MIQS stores the metadata index in the MIQS index file. Therefore, when applications need to be restarted or deployed at another HPC facility, MIQS can simply reload the index files and get ready for query processing. By following the data

layout in the MIQS index file format, the MIQS index builder reads each attribute frame, retrieves the corresponding index records and loads them into the in-memory index. Utilizing the same index file loading procedure as in retrospective loading, MIQS performs one more RD step to load a full copy of the metadata index for each process when performing index reloading. For example, as shown in Figure 7, three processes have stored their in-memory index into three different index files at the WR step when writing in-memory index to index files. After shifting the index file read in the consequent two RD steps by following the same procedure as in retrospective loading, process 0 loads file 0 at RD step 2, process 1 loads file 1 and process 2 loads file 2. Once completed, all three processes have the same full copy of the metadata in-memory index.

3.5 Serving Queries

In order to process metadata queries, MIQS follows the procedure described in Algorithm 2. MIQS first searches the root level of the ART to locate leaf node $leaf_k$ which has a matching attribute name. Then, based on the data type of the attribute value, MIQS starts from leaf node $leaf_k$ and searches through either the ART or the SBST associated with $leaf_k$. When searching the appropriate tree for attribute values, the relationship R is passed to the search function in order to find appropriate leaf nodes which satisfy the specified relationship between itself and the given attribute value. Once the leaf nodes are found, the lists of tuples containing the path identifiers to the owner files and the owner object can be retrieved. After looking up the file path list and the object path list with the identifier of each file path or object path, the actual file paths and object paths are retrieved and returned as the search result.

Algorithm 2 Serving Query

```
1:  procedure ServingQuery(key, R, value, index_anchor)
2:      ART_root ← index_anchor
3:      leaf_k ← search_ART(key, ART_root)
4:      if value is a string then
5:          ART_value ← get_ART(leaf_k).
6:          leaf_v ← search_ART(value, R, ART_value).
7:          result ← get_tuple_list(leaf_v).
8:      else if value is a number then
9:          SBST_value ← get_SBST(leaf_k).
10:         leaf_v ← search_SBST(value, R, SBST_value)
11:         result ← get_tuple_list(leaf_v).
12:     end if
13:     return result
14: end procedure
```

Each process with MIQS can perform metadata searching, and, as a result of the shared-nothing architecture of MIQS, no communication is needed among these processes. Consequently, the throughput of the search performance is expected to be proportional to the number of processes, which makes the metadata search process highly scalable. Also, as there is no communication with other data sources like DBMS, the search latency is expected to be small.

4 EVALUATION

MIQS is designed to provide an efficient metadata search service that complies with the self-contained data management paradigm. In this section, we show how MIQS is suitable for the self-describing and self-contained data management solution by reporting the evaluation results of index construction performance, query performance, and storage overhead. In our evaluation, we compare MIQS with a MongoDB-powered metadata search solution which we consider to be a typical example of DBMS-based solution as the document-based data model of the MongoDB is more flexible than RDBMS-based solutions and hence is better suited for both hierarchical and flat metadata organization. As such, we do not compare against RDBMS solutions in this paper. Also, for fair comparison, we evaluate both solutions by building metadata index while scanning a set of HDF5 files, since we did not find any in-situ index construction use case through our investigation study on MongoDB solutions.

4.1 Experimental Setup

4.1.1 Platform. We conducted our evaluation of MIQS on the Edison supercomputer hosted at the National Energy Research Scientific Computing Center (NERSC). This system consists of 5586 "Ivy Bridge" compute nodes, where each node features two 12-core Intel® "Ivy Bridge" processors at 2.4GHz and 64 GB of DDR3 1866 memory. The Edison employs a Cray Aries interconnect with Dragonfly topology and 23.7 TB/s global bandwidth. The compute nodes use GPFS for its home directory and multiple Lustre file systems as scratch spaces. We used a 30 PB Lustre file system with over 700 GB/s peak I/O bandwidth for our evaluation.

For the comparison experiment against MongoDB, we used the MongoDB instance that is also maintained by NERSC. It is installed on a separate machine that can be accessed from the Edison compute nodes through an Infiniband network with 56 Gb/s bandwidth. The host machine of MongoDB has two 16-core Intel® Xeon™ processors E5-2698 v3 ("Haswell") at 2.3 GHz and 128 GB of DDR4 2133 MHz memory. It features a 6 TB 7200rpm HDD with 6 Gb/s SAS interface. The MongoDB was initialized with the default configuration, which enables data compression on top of the WiredTiger storage engine [44].

4.1.2 Dataset. Our dataset contains a set of 100 real-world HDF5 files coming from the Baryon Oscillation Spectroscopic Survey (BOSS) [5]. The total data size of these 100 HDF5 files on disk is approximately 145GB with 144 million attribute key-value pairs attached to 1.5 million data objects (roughly 96 attribute key-value pairs on each object). Figure 8 summarizes the statistics of this dataset. The sizes of the files range from 400MB to 2.2GB (shown in Figure 8a). The number of objects per file spans between 4,800 to 23,000 (shown in Figure 8b). With over 250 different attributes attached to different data objects in each file, the number of attribute key-value pairs per file ranges from 300,000 to 2,300,000 (Figure 8c).

4.1.3 Evaluation Procedure. In our evaluation, we built two test drivers - the MongoDB test driver and the MIQS test driver. Both test drivers are MPI programs implemented with metadata indexing and querying functions. For the evaluation on MongoDB and MIQS,

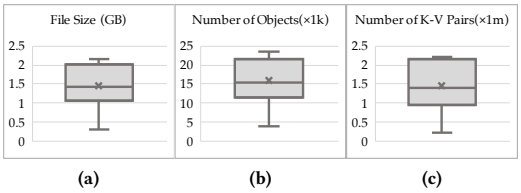


Figure 8: Statistics of datasets

we performed 5 test rounds and for each test round $r = \{1, 2, 3, 4, 5\}$ we ran $20 \times r$ processes for each test driver and we bound each process to a single core of the CPU and a different HDF5 file. As there are two 12-core CPUs on each compute node, each test round r uses r compute nodes for running $20 \times r$ processes.

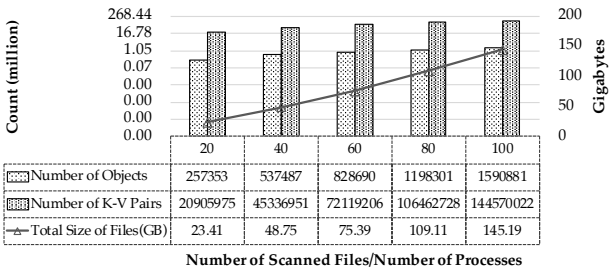


Figure 9: Dataset size at each test round

Figure 9 shows the size of the dataset being scanned at each test round, including the number of scanned objects and the number of scanned attribute key-value pairs in log-scale (the left log-scale axis with base 2), and the total size of the indexed files in gigabytes (the right axis).

For the evaluation on MongoDB, due to the schema constraints discussed in Section 2.2, it is impossible to know the paths to the owner objects of all the attributes without scanning the entire dataset. Therefore, we selected 16 representative metadata attributes with known owner object paths for creating attribute indexes. As shown in the first and second column of Table 2, these 16 attributes include 5 most frequently occurring integer attributes, 5 most frequently occurring float attributes, and 6 most frequently occurring string attributes. We consider the number and the diversity of these attributes to be sufficient for simulating real-world metadata search scenarios.

In order to build the metadata index on MongoDB at each test round, we first sent index creation commands to MongoDB to create indexes on these 16 attributes. Each test driver process then scanned a single HDF5 file and transformed the HDF5 metadata into several BSON documents that are smaller than 16MB [1]. These documents were then inserted into MongoDB. We believe, based on our study of MongoDB-based solutions, that this practice is necessary and in accordance with the realistic MongoDB-based use cases.

We evaluated MIQS for 10 test rounds. In the first 5 rounds, we ran MIQS test drivers with the settings shown in Figure 9, and created indexes for the 16 attributes we selected in Table 2. For the second set of five test rounds, we used the same settings but we let

Table 2: Attributes and sample query conditions

Data Type	Attributes	Sample Query Conditions
Integer	BESTEXP	BESTEXP=103179
	DARKTIME	DARKTIME=0
	BADPIXEL	BADPIXEL= 155701
	COLLB	COLLB=26660
	HIGHREJ	HIGHREJ=8
Floating-point	HELIO_RV	HELIO_RV=26.6203
	IOFFSTD	IOFFSTD=0.0133138
	CRVAL1	CRVAL1=3.5528
	M1PISTON	M1PISTON=661.53
	FBADPIX2	FBADPIX2=0.231077
String	AUTHOR	AUTHOR="Scott Burles & David Schlegel"
	FILENAME	FILENAME="badpixels-56149-b1.fits.gz"
	EXPOSURE	EXPOSURE="sdR-b2-00154990.fit"
	LAMPLIST	LAMPLIST="lamphgcdne.dat"
	COMMENT	COMMENT="sp2blue cards follow"
	DAQVER	DAQVER="1.2.7"

the test driver index every attribute it encountered while scanning the HDF5 files.

Throughout our evaluation, we refer to the index construction procedure as "indexing", a file or attribute that has been scanned as "scanned file" or "scanned attribute", and a file or attribute that has been successfully indexed as "indexed file" or "indexed attribute".

After indexing, both test drivers performed 1,024 metadata queries from each process against their target data sources - MongoDB and MIQS indexes, respectively. Each query asked for identifiers of objects that match with the given query condition. Our query conditions covered the 16 selected attributes with non-repetitive attribute values of three different data types - integer, floating-point, and string. Thus, these 1,024 metadata queries covered 64 different values for each selected attribute. The last column in Table 2 shows selected sample query conditions for each data type from the 1,024 queries we issued.

4.2 Index Construction Time

Index construction time is very important to a metadata search service for large-scale self-describing datasets. It determines the time for getting metadata index ready to use and hence has a significant impact on the entire software life cycle when the metadata search service is integrated.

Figure 10 shows the index construction time for MongoDB and MIQS during the 5 test rounds. For MongoDB, the total indexing time needed to create indexes for the 16 specified attributes ranged from 373 to 587 seconds. In comparison, when indexes were constructed on the same set of 16 attributes in MIQS, the indexing time ranged from 194 seconds to 310 seconds, which is approximately half of the indexing time of MongoDB. We also tested indexing all attributes in MIQS and the indexing time ranged from 533 seconds to 860 seconds. Finally, we tested building the in-memory index from the MIQS index files. The time required for loading 16-attribute index files varied from 3 seconds to 27 seconds, approximately a 95%-99% time reduction compared to MongoDB-powered solution. Moreover, MIQS was able to load the index files for all attributes in

10.3 - 112 seconds, which is equivalent to only 2% - 19% of the time taken by the MongoDB-powered solution.

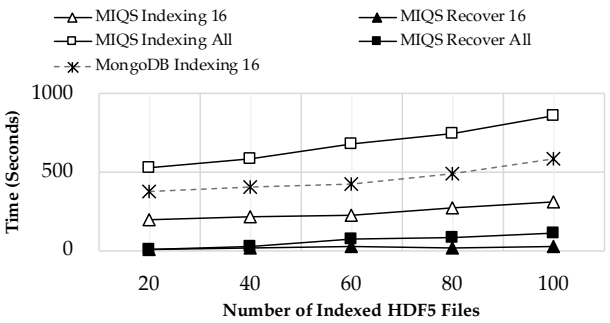


Figure 10: Time for index construction

For a better understanding on this result, we further decompose the indexing time. As shown in Figure 11, the time spent scanning HDF5 files was roughly the same, approximately 200 seconds, for both the MongoDB solution and MIQS, regardless of the number of indexed attributes (Figure 11a, 11b, and 11c). However, the MongoDB test driver also spent 200 to 359 seconds for inserting the metadata BSON [7] documents into MongoDB via network. The total time spent inserting BSON documents is 1.5 - 2× the time spent scanning HDF5 files and is significant given only 16 attributes were indexed (Figure 11a). In fact, this time may include time spent transferring BSON documents over the network and performing I/O operations at the MongoDB server.

The time for inserting BSON documents was even 1.5 to 2 times of the time spent for scanning the HDF5 data files, and this may include the time for transferring the BSON document over the network and also the I/O operation occurred at MongoDB server. Apart from the data file scanning procedure, building the actual in-memory indexes for 16 attributes only took 24-32 seconds for MIQS, while index persistence and index retrospective loading only took 3 to 28 seconds (Figure 11b). For indexing all attributes, MIQS spent 287 to 470 seconds building the in-memory indexes and the index persistence and retrospective loading only took 21 to 114 seconds (Figure 11c). This is quite comparable to the metadata indexing time in MongoDB; however, MIQS built indexes for all attributes while only 16 attributes were indexed in MongoDB.

Figure 11d illustrates the case where self-describing data files are transferred to a new site and the in-memory index has to be reloaded from the index files in MIQS. It took less than 40 seconds to load metadata indexes on 16 attributes utilizing up to 100 processes to initialize the metadata search service. The metadata search service can be initialized within 2 minutes when loading the metadata indexes of all attributes.

Overall, our evaluation on index construction time indicates that MIQS requires much less time than the MongoDB-powered solutions. It can also be rapidly deployed in the case where metadata index files already exist. Consequently, applications with MIQS can begin to perform metadata query much earlier than with MongoDB.

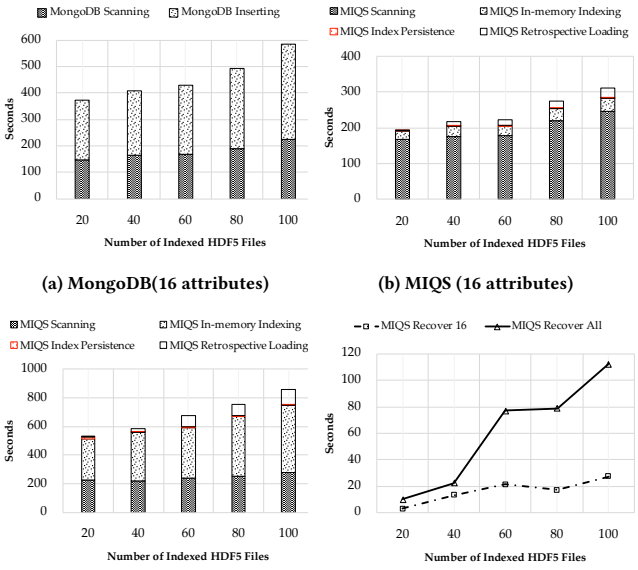
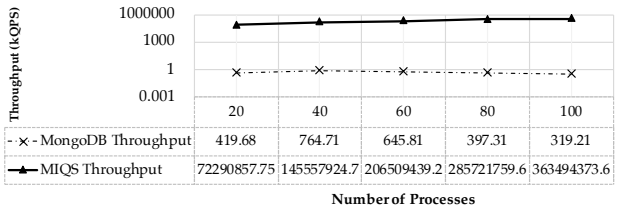


Figure 11: Breakdown analysis of indexing time

4.3 Query Performance

Query performance is critical to a metadata search service. By utilizing an in-memory shared-nothing parallel architecture, MIQS delivers an efficient metadata search service.



	20	40	60	80	100
MongoDB Throughput	419.68	764.71	645.81	397.31	319.21
MIQS Throughput	72290857.75	145557924.7	206509439.2	285721759.6	363494373.6

Figure 12: Query throughput comparison (kQPS)

As reported in Figure 12, for an application with 20 processes, the overall query throughput of MIQS was 72.2 billion queries per second while the MongoDB only achieved 419.7k queries per second. This is a drastic performance improvement of over 172,000× speedup. As shown in the figure, the query throughput of MIQS scales almost linearly with the number of processes, and ranges from 72.2 billion QPS all the way up to 363.4 billion QPS. However, the MongoDB query throughput only slightly increased to 764.7 kQPS when queries were issued by 40 processes and dropped from 645.8 kQPS to 319.2 kQPS when the number of processes increased from 60 to 100.

The query latency is depicted in Figure 13. Although the latency of each query in MongoDB remains steady around 0.28 to 0.29 milliseconds, as the number of processes increases from 20 to 100, the query latency dramatically increases from 47 seconds all the way up to 313 seconds. This is almost equal to the time

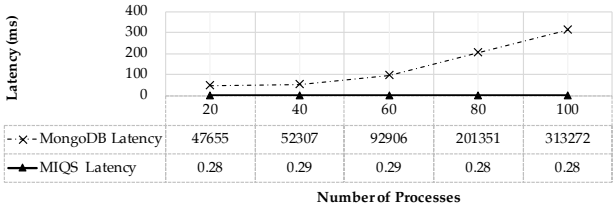


Figure 13: Query latency comparison(ms)

spent scanning 100 HDF5 data files during our index construction experiment. These results confirm that, as a benefit of following the self-contained data management paradigm, MIQS outperforms MongoDB-powered solutions and delivers superior query performance for metadata searches.

4.4 Storage Consumption

MIQS is designed as an in-memory metadata search service wherein processes have their own full copy of the metadata index. Thus, the memory footprint has to be small. In addition, the storage consumption must be minimized so that the index files can be integrated into existing self-describing file collections without significantly increasing the size of the dataset.

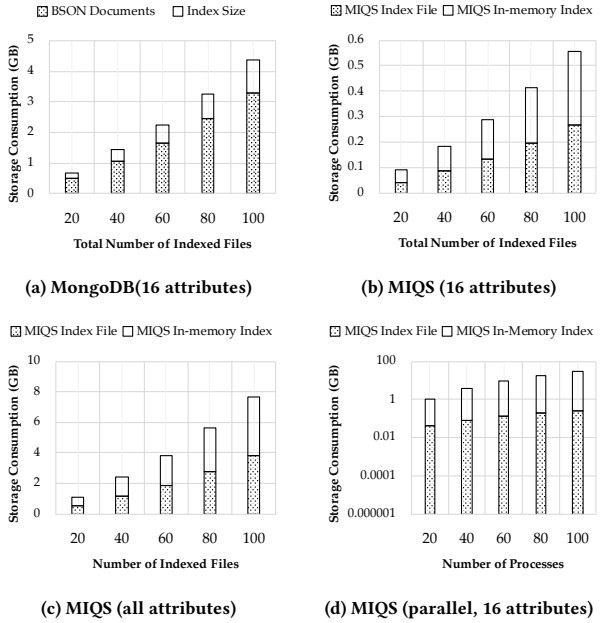
In our evaluation, we collected MongoDB storage consumption statistics using the administrative command "`dbStats`". The result of this command includes the "`indexSize`" field which represents the total size of the indexes and also the "`dataSize`" field which represents the overall storage consumption of MongoDB. The storage overhead for maintaining BSON documents in MongoDB is then given by the difference between "`dataSize`" and "`indexSize`". For the memory consumption of MIQS, we track every memory allocation required by the indexing data structures. For storage consumption, we measure and analyze the size of MIQS index files.

Figure 14 shows the storage consumption comparison between MongoDB and MIQS. As shown in Figure 14a, when the index is built against only 16 attributes, the total storage consumption of MongoDB ranges from about 680 MB to approximately 4.2 GB. Regardless of scale, the index size in MongoDB occupies only about 20% of the overall storage space consumption while 80% is used to store metadata BSON documents.

When it comes to MIQS, as depicted in Figure 14b, the total storage consumption for indexing 16 attributes does not exceed 0.6 GB at any scale. The index files consume roughly the same amount of space on disk as the in-memory index consumes in the memory. For our largest test, MIQS reduces the total storage consumption by about 75.4% when compared to MongoDB.

Indexing all attributes in MIQS yields a total storage consumption that ranges from 1.5GB to 7.8 GB where the in-memory index consumes half of the total storage space (Figure 14c). Although the total storage consumption of MIQS is twice that of MongoDB, MIQS indexes all attributes while MongoDB indexes only 16. Further, MongoDB wastes storage space storing duplicated metadata that already exists in the self-describing data files.

Figure 14d shows the overall storage consumption in parallel scenarios where the number of processes increases from 20 to 100.



(a) MongoDB(16 attributes) **(b) MIQS (16 attributes)**

(c) MIQS (all attributes) **(d) MIQS (parallel, 16 attributes)**

Figure 14: Breakdown analysis on memory consumption

As illustrated in the figure, parallel applications with MIQS consume up to approximately 30 GB of storage space when indexing 16 attributes. However, considering that each process only needs to maintain around 600 MB of memory for metadata search service, which provides the high query throughput we have reported in the previous section, we believe the storage consumed by the index is worthwhile. The low storage consumption of MIQS further reinforces the idea that the design of MIQS complies with self-contained data management paradigm.

4.5 Summary of Evaluation

MIQS is designed as a metadata indexing and querying service that follows the self-contained data management paradigm. From our evaluation, we can discern three significant benefits that result from adherence to this paradigm: schema-free metadata indexing (MIQS was capable of indexing and querying all attributes but MongoDB was not), rapid index construction (hundreds of seconds saved in MIQS as compared to MongoDB) and superior search performance (172k× throughput improvement compared to MongoDB-powered solution). Our evaluation also shows that MIQS takes only a small amount of memory and that the small size of its index files enhances portability. Our results demonstrate the promise of MIQS and present a rationale for integration with existing self-describing and self-contained data management solutions.

5 CONCLUSION & FUTURE WORK

Self-describing data formats store metadata alongside the data objects themselves. Existing metadata search services for these self-describing data formats are primarily built upon external database management systems (DBMS). Unfortunately, these solutions disregard the principle of self-describing and self-contained data

management paradigm, hinder schema-transparency and portability and hence introduce additional complexity in deployment and maintenance.

In this paper, we have proposed the Metadata Indexing and Querying Service (MIQS) for efficient metadata search on self-describing file formats such as HDF5. MIQS provides portable and schema-free indexing and querying solution that complies with the self-contained data management paradigm. It eliminates the necessity of maintaining a disjoint data source that duplicates metadata and thereby requires a smaller overall storage footprint, as compared to the DBMS-powered solutions. We have conducted extensive evaluations to compare MIQS against a state-of-the-art DBMS-based metadata querying solution - the MongoDB-powered metadata querying service. MIQS significantly outperforms MongoDB-powered solution in both index construction and metadata search. More importantly, our evaluation shows that MIQS complies with the self-contained data management principle of self-describing data formats.

At the time of writing this paper, we are in contact with the HDF5 developers to make MIQS available as an indexing and querying component for the HDF5 library. In the future, we will enhance MIQS to support complex queries on compound data types in addition to numeric values or strings.

ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grant CNS-1338078, CNS-1362134, CCF-1409946, CCF-1718336, OAC-1835892, and CNS-1817094. This work is supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. (Project: EOD-HDF5: Experimental and Observational Data enhancements to HDF5, Program managers: Dr. Laura Biven and Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

REFERENCES

[1] 2018. MongoDB Limits and Thresholds. https://docs.mongodb.com/manual/reference/limits/.
[2] 2018. Simple and fast C library implementing a thread-safe API to manage hash-tables, linked lists, lock-free ring buffers and queues. https://github.com/xant/libhl.
[3] MG Aartsen, K Abraham, M Ackermann, J Adams, JA Aguilar, M Ahlers, M Ahrens, D Altmann, K Andeen, T Anderson, et al. 2016. Search for sources of High-Energy neutrons with four years of data from the Icetop Detector. *The Astrophysical Journal* 830, 2 (2016), 129.
[4] MG Aartsen, M Ackermann, J Adams, JA Aguilar, Markus Ahlers, M Ahrens, I Al Samarai, D Altmann, K Andeen, T Anderson, et al. 2017. Constraints on galactic neutrino emission with seven years of IceCube data. *The Astrophysical Journal* 849, 1 (2017), 67.
[5] Christopher P Ahn, Rachael Alexandroff, Carlos Allende Prieto, Scott F Anderson, Timothy Anderton, Brett H Andrews, Éric Aubourg, Stephen Bailey, Eduardo Balbinot, Rory Barnes, et al. 2012. The ninth data release of the Sloan Digital Sky Survey: first spectroscopic data from the SDSS-III Baryon Oscillation Spectroscopic Survey. *The Astrophysical Journal Supplement Series* 203, 2 (2012), 21.
[6] Shadab Alam, Metin Ata, Stephen Bailey, Florian Beutler, Dmitry Bizyaev, Jonathan A Blazek, Adam S Bolton, Joel R Brownstein, Angela Burden, Chia-Hsun Chuang, et al. 2017. The clustering of galaxies in the completed SDSS-III Baryon Oscillation Spectroscopic Survey: cosmological analysis of the DR12 galaxy sample. *Monthly Notices of the Royal Astronomical Society* 470, 3 (2017), 2617–2652.
[7] bsonspec.org. 2018. BinÂŋary JSON Specification. http://bsonspec.org/spec.html.
[8] Chi Chen, Zhi Deng, Richard Tran, Hanmei Tang, Iek-Heng Chu, and Shyue Ping Ong. 2017. Accurate force field for molybdenum by machine learning large materials data. *Physical Review Materials* 1, 4 (2017), 043603.
[9] Tull Craig E., Essiari Abdelilah, Gunter Dan, et al. 2013. The SPOT Suite project. http://spot.nersc.gov/.
[10] Digital Curation Conference (DCC). 2018. Scientific Metadata. http://www.dcc.ac.uk/resources/curation-reference-manual/chapters-production/scientific-metadata.
[11] Jeffrey J Donatelli, James A Sethian, and Peter H Zwart. 2017. Reconstruction from limited single-particle diffraction data via simultaneous determination of state, orientation, intensity, and phase. *Proceedings of the National Academy of Sciences* 114, 28 (2017), 7222–7227.
[12] Bin Dong, Surendra Byna, and Kesheng Wu. 2015. Spatially clustered join on heterogeneous scientific data sets. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 371–380.
[13] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of supercomputing*, Vol. 99. 5–33.
[14] P Greenfield, M Droettboom, and E Bray. 2015. ASDF: A new data format for astronomy. *Astronomy and Computing* 12 (2015), 240–251.
[15] The HDF Group. 2018. HDF5 Topic Parallel Indexing Branch. https://git.hdfgroup.org/users/jsoumagne/repos/hdf5/browse?at=refs%2Fheads%2Ftopic-parallel-indexing.
[16] The HDF Group. 2018. HDF5 Users. https://support.hdfgroup.org/HDF5/users5.html.
[17] Kohei Hiraga, Osamu Tatebe, and Hideyuki Kawashima. 2018. PPMDS: A Distributed Metadata Server Based on Nonblocking Transactions. In *Fifth International Conference on Social Networks Analysis, Management and Security, SNAMS 2018, Valencia, Spain, October 15-18, 2018*. 202–208. https://doi.org/10.1109/SNAMS.2018.8554478
[18] Joint Genome Institute. 2013. The JGI Archive and Metadata Organizer(JAMO). http://cs.lbl.gov/news-media/news/2013/new-metadata-organizer-streamlines-jgi-data-management.
[19] json.org. 2018. Introducing JSON. https://www.json.org.
[20] Daniel Korenblum, Daniel Rubin, Sandy Napel, Cesar Rodriguez, and Chris Beaulieu. 2011. Managing biomedical image metadata for search and retrieval of similar images. *Journal of digital imaging* 24, 4 (2011), 739–748.
[21] Margaret Lawson and Jay Lofstead. 2018. Using a Robust Metadata Management System to Accelerate Scientific Discovery at Extreme Scales. In *Proceedings of the 2nd PDSW-DISCS '18*. https://doi.org/10.1109/PDSW-DISCS.2018.00005
[22] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812
[23] Andrew W Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L Miller. 2009. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems.. In *FAST*, Vol. 9. 153–166.
[24] Jialin Liu, Debbie Bard, Quincey Koziol, Stephen Bailey, et al. 2017. Searching for millions of objects in the BOSS spectroscopic survey data with H5Boss. In *2017 New York Scientific Data Summit (NYSDS)*. 1–9. https://doi.org/10.1109/NYSDS.2017.8085044
[25] Yaning Liu, George Shu Heng Pau, and Stefan Finsterle. 2017. Implicit sampling combined with reduced order modeling for the inversion of vadose zone hydrological data. *Computers & Geosciences* (2017).
[26] Jay F. Lofstead, Scott Klasky, et al. 2008. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE*. 15–24.
[27] Arun Mannodi-Kanakkithodi, Tran Doan Huan, and Rampi Ramprasad. 2017. Mining materials design rules from data: The example of polymer dielectrics. *Chemistry of Materials* 29, 21 (2017), 9001–9010.
[28] Vilobh Meshram, Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar, Ravi Prakash, and Dhabaleswar K. Panda. 2011. Can a Decentralized Metadata Service Layer Benefit Parallel Filesystems?. In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. 484–493. https://doi.org/10.1109/CLUSTER.2011.85
[29] MongoDB. 2018. MongoDB. https://www.mongodb.com.
[30] mongodb.com. 2018. The MongoDB 4.0 Manual. https://docs.mongodb.com/manual/.
[31] David Paez-Espino, I Chen, A Min, Krishna Palaniappan, Anna Ratner, Ken Chu, Ernest Szeto, Manoj Pillay, Jinghua Huang, Victor M Markowitz, et al. 2017. IMG/VR: a database of cultured and uncultured DNA Viruses and retroviruses. *Nucleic acids research* 45, D1 (2017), D457–D465.
[32] PostgreSQL. 2018. PostgreSQL. https://www.postgresql.org.
[33] Russ Rew and Glenn Davis. 1990. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications* 10, 4 (1990), 76–82.
[34] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *FAST*, Vol. 2.

[35] 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.

[36] Self-balancing binary search tree. 2019. Self-balancing binary search tree — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree [Online; accessed 10-April-2019].

[37] Arie Shoshani and Doron Rotem. 2009. *Scientific data management: challenges, technology, and deployment.* Chapman and Hall/CRC.

[38] Hyogi Sim, Youngjae Kim, Sudharshan S. Vazhkudai, Geoffroy R. Vallée, Seung-Hwan Lim, and Ali Raza 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, SC 2017, Denver, CO, USA, November 12 - 17, 2017*. 5:1–5:12. https://doi.org/10.1145/3126908.3126929

[39] sqlite.org. 2017. SQLite. https://sqlite.org.

[40] 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.

[41] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, Jiri Schindler and Erez Zadok (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/fast15/technical-sessions/presentation/thomson

[42] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. 2017. MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 1174–1183. https://doi.org/10.1109/IPDPS.2017.39

[43] Zeyi Wen, Xingyang Liu, Hongjian Cao, and Bingsheng He. 2018. RTSI: An Index Structure for Multi-Modal Real-Time Search on Live Audio Streaming Services. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 1495–1506. https://doi.org/10.1109/ICDE.2018.00168

[44] WiredTiger. 2018. WiredTiger. http://www.wiredtiger.com/.

[45] Quanqing Xu, Rajesh Vellore Arumugam, Khai Leong Yang, and Sridhar Mahadevan. 2013. Drop: Facilitating distributed metadata management in eb-scale storage systems. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–10.

[46] Wei Zhang, Houjun Tang, Suren Byna, and Yong Chen. 2018. DART: Distributed Adaptive Radix Tree for Efficient Affix-based Keyword Search on HPC Systems. In *Proceedings of The 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. https://doi.org/10.1145/3243176.3243207

[47] Dongfang Zhao, Kan Qiao, Zhou Zhou, Tonglin Li, Zhihan Lu, and Xiaohua Xu. 2017. Toward Efficient and Flexible Metadata Indexing of Big Data Systems. *IEEE Trans. Big Data* 3, 1 (2017), 107–117.

[48] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling embedded in-situ indexing with deltaFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. 3:1–3:15. http://dl.acm.org/citation.cfm?id=3291660