2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)

Exploring Metadata Search Essentials for Scientific Data Management

Wei Zhang[1], Suren Byna[2], Chenxu Niu[1], Yong Chen[1]
[1]Department of Computer Science, Texas Tech University
[2]Computational Research Division, Lawrence Berkeley National Laboratory
X-Spirit.zhang@ttu.edu, sbyna@lbl.gov, {Chenxu.Niu, yong.chen}@ttu.edu

***Abstract*—Scientific experiments and observations store massive amounts of data in various scientific file formats. Metadata, which describes the characteristics of the data, is commonly used to sift through massive datasets in order to locate data of interest to scientists. Several indexing data structures (such as hash tables, trie, self-balancing search trees, sparse array, etc.) have been developed as part of efforts to provide an efficient method for locating target data. However, efficient determination of an indexing data structure remains unclear in the context of scientific data management, due to the lack of investigation on metadata, metadata queries, and corresponding data structures. In this study, we perform a systematic study of the metadata search essentials in the context of scientific data management. We study a real-world astronomy observation dataset and explore the characteristics of the metadata in the dataset. We also study possible metadata queries based on the discovery of the metadata characteristics and evaluate different data structures for various types of metadata attributes. Our evaluation on real-world dataset suggests that trie is a suitable data structure when prefix/suffix query is required, otherwise hash table should be used. We conclude our study with a summary of our findings. These findings provide a guideline and offers insights in developing metadata indexing methodologies for scientific applications.**

***Index Terms*—Metadata Indexing, Metadata Search, Data Management, HDF5**

I. Introduction

Many scientific applications nowadays, including experiments, observations, and simulations, tend to store the data in self-describing data formats, such as HDF5 [1], netCDF [2], ROOT files [3] and FITS [4]. A few recently developed self-describing file formats including ADIOS-BP [5], Exdir [6], SDXF [7], and ASDF [8], are also being used. In these file formats, the metadata is stored alongside the data objects, and the applications working with these data formats are able to access the metadata and the dataset all at once. Such characteristic makes these formats both self-describing and self-contained.

As these scientific applications continuously generate enormous amount of data [9]–[16], the rapid data growth imposes significant challenges on scientific computing. One of the critical challenges is to locate the desired data among a massive number of data objects or data files. This process is usually accomplished by metadata search, which aims to find and collect data items (or their identifiers) wherein the metadata attributes match with given query conditions [17]–[21].

Traditionally, the database management systems (DBMS) can be used for serving such query. Several state-of-the-art examples can be seen from BIMM [22], EMPRESS [23], the SPOT Suite [24], and JAMO [25], where relational databases (e.g., SQLite [26], PostgreSQL [27]) and NoSQL databases (e.g., MongoDB [28]) are used for maintaining the metadata and providing metadata search functionality.

However, DBMS may not be the optimal solution of metadata search for self-describing data format due to the following reasons:

- For self-describing data formats, to use database systems to manage metadata and indexes, one needs to deploy and maintain a database system that should always be available. However, this requires substantial database maintenance effort, which is often not what scientists can offer with their domain knowledge or adds additional complexity to system administrators' daily jobs.
- To enable DBMS-powered metadata querying capability, the metadata must be imported into the DBMS system, which causes data duplication between the self-describing data files and also the database. Furthermore, such a solution introduces an additional need of synchronizing the metadata between self-describing data files and the database, which translates to additional latency and overhead.
- Last but not the least, self-describing data formats advocate the self-describing and self-contained data management, and the metadata is already contained in the data files along with the dataset. Employing an external database for metadata search purpose, in fact, violates and defeats the principle of self-describing and self-contained data management.

Instead of using external databases, it is highly efficient to build and manage metadata index via self-describing data libraries for applications to search for data. In our recent work [29], we have developed an in-memory index based on adaptive radix tree and self-balancing search tree, which obtained orders of magnitude faster than MongoDB in querying scientific metadata. However, there has been no system-

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00021

atic prior study in discovering which type of in-memory indexes are the most beneficial for searching metadata in self-describing file formats in different situations.

In order to investigate what might be the appropriate indexing data structures for metadata search under different circumstances, we first need to understand the critical aspects of metadata search problem. In this study, we approach this problem and develop in-depth understandings from three essential and orthogonal components of the metadata search problem, i.e., metadata, query, and index. We analyze them in detail and we call them "metadata essentials", "query essentials" and "index essentials". For metadata essentials, we provide the definition of scientific metadata, and we study the characteristics of the scientific metadata, including the volume and the distribution of metadata, as well as the data types of metadata attributes. Based on the essential data types of metadata attributes, we further discuss the query essentials, which refer to various essential types of scientific metadata queries. These queries are typically different from DBMS queries that were developed primarily for transaction processing. In order to better understand scientific metadata queries and the impact of different data structures on different queries, we classify metadata queries and investigate the impact of data structures on each type of metadata query. Together with the metadata essentials and the query essentials, we also discuss the index essentials. The index essentials are about the indexing data structures and their performance metrics. We conduct an empirical evaluation of indexing data structures using a real-world scientific data set to observe how data structures perform for different types of metadata queries.

The contributions of this study are as follows:

- We investigate and present understandings of scientific metadata essentials including data types and characteristics of scientific metadata.
- We introduce the essential types of metadata queries and provide a classification of these queries.
- We investigate the essential supporting data structures for different types of metadata queries and analyze how these data structures perform.
- We present an empirical study performed on a real-world scientific dataset to observe how various indexing data structures perform. Based on our study, we conclude our findings and suggest the desired data structures to use under different metadata search circumstances.

The rest of the paper is organized as follows. In Section II, we review the scientific metadata search essentials including metadata characteristics, common querying patterns, and index data structures. In Section III, we provide a detailed review and analysis on the indexing data structures and particularly the behavior and complexity of the supporting data structures. In Section IV, we present our empirical study performed on a real-world scientific data set from astronomy observations and then summarize our findings regarding indexing data structures. We conclude this study in Section V.

II. Metadata Search Essentials

A. Overview

In self-describing data formats, such as HDF5 [1], netCDF [2], ROOT [3], and FITS [4], metadata exists as a collection of attributes, where each attribute ($A = \{k, v\}$) contains an attribute name k and an attribute value v. Based on this representation, we formally define the metadata search problem we aim at in this study as follows:

> Given a collection of data files or data objects $\mathcal{C}$, the metadata search problem for scientific data formats is to find a collection of data files or data objects $\mathcal{C}_{sub}$, where a certain metadata attribute $A = \{k, v\}$ matches with the given query condition $\mathcal{Q} = \{k_q, R, v_q\}$, such that $\mathcal{C}_{sub} \subseteq \mathcal{C}$. k_q and v_q are the specified key and value in the query condition and R is the relationship between k_q and v_q.

TABLE I: Queries over String Attribute Values

Query Type	Sample Query
Exact Query	All objects with attribute "OBJFILE" = "sdR-b2-00115171.fit"
Prefix Query	All objects with attribute "OBJFILE" starting with "sdR-"
Suffix Query	All objects with attribute "OBJFILE" ending with ".fit"
Infix Query	All objects with attribute "OBJFILE" including "-b2-"

For better understanding on the definition of metadata search problem, we provide sample metadata queries in Table I and Table II.

TABLE II: Queries over Numeric Attribute Values

Query Type	Sample Query
Exact Query	All objects with attribute "DEREDSN2" is 8.3665
Range Query	All objects with attribute "DEREDSN2" between 0.2 and 8.4

In our recent study [29], to address the above metadata search problem, we have developed an approach of building in-memory indices directly on top of the metadata without using an external database. However, in order to better address a variety of scientific metadata queries, it is highly desired to understand the criteria for choosing appropriate indexing data structures. Based on our definition of the metadata search problem, we consider three orthogonal aspects of the problem - the metadata itself, different types of queries, and the indexes. We investigate the metadata search problem by categorizing these aspects into "Essentials" as shown in Figure 1 and discussed briefly below. We further discuss each aspect in the remaining sub-sections.

1) **Metadata Essentials** - We study the data types and the data characteristics of scientific metadata.
2) **Query Essentials** - The data type of metadata determines the types of queries over the metadata. Therefore, we consider different types of queries for different metadata.
3) **Index Essentials** - As there are different metadata queries and different data types, when determining an index data structure, we must consider the data type of the metadata, the query types a data structure can support, and the performance of the data structure with regard to

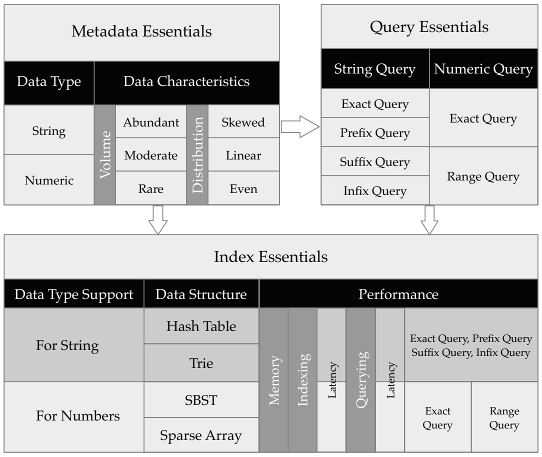

Figure 1: Three essential aspects of metadata search.

how the data characteristics of the metadata may affect the performance of the indexing data structure.

B. Metadata Essentials

Scientific metadata provides users with descriptive information about the underlying dataset and is therefore prevalently used for finding required data. In self-describing data formats, metadata can be seen as a collection of attributes. For each attribute $A = \{k, v\}$, where k denotes the attribute names that are typically strings, and the attribute value v can be of various data types. The data types of these metadata attributes have significant impact on possible types of metadata queries, as well as a choice of indexing data structures. According to the Digital Curation Conference (DCC) [30], the major data types of metadata attribute values are strings and numbers. Therefore, in this study, we focus on metadata queries where q_k is a string and q_v is a string or a number.

In addition to the data types of the metadata, another important aspect of the metadata is the data characteristics, including the volume and the distribution of the metadata. The volume of the metadata, such as the number of unique attribute values of a single attribute, plays an important role in determining indexing data structures. For example, when the number of indexed keys is high, it is prudent to choose the indexing data structure with lower time complexity as well as lower space complexity. However, when the number of indexed keys is average or small, the concern regarding space complexity tends to be less intense. Moreover, the data distribution of the metadata may also affect the performance of the indexing data structure. For example, while most data structures work well with evenly distributed data items or even linearly distributed data items, for highly skewed data items, the choice of indexing data structures will be based on how well the data structure can balance off the uneven distribution.

C. Query Essentials

Metadata queries supported are dependent on the data types of the metadata. As the major data types of attribute values are strings and numbers, we mainly focus on two categories of metadata queries - the queries on string attribute values and the ones on numeric attribute values.

1) String Attribute Values: For string attribute values v_s, the task of finding the data collection is usually done by finding a resultant exact or partial match on v_s. Thus, we summarize associated typical query types as shown below:

- **Exact Query**: To find a collection of data objects $\mathcal{C}$ where v_s of the specified attribute matches with a given string t exactly.
- **Prefix Query**: To find a collection of data objects $\mathcal{C}$ where the first some characters in v_s of the specified attribute match with a given string t.
- **Suffix Query**: To find a collection of data objects $\mathcal{C}$ where the last some characters in v_s of the specified attribute match with a given string t.
- **Infix Query**: To find a collection of data objects $\mathcal{C}$ where v_s of the specified attribute contains the given string t.

Table I lists sample queries on the string attribute values.

2) Queries on Numeric Attribute Values: For numeric attribute values v_n, they can be either integer numbers or floating-point numbers. For both integer and floating-point numbers, the queries can be of the following types:

- **Exact Query**: To find a collection of data objects $\mathcal{C}$ where the value v_n of the the specified attribute A is equal to the given number q_v from the query condition.
- **Range Query**: To find a collection of data objects $\mathcal{C}$ where the value v_n of the specified attributes A is within a given range $R[q_{min}, q_{max})$, where R represents a range of real numbers with lower boundary q_{min} inclusive and upper boundary q_{max} exclusive.

Note that the data object collection $\mathcal{C}$ can contain zero or more data objects and q_{min} can be $-\infty$, q_{max} can be $+\infty$. Table II lists sample queries over numeric attribute values.

D. Index Essentials

The choice of optimal indexing data structures not only relies on the time and space complexity, but also relies on the scenarios where data structures are used. For example, the self-balancing search tree (SBST) can be used for indexing numbers, while the trie (a.k.a prefix tree) is more appropriate for indexing strings other than numbers. Therefore, functionality must be the first consideration when determining indexing data structures.

The performance of indexing data structures is another crucial factor that needs to be taken into account. However, it is important to note that the performance of data structures can be influenced by the characteristics of the indexed data as well. As such, in this study we investigate the memory footprint, the indexing performance, and the querying performance of different data structures with respect to the characteristics of the indexed metadata.

III. Indexing Data Structures

A. Overview

From the discussion in the previous section, we understand that, for indexing data structures, we focus only on those that can be used to index strings and numbers. According to [31], there are 4 basic categories of indexing data structures, including linear data structures, trees, hash-based structures, and graphs. While graphs are mainly used for representing relationships between data objects, linear data structures, trees and hash-based structures are suitable choices for strings and numbers.

The most prominent linear data structure is an array. Arrays are structured as a series of elements that are placed sequentially in memory where each element is identified by at least one array index or key (usually represented by integers). For example, elements in a 1-D sparse array are accessed via an integer number serving as the array index. In this study, we consider a 1-D sparse array that supports fast look-up of the indexed elements to be the representative linear data structures for indexing numbers.

In contrast to linear data structures, trees, by design, are always used for the representation of hierarchical organizations. Every tree is comprised of a collection of tree nodes that are used for storing data elements or references to data items. Each tree starts from a root node, branches out to one or more internal nodes, and ends at various leaf nodes. Among various trees, the self-balancing search tree (SBST) stores the indexed elements in a hierarchy organized by the order of the index elements, and maintains balance among the branches in terms of the length of the branch. SBSTs can be used for indexing real numbers by utilizing the natural total order that exists in the set of real numbers. Another type of tree, the prefix tree, more commonly known as a trie, does not put the data items on the tree nodes. Instead, it transforms the data items into a sequence of characters, and puts them into its hierarchy according to the lexicographic order of these characters. A trie starts with an empty node and branches out based upon whatever the first characters in different indexed sequences. Tries are ideal for indexing strings that are short in the length. In this study, we use a SBST as data structure for indexing real numbers, while using a trie as the data structure for indexing strings.

As another important category of the indexing data structure, hash-based data structures heavily rely on an associated hash function to locate the indexed elements. Some hash-based data structures, such as bloom and quotient filters, are designed to test whether a data item belongs to a set. Others, including the hash tree and hash trie, include trees and take a hybrid approach. However, since the functionality of retrieving indexed data items is so well supported in an intuitive way, the most prominent and commonly used hash-based data structure is still the hash table. In a hash table, the capability for indexing numbers is similar to that of 1-D sparse array. Therefore, we use hash table as our indexing data structure for strings in this study.

For simplicity and coherency, we use the sample data shown in Table III when presenting the mechanisms of various data structures.

TABLE III: Sample Metadata Attribute Values

String Attribute Values	Numeric Attribute Values
sdR-b2-00115171.fit	8.3665
sdR-r1-00119815.fit	7.72005
sdR-b2-00120324.fit	4.07625
sdR-r1-00123604.fit	2.51506
sdR-r2-00120606.fit	0.201
sdR-b2-00121908.fit	0.03108

B. Indexing Data Structures for Strings

The purpose of indexing data structures is to support various metadata queries. As we have discussed, metadata queries for string values can be classified into four categories - the exact query, the prefix query, the suffix query, and the infix query. As shown in Table IV, we examine the complexity of string-based queries when using the aforementioned hash table and trie as indexing data structures.

TABLE IV: Data structures for string metadata search. We use n for the number of indexed data items, m for the number of buckets in hash table, l for the length of the string to look up, l_i for the length of infix and l_{avg} for the average length of all indexed strings on trie.

Query Type	Hash Table	Trie
Exact Query	$\mathcal{O}(n/m)$	$\mathcal{O}(l)$
Prefix Query	$\mathcal{O}(n)$	$\mathcal{O}(l)$
Suffix Query	$\mathcal{O}(n)$	$\mathcal{O}(l)$
Infix Query	$\mathcal{O}(n)$	$\mathcal{O}(nl_il_{avg})$

1) Hash Table: A hash table [32] is comprised of m buckets which usually can be implemented as an array of m elements. Through the use of one or more hash functions, the data items are mapped into different locations of the hash table. Note that collision can occur due to limited space in the hash table and large volumes of data items to be indexed. Two types of collision resolution techniques are available - open addressing and separate chaining.

The open addressing technique rehashes the conflicting items or simply replaces them in a hash table where the number of buckets is fixed. In contrast, the separate chaining technique allows the total number of indexed data items to exceed the number of buckets in the hash table by extending collision buckets with other data structures such as linked list or self-balancing trees. As the example demonstrated in Figure 2, attribute value "sdR-b2-00115171.fit" and "sdR-r1-00119815.fit" fall into the same bucket since the modulus function yields the same result for the hash codes of these two strings. In order to rectify this situation, they are organized into a linked list.

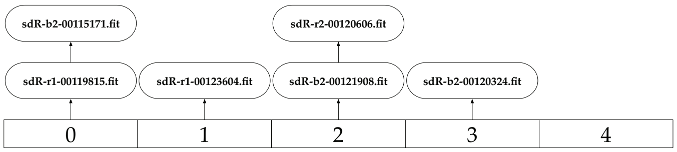

Figure 2: String metadata attribute value on hash table

One can easily apply hash functions on strings and index these strings using a hash table. For exact queries, hash tables are sufficiently effective with $\mathcal{O}(\frac{n}{m})$ time. However, for prefix, suffix, and infix queries, a traversal across the entire hash table is inevitable as result of the need to perform partial string matching against each indexed string. This forces the worst case time complexity, $\mathcal{O}(n)$, for each individual query. For example, when indexing the string values in Table III, it would take at least 1 operation to locate a string value, but it is necessary to go through the entire hash table in order to collect the result matching their common prefix "sdR" or their common suffix ".fit", while infix search on "-001" also requires a full scan of the entire hash table.

2) Tries: Tries [33] (a.k.a. prefix trees, Patricia trees, or radix trees) put every character of each indexed string in the tree nodes.

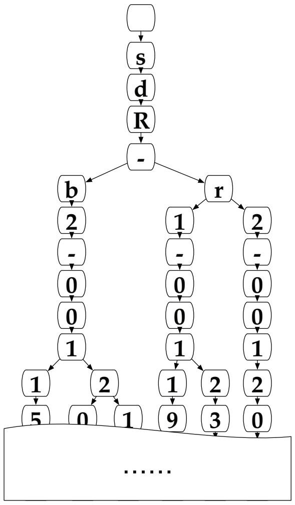

Figure 3: String metadata attribute values on trie. We only show the part of the trie where branching occurs.

For an exact string query with query string of length l, there will be at most l comparisons in a trie. Thus, the time complexity of exact query has an upper bound of $\mathcal{O}(l)$. As shown in Figure 3, 19 comparisons have to be made in order to accomplish the query against string "sdR-b2-00115171.fit" with 19 characters.

For prefix query on a given prefix of length l, l comparisons must be made on the l internal nodes from the root node to leaf nodes. For example, in Figure 3, it would take 4 comparisons to find the longest common prefix "sdR-" for all strings indexed on that trie. The time complexity for finding the longest prefix is therefore $\mathcal{O}(l)$.

Similarly, in order to support suffix query, one can insert the inverse of each string into the trie [21]. When performing a suffix query on the trie, each given suffix can be reversed and the reversed suffix is then used in the query condition. Using this approach, suffix query can be addressed in $\mathcal{O}(l)$ time if the given suffix is of length l. Since the inverse of all index strings must be inserted into the trie as well, the space consumption will be approximately doubled.

However, to find an infix of length l_i over n strings, the entire trie needs to be traversed regardless of whether or not the inverse of string has been inserted into the trie. For example, to find infix "-r1-" on the trie in Figure 3, a string matching on infix "-r1-" has to be made against each branch of the root node of the trie. This causes the time complexity to be $\mathcal{O}(nl_il_{avg})$ (where l_{avg} represents the average length of the strings on the trie), given that the string matching between the given infix and the indexed strings are done with the BoyerMoore string search algorithm [34].

C. Indexing Data Structures for Real Numbers

As discussed in Section II, there are two types of queries against the numeric values of metadata attributes - the exact number query and the range query. As shown in Table V, we examine the sparse array and the SBST for indexing numeric values.

TABLE V: Data structures for numeric metadata search. We use n for the number of indexed data items, m for the number of elements in the sparse array, k for the number of elements within the given range in a range query.

Query Type	Sparse Array	SBST
Exact Query	$\mathcal{O}(n/m)$	$\mathcal{O}(log\ n)$
Range Query	$\mathcal{O}(k)$	$\mathcal{O}(n)$

1) Sparse Array: The concept of a sparse array comes from the sparse matrix [35]. One can consider a sparse array to be a 1-D sparse matrix with only one row but multiple columns. In a sparse array, each element is identified by an integer number. Therefore, when indexing real numbers, the real numbers must first be transformed into their integer format analogs. The original real number can then be placed at the index corresponding to its analog in the sparse array.

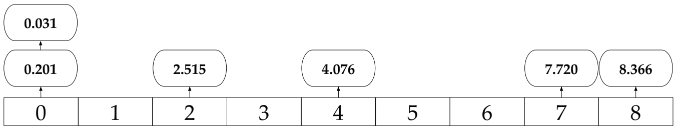

Figure 4: Sparse array on float numbers

As shown in Figure 4, floating-point numbers 0.031 and 0.201 are both assigned to the first element, identified by integer 0, and they are connected via a linked list. When indexing integer numbers, each integer is indexed in a single element of the array, and each element is only used for indexing a single integer number.

Given a sparse array, the exact query can be addressed by locating the element with the given integer as the array index. Such an operation can be efficiently done in $\mathcal{O}(1)$ time if the indexed real numbers are all integers. For floating-point numbers, multiple floating-point numbers may share the same integer digit. Therefore, the time complexity for exact queries can be up to $\mathcal{O}(n/m)$ for a sparse array with m elements. For range queries, the bounding integer(s) serve for location purposes, and k elements within the range are accessed. Therefore, the time complexity of a range query is $\mathcal{O}(k)$.

It is noteworthy that not all elements in sparse arrays will be filled. It usually depends on the distribution of the numeric attribute values among all data objects. Thus, there must be some space overhead in reserving empty elements for possible indexed data items. For small amount of indexed data items, the search efficiency is good. However, when the amount of indexed data items grows sufficiently large, the space overhead may negate the benefit provided by its search efficiency.

2) SBST: Self-balancing search trees (SBSTs) can also be used for indexing real numbers. The natural total order presents in the set of real numbers makes indexing them using SBST straightforward.

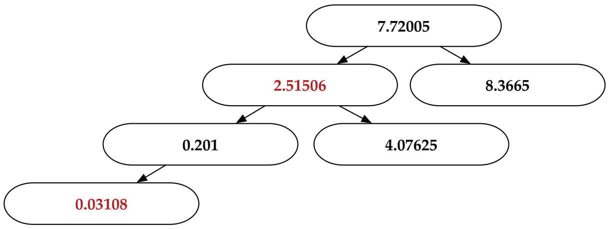

Figure 5: Floating-point numbers on red-black tree

We take a red-black tree as an example. In Figure 5, the floating-point number 2.51506 is larger than 0.201 and smaller than 7.72005. Due to multiple node recoloring and node rotation steps, 2.51506 becomes the left child of root node 7.72005 rather than the root node of the tree.

For an exact query, in the worst case the comparisons have to be made from the root node all the way to the leaf node. Thus, the time complexity of exact query on red-black tree is bounded by $\mathcal{O}(log\ n)$. In fact, for any SBST, the time complexity of exact query is $\mathcal{O}(log\ n)$.

For a range query, the average case is that, some node traversals can be omitted based on the given range. For example, in Figure 5, if the given range is [2, 8), then starting traversal from the root node 7.72005 will not require visiting node 0.03108 for comparison. However, in the worst case, a full scan over the tree is necessary when all the nodes in the tree are within the given range. In this case, the time complexity for a range query is $\mathcal{O}(n)$.

IV. Experimental Evaluation

In this section, we first give a brief overview of our experiment setup, including the platform where we run our experiments and the data structure implementations for our empirical study. We then present the characteristics of datasets in our evaluation and provide our common hypothesis for all the selected data structures according to the characteristics we find in the dataset. Afterwards, for the string attribute values and the numerical attribute values, we further examine the data characteristics and we show how the selected data structures perform in terms of indexing latency, memory consumption, and the latency of various queries. Finally, we provide a brief summary on the findings we observed throughout our evaluations.

A. Experiment Setup

We conducted our experiments on Cori, a Cray XC40 supercomputing system located at the National Energy Research Scientific Computing Center (NERSC). Each of the 2,388 Haswell compute nodes contains two 16-core Intel® Xeon™ E5-2698 v3 (Haswell) 2.3GHz processors and 128GB memory.

We test using the four data structures shown in the “Index Essentials” table in Figure 1, i.e., hash table, trie, SBST, and sparse array. Our discussion in the previous sections has clarified how these data structures cover the 3 major categories of indexing methods: hash-based (hash table), tree-based (SBST and trie), and linear (sparse array). In our experiment, we use `libhl` [36] for the implementation of the selected indexing data structures, and, particularly, we use red-black tree for an implementation of the SBST.

B. Dataset

As our study focuses on the scenario where in-memory index is built for self-describing data format, we consider HDF5 as a representative of the self-describing format, since it is widely used and also the de-facto standard of scientific data format. We choose a real-world dataset from the Baryon Oscillation Spectroscopic Survey (BOSS) [37], which we call the BOSS dataset. The BOSS dataset contains 400 HDF5 files and the size of the entire dataset is over 689 GB. Each individual HDF5 file ranges in size from 1 to 2 GB and contains approximately 4,800 to 23,000 data objects. The entire dataset contains 1,590,881 data objects, which makes metadata search necessary for identifying a specific data object. We consider that a real science dataset, i.e., BOSS, provides us with the authenticity in terms of its volume, its organization and form of the metadata and its data types of the metadata. We have explored several experimental and observational datasets from light sources, astronomy, accelerator physics, and they show similar characteristics to the BOSS dataset. The other end of the spectrum are simulation datasets which have a few objects and do not suffer from metadata search challenge, and hence are out of our consideration.

C. On String Attribute Values

Our selected dataset features metadata attributes whose values are strings and numbers. There are 137 attributes whose values are strings. For indexing string attribute values, we choose to compare the performance of hash table and trie, as mentioned in Section III.

Since it is impossible to evaluate the performance of indexing data structures separately for each string attribute, we need to select representative attributes for our evaluation. Therefore, we select four representative string attributes based on the characteristics of their attribute values. We then investigate the performance of different indexing data structures with respect to the attribute value characteristics.

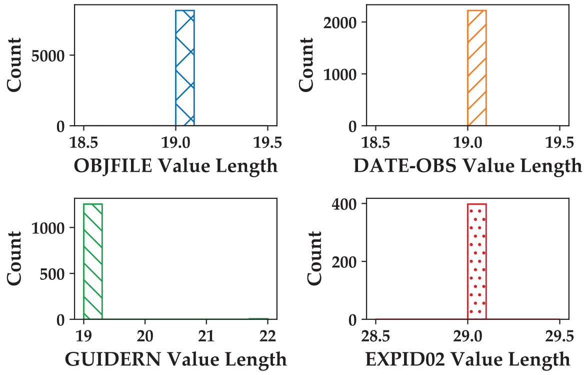

Figure 6: Length of selected string attribute values

As shown in Figure 6, we select attributes “OBJFILE”, “DATE-OBS”,“GUIDERN” and “EXPID02” for our evaluation. The attribute “OBJFILE” has the largest number of unique attribute values at over 8,000. The “DATE-OBS” and “GUIDERN” attributes have 2,222 and 1,261 unique attribute values, respectively. The values of these first three attributes all consist of 19 characters. With 29 characters, the values of attribute “EXPID02” are the longest (29 characters), however, there are only 398 unique attribute values for this attribute. Therefore, the selected attributes contain representative cases for different data characteristics as detailed below. The “OBJFILE” attribute features a large amount of indexed values but moderate value length. “DATA-OBS” and “GUIDERN” both feature moderate amounts of indexed values with moderate value length. Finally, EXPID02 features only a small amount of indexed values but the length of these values is the longest.

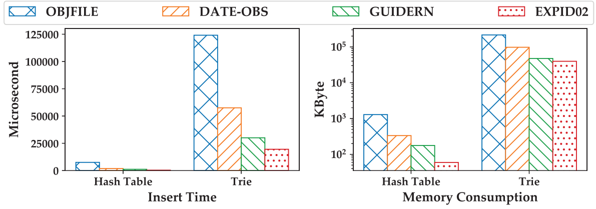

Figure 7: Insertion performance and memory consumption on selected string attribute values

For a single given attribute, the time necessary for inserting all its attribute values is primarily related to the number of times memory allocation occurs. In other words, the more memory space a data structure takes for indexing attribute values, the more time it spends to index them. As shown in Figure 7, a hash table takes the smallest amount of insertion time and space to index the attribute values of our selected attributes (less than 35 milliseconds in time and less than 1.2 MB in memory), while a trie takes the longest (49 - 279 milliseconds in time and 39 - 215 MB in memory). This is because each character in the indexed string results in a node creation in a trie, while hash table indexes the string as a whole.

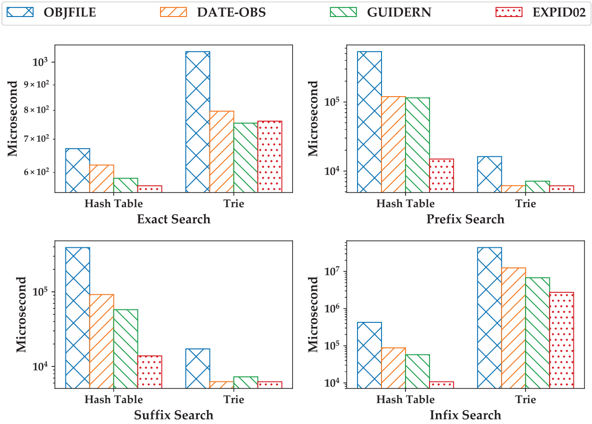

Figure 8: Search performance on selected string attribute values

After creating indexes for string attribute values of our selected attributes, we also perform metadata queries against the string attribute values. For our queries on string attribute values, we test each of our selected attributes using all four aforementioned string query types (exact, prefix, suffix, infix). For example, for attribute “OBJFILE” and its value “sdR-b2-00115171.fit”, we perform an exact query on “sdR-b2-00115171.fit”, prefix query on “sdR-”, suffix query on “.fit”, and also an infix query on “-b2-” (more examples can be seen in Table I). As shown in Figure 8, for exact queries, the hash table outperforms trie and offers the fastest response time (less than 2,000 microseconds) for all 4 selected attributes, which is slightly faster than that of the trie. This is because the exact search on hash table takes $\mathcal{O}(1)$ time which is slightly lower than the $\mathcal{O}(k)$ time of trie. However, when it comes to prefix and suffix queries, the hash table is the most time-consuming while the trie takes the least amount of time. This is a result of the fact that the hash table requires a full scan of the target value, using $\mathcal{O}(n)$ time for both prefix and suffix queries, while the trie only needs to compare the l characters of the prefix/suffix. The latter operation can performed in $\mathcal{O}(l)$ time. When it comes to infix search, once again, the trie again takes the longest time for all four attributes and the hash table is relatively more efficient. The reason for this is still the same as we have discussed for infix search on attribute names.

D. On Numeric Attribute Values

In our BOSS dataset, there are 179 numeric attributes, of which 127 are floating-point attributes and 52 are integer attributes. We select 4 representative attributes for the integer attributes and also 4 representative attributes for the floating-point attributes. We insert the attribute values of each of these attributes into sparse array and SBST - the selected data structures for indexing numeric values, and we show how these data structures perform when indexing different series of attribute values.

As shown in Figure 9, we selected “EXPOSURE”, “COLLB”, “DUSTA”, and “TILEID” to be the four representatives of integer attributes. The total number of distinct values for these attributes are 2222, 1108, 830, and 394

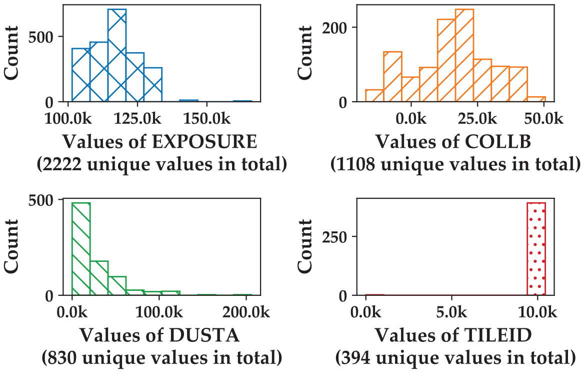

Figure 9: Attribute value distribution of selected integer attributes

respectively. The values of different attributes tend to be distributed within a specific range. For example, the values of attribute "EXPOSURE" are mostly distributed within the range between 100,000 and 135,000, all values of attribute "COLLB" and attribute "DUSTA" are within the range of -20,000 to 50,000 and the range of 0 to 200,000, respectively. For attribute "TILEID", all of its values are within the range between 10,000 to 11,000.

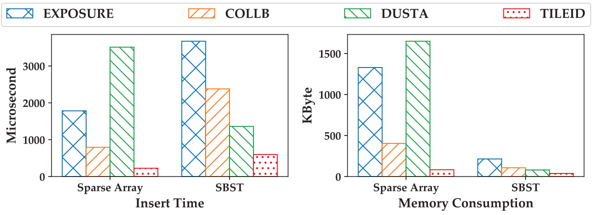

Figure 10: Insertion performance and memory consumption on selected integer attribute values

We insert the values of each attribute into the selected data structures and report the memory consumption and insertion time in Figure 10. In general, the sparse array requires the largest amount of memory while consuming the smallest indexing time (insert time). Orthogonally, SBST has the smallest memory consumption and a moderate indexing time. We can also see that for the sparse array, it would take the most amount of time and space to index the value of "DUSTA", the second most for "EXPOSURE", followed by "COLLB", and finally the least for "TILEID". Sparse arrays must allocate memory for every possible index mapping. However, application behavior often maps values to only a small subset of these indexes. As a result, large amounts of memory space are wasted. This is particularly true for the values of "DUSTA". The entire value range of this attribute falls between 0 and 200,000, but most values are within the range of 0 to 20,000, which causes an excess of wasted memory space. However, for the SBST, the memory consumption is directly proportional to the actual number of indexed data items. Therefore, it consumes less memory space. However, every node in the SBST is individually allocated, and each node corresponds to only a single value, indexing a large number of values can take a significant amount of time.

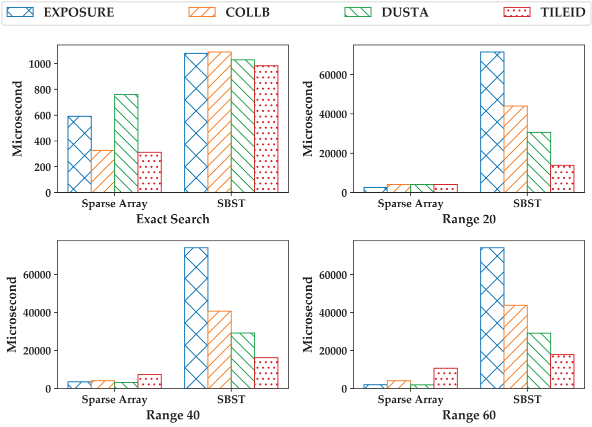

Figure 11: Search performance on selected integer attribute values

After indexing the attribute values of each attribute using the selected data structures, we issue both exact number queries and range queries to test the performance of these data structures. For each selected attribute on a given data structure, we issue 1000 exact number of queries (exact search), 1000 range queries with query range across 20 integer numbers (range 20), 1000 range queries with query range across 40 integer numbers (range 40), and 1000 range queries with query range across 60 integers (range 60). For each range query we issued, the query range starts from a different value randomly selected from the value set of the given attribute. In this case, regardless of the range size, the 1000 range queries still covers a large portion of the actual value range for any selected attribute.

As shown in Figure 11, exact queries on the sparse array spend the least amount of time while the time spent by the SBST remains stable. For all range queries, the sparse array outperforms the SBST in terms of search time. One subtle detail of the range query performance on the sparse array for attribute "TILEID" deserves mentioning. As the range increases from 20 to 60, the query time also grows from 3 ms to 10 ms. The reason for this is that all of attribute "TILEID"s values are in the range between 10,000 to 11,000 while our range queries spanned at most 60 integers. As a result, most of the range queries collected an empty element for each integer in the range. In contrast, the ranges of the other attributes are not fully filled due to the skewed distribution of their values.

After investigating the performance of the selected data structures on integer values, we also selected 4 floating-point attributes for evaluation. As shown in Figure 12, the four attributes we selected still have different number of values (8112 for "DEREDSN2", 2576 for "AZ", 1928 for "RMSOFF20", and 1804 for "ARCOFFX"). The values of "DEREDSN2" range between 0 and 26, and the values of "AZ" range between -100 and 300. For attributes "RMSOFF20", the values are floating-point numbers between 0 and 0.05, and for "ARCOFFX", the values are all between -0.006461 and 0.005509.

As before, we insert the values of each floating-point attribute into the selected data structures. As shown in Figure 13, overall the sparse array has the shortest insertion time and the least memory consumption. This is because the values of the selected attributes are all mapped into a very small integer range. Some attributes like "RMSOFF20" and "ARCOFFX" ,

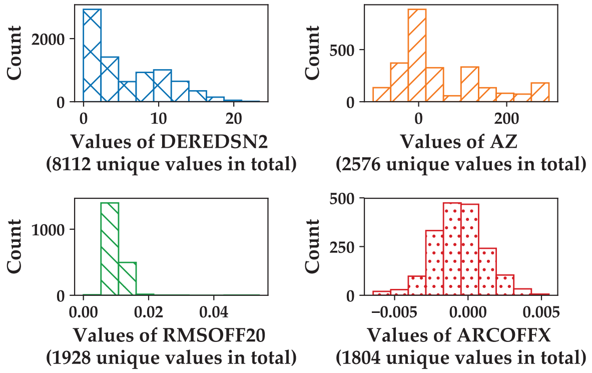

Figure 12: Attribute value distribution of selected floating-point attributes

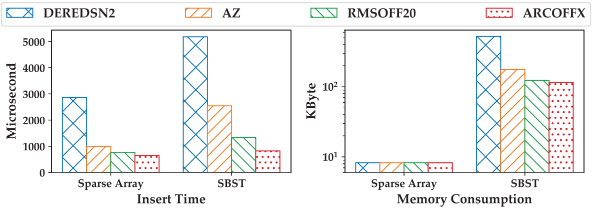

Figure 13: Insertion performance and memory consumption on selected floating-point attribute values

have their values mapped exclusively to integer 0. In this case, the initial size of our sparse array implementation (4096) is sufficient. Then, the SBST takes the longest insertion time and the largest memory consumption, as its space complexity is $\mathcal{O}(n)$ and allocating nodes for the indexed values will take $\mathcal{O}(n)$ time.

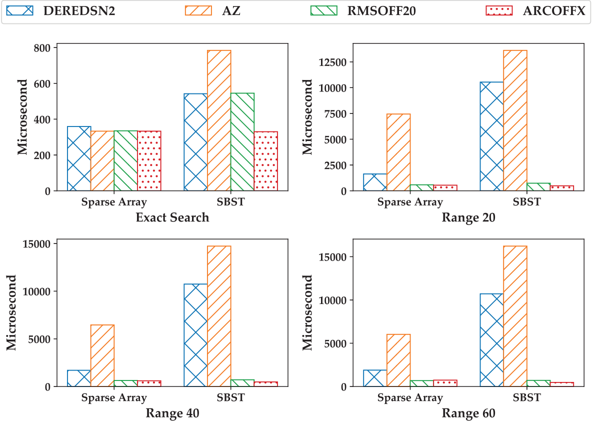

Figure 14: Search performance on selected floating-point attribute values

After indexing the selected floating-point attributes, we continue to investigate the search performance. As reported in Figure 14, we can see that the sparse array still offers the shortest exact search time, while SBST takes longer. This is because, in the sparse array, after mapping a floating-point number into its integer form, locating the element only takes $\mathcal{O}(1)$ time, but the SBST can take $\mathcal{O}(log\ n)$ time. For range queries, all three data structures report long search times on attribute "AZ" as compared to other attributes. We attribute this behavior to the fact that the value range of "AZ" is the largest and most evenly distributed among all the selected attributes. As such, the range queries performed on ”AZ” consistently operate on more value-filled ranges than are typically found within other attributes. Therefore, the range queries on "AZ" will go through more data element or nodes in the selected data structures than on other attributes. The range queries on SBST for "DEREDSN2" still take longer than on the sparse array since the range query time complexity for the SBST is $\mathcal{O}(n)$, but the sparse array takes constant time. Overall, the sparse array still takes the least amount of time for range queries and outperforms the SBST.

E. Summary

Throughout our empirical study, we evaluated a combination of six indexing scenarios, including indexing strings and indexing real numbers. For each scenario, we tested 3 selected data structures. In conclusion, we summarize our findings in Table VI.

TABLE VI: Summary of the empirical study

Query Scenarios	Abundant Memory	Limited Memory
Exact String Query	Hash Table	Hash Table
Prefix Query	Trie	Trie
Suffix Query	Trie	Trie
Infix Query	Hash Table	Hash Table
Exact Number Query	Sparse Array	SBST
Numeric Range Query	Sparse Array	SBST

In this table, we report our findings regarding the indexing data structure best suited to optimized performance with respect available system memory. If only exact query is needed, it is always better to use a hash table for indexing data structures. Whenever prefix/suffix queries are needed, we suggest using a trie regardless of the amount of available memory. For infix queries, a hash table remains the optimal choice regardless of how much memory is available.

For exact queries and range queries on numeric values, when abundant memory is available, we suggest using a sparse array since it outperforms other indexing data structures in terms of search time. In contrast, when utilizing a system with a limited amount of memory, an SBST is the best choice for saving memory while achieving the second best search performance.

V. Conclusion

In this study, we investigated the essentials of metadata search problem comprehensively. These include the metadata essentials regarding the definition and key properties of the metadata, the query essentials regarding the type of various metadata queries and their formations, and also the index essentials regarding the supporting data structures and their performance requirements. Based on the metadata essentials and the query essentials, we analyze the supporting data structures with respect to how they actually work when indexing metadata attributes and we provided an analysis of their time and space complexity. Our empirical study based on the real-world scientific data set provided a comprehensive evaluation on the selected supporting data structures, and we analyzed the performance of each data structure as well. We also concluded with our suggestions on which data structures are desired for use under different circumstances, including the cases when abundant memory is available and the case when memory limitation is enforced. This study provided guidelines and insights for developing advanced metadata indexing methodologies for increasingly critical scientific data management.

Acknowledgment

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, 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, Program manager: Dr. Laura Biven). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *Proceedings of supercomputing*, vol. 99, 1999, pp. 5–33.

[2] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[3] root.cern.ch, "ROOT Files," https://root.cern.ch/input-and-output, 2019.

[4] nasa.gov, "FITS: Flexible Image Transport System," https://fits.gsfc.nasa.gov, 2019.

[5] J. F. Lofstead, S. Klasky *et al.*, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *CLADE*, 2008, pp. 15–24.

[6] S.-A. Dragly, M. H. Mobarhan, M. E. Lepperød, S. Tennøe, M. Fyhn, T. Hafting, and A. Malthe-Sørenssen, "Experimental Directory Structure (Exdir): An alternative to HDF5 without introducing a new file format," *Frontiers in neuroinformatics*, vol. 12, 2018.

[7] M. Wildgrube, "Structured Data Exchange Format (SDXF)," *ietf.org*. [Online]. Available: https://tools.ietf.org/html/rfc3072

[8] P. Greenfield, M. Droettboom, and E. Bray, "ASDF: A new data format for astronomy," *Astronomy and Computing*, vol. 12, pp. 240–251, 2015.

[9] J. Liu, D. Bard, Q. Koziol, S. Bailey, and Prabhat, "Searching for millions of objects in the BOSS spectroscopic survey data with H5Boss," in *2017 New York Scientific Data Summit (NYSDS)*, Aug 2017, pp. 1–9.

[10] C. Chen, Z. Deng, R. Tran, H. Tang, I.-H. Chu, and S. P. Ong, "Accurate force field for molybdenum by machine learning large materials data," *Physical Review Materials*, vol. 1, no. 4, p. 043603, 2017.

[11] A. Mannodi-Kanakkithodi, T. D. Huan, and R. Ramprasad, "Mining materials design rules from data: The example of polymer dielectrics," *Chemistry of Materials*, vol. 29, no. 21, pp. 9001–9010, 2017.

[12] D. Paez-Espino, I. Chen, A. Min, K. Palaniappan, A. Ratner, K. Chu, E. Szeto, M. Pillay, J. Huang, V. M. Markowitz *et al.*, "IMG/VR: a database of cultured and uncultured DNA Viruses and retroviruses," *Nucleic acids research*, vol. 45, no. D1, pp. D457–D465, 2017.

[13] Y. Liu, G. S. H. Pau, and S. Finsterle, "Implicit sampling combined with reduced order modeling for the inversion of vadose zone hydrological data," *Computers & Geosciences*, 2017.

[14] J. J. Donatelli, J. A. Sethian, and P. H. Zwart, "Reconstruction from limited single-particle diffraction data via simultaneous determination of state, orientation, intensity, and phase," *Proceedings of the National Academy of Sciences*, vol. 114, no. 28, pp. 7222–7227, 2017.

[15] M. Aartsen, K. Abraham, M. Ackermann, J. Adams, J. Aguilar, M. Ahlers, M. Ahrens, D. Altmann, K. Andeen, T. Anderson *et al.*, "Search for sources of High-Energy neutrons with four years of data from the Icetop Detector," *The Astrophysical Journal*, vol. 830, no. 2, p. 129, 2016.

[16] M. Aartsen, M. Ackermann, J. Adams, J. Aguilar, M. Ahlers, M. Ahrens, I. Al Samarai, D. Altmann, K. Andeen, T. Anderson *et al.*, "Constraints on galactic neutrino emission with seven years of IceCube data," *The Astrophysical Journal*, vol. 849, no. 1, p. 67, 2017.

[17] Z. Wen, X. Liu, H. Cao, and B. He, "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*, 2018, pp. 1495–1506. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00168

[18] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "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*, 2018, pp. 3:1–3:15. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291660

[19] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S. Lim, and A. R. Butt, "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*, 2017, pp. 5:1–5:12. [Online]. Available: https://doi.org/10.1145/3126908.3126929

[20] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-Centric Metadata Management for High Performance Computing," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 359–369.

[21] W. Zhang, H. Tang, S. Byna, and Y. Chen, "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)*, November 2018.

[22] D. Korenblum, D. Rubin, S. Napel, C. Rodriguez, and C. Beaulieu, "Managing biomedical image metadata for search and retrieval of similar images," *Journal of digital imaging*, vol. 24, no. 4, pp. 739–748, 2011.

[23] M. Lawson and J. Lofstead, "Using a Robust Metadata Management System to Accelerate Scientific Discovery at Extreme Scales," in *Proceedings of the 2nd PDSW-DISCS '18*, November 2018.

[24] T. Craig E., E. Abdelilah, G. Dan *et al.*, "The SPOT Suite project," http://spot.nersc.gov/, 2013.

[25] J. G. Institute, "The JGI Archive and Metadata Organizer(JAMO)," http://cs.lbl.gov/news-media/news/2013/new-metadata-organizer-streamlines-jgi-data-management, 2013.

[26] sqlite.org, "SQLite," https://sqlite.org, 2017.

[27] PostgreSQL, "PostgreSQL," https://www.postgresql.org, 2018.

[28] MongoDB, "MongoDB," https://www.mongodb.com, 2018.

[29] W. Zhang, S. Byna, H. Tang, B. Williams, and Y. Chen, "MIQS: Metadata Indexing and Querying Service for Self-Describing File Formats," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2019.

[30] D. C. Conference, "Scientific Metadata," http://www.dcc.ac.uk/resources/curation-reference-manual/chapters-production/scientific-metadata, 2018.

[31] Wikipedia.org, "List of data structures," https://en.wikipedia.org/wiki/List_of_data_structures, 2019.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Hash Table," in *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009, ch. 11.

[33] R. De La Briandais, "File Searching Using Variable Length Keys," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western). New York, NY, USA: ACM, 1959, pp. 295–298. [Online]. Available: http://doi.acm.org/10.1145/1457838.1457895

[34] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977. [Online]. Available: http://doi.acm.org/10.1145/359842.359859

[35] S. Pissanetzky, *Sparse Matrix Technology-electronic edition*. Academic Press, 1984.

[36] "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, 2018.

[37] C. P. Ahn, R. Alexandroff, C. A. Prieto, S. F. Anderson, T. Anderton, B. H. Andrews, É. Aubourg, S. Bailey, E. Balbinot, R. Barnes *et al.*, "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*, vol. 203, no. 2, p. 21, 2012.