Parallel Query Service for Object-centric Data Management Systems

Houjun Tang, Suren Byna, Bin Dong, and Quincey Koziol
Lawrence Berkeley National Laboratory
Berkeley, California 94720
{htang4, sbyna, dbin, koziol}@lbl.gov

***Abstract*—While large-scale scientific experiments and simulations produce massive amounts of data, a small fraction of data contains useful information. Efficient querying on such volume of data to extract that information increases the productivity of the scientific discovery process. Although querying has been explored extensively in relational databases, research and adoption of querying tools for scientific data that is stored in parallel file systems on high performance computing (HPC) systems are still in infancy. In this paper, we introduce a parallel query service, called PDC-Query, for an object data management systems (ODMS) on HPC systems. It operates on partitioned objects in parallel, and provides several optimization strategies for fast query evaluation. The ODMS paradigm for HPC systems is promising in reducing the burden on users in data management and in moving data transparently across the deep memory hierarchy in modern HPC systems. We propose a 'global histogram'-based approach to accelerate query evaluation, through selectivity estimation and reducing the amount of data that needs to be loaded from storage and processed. We compare querying performance and demonstrate the efficiency and scalability of different approaches PDC-Query supports, including using global histograms, bitmap indexes, sorting, and full scan, in performing various queries on top of a plasma physics dataset with 125 billion particles and an astronomy dataset with 25 million objects.**

I. Introduction

Scientific applications in the upcoming exascale era have pushed data management and storage technology to advance in order to deal with the massive data generated from simulations, experiments, and observations [1]. The deep and heterogeneous memory hierarchy as well as simple and effective application programming interface (API) that relaxes the POSIX-IO semantics have emerged as indispensable components of future HPC data management systems.

Object-based storage systems that manage data as objects can consolidate these new technologies and provide a fast and easy-to-use storage system to users. For example, DAOS [2], MarFS [3], and RADOS [4] have been proposed as the next generation HPC file systems. We have recently developed the Proactive Data Containers (PDC) system [5], [6] that runs in user-space to provide scalable and efficient data and metadata object management. PDC takes advantage of all layers of an HPC system's resources and provides an object interface that hides the complexity of data management from the users.

Though existing object-based systems have demonstrated high scalability and efficiency in managing data and metadata objects, *data querying*, an important aspect of the scientific discovery process in accessing only the data that matches a user-defined condition, is yet to be addressed. Data querying is a natural way to explore and extract information from a massive amount of data. For example, plasma or accelerator physicists often need to locate and/or visualize the highly energetic particles. Being able to specify a query condition, such as "$Energy > 2.0$", and to retrieve all the matching particles efficiently increases productivity. Similar queries exist in other areas of science as well, to retrieve information in data objects that match value range conditions.

Existing data indexing and querying methods are not designed for an object-centric data management system (ODMS), where data are accessed as objects and the objects may reside in different storage devices and across multiple storage layers. For example, SDS-Query [7], FastQuery [8], and ADIOS-Query [9], support querying directly on data files stored in scientific file formats, such as HDF5 and ADIOS-BP, on a single storage layer. SciDB [10] is developed to store and query array-structured data, but it requires converting data to its own format, which can be time-consuming and requires extensive user involvement [11]. Database management systems (DBMS), such as PostgreSQL [12] and MongoDB [13], provide SQL-like query interfaces. However, these systems are not optimized for the data objects containing multi-dimensional array data stored in parallel file systems and result in poor performance [7]. Additionally, DBMSs typically face scalability issues in massive parallel HPC environment and the performance can be several times slower than methods designed for HPC [6].

To provide a solution that overcomes these challenges, we design PDC-Query, a parallel querying service that operates directly on metadata and data objects with an object-centric interface. It allows users to construct both simple and complex query conditions, and offers efficient query processing using a combination of global histograms, sorted data, and bitmap indexes. We have integrated this service into the PDC system and demonstrated its efficiency and scalability with different types of queries. The main contributions of the paper are:

- Introduction to a set of APIs for users to specify query conditions and to execute queries using different indexes.
- Design of a novel global histogram-accelerated parallel query service to provide efficient query processing of

massive amounts of metadata and data objects.
- Evaluation of several query optimization strategies and their impact on the overall query processing performance, including region size selection, region pruning and selectivity estimation with the global histogram, bitmap index, and data reorganization with sorting.

We have evaluated the proposed data query service on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). Experimental results show that our method achieves a multi-fold speedup compared to the conventional approach of scanning the entire data, and the proposed 'global histogram' can complement and further speedup the query processing with indexing and reorganization techniques. The remainder of the paper is organized as follows: We provide a brief background to ODMS and the PDC system in Section II before introducing the parallel query service in Section III. In Section V, we present our experimental setup and show results in Section VI. We discuss the relevant literature in Section VIII and conclude the paper.

II. Background - Object-centric Data Management System

An object-centric data management system (ODMS) can manage any type of data, from small objects (e.g., human-readable text) to large multi-dimensional arrays. By using the Proactive Data Containers (PDC) system [5] as an example, we briefly describe the major components of an ODMS developed for large-scale scientific data management on HPC systems.

PDC [5] offers object-centric APIs, asynchronous data movement across a hierarchy of memory and storage layers, and provides extensible metadata management. PDC uses a client-server model, where the client is a library linked with an application and the servers run different services for metadata management, data movement, and querying (proposed in this paper). These servers run in the user space as additional service processes with minimal disruption to the application. They can also run on dedicated compute nodes away from the application processes. With the PDC system managing data, metadata, and their placement in the storage hierarchy transparently, users are relieved from the burden of managing data manually. More details of the PDC architecture are available in our previous paper [5].

PDC organizes data as a collection of *objects* in a number of *containers*. *Object* is a generic term to describe a byte stream in an abstract manner. Each data object is associated with metadata, including a name, ID, and other attributes such as time of data generation, ownership, relations to other objects, etc. Large objects are partitioned into smaller *regions*, where the actual data as well as the metadata associated with it are maintained. A region is the basic unit in PDC, and can reside on any layer of the memory/storage hierarchy (i.e., main memory, NVRAM, disk, tape, etc.). Such an approach enables the flexibility to spread the data of the same or different objects to various locations and also allow efficient data movement.

In PDC, metadata is managed as an object too. As most metadata are naturally small in size, such as object information, storage location, histogram, etc., they are pre-loaded at server start time and stored as in-memory objects for efficient operations. A metadata object is managed by only one server to guarantee consistency and is periodically persisted to the storage system for fault tolerance.

III. Parallel Querying Service for Objects

The PDC-Query API allows users to construct a simple query condition on a single data object or a combination of multiple conditions to form a complex query. Querying on multiple objects is allowed when the object dimensions are identical. PDC-Query returns either the number of hits for a given query, or the locations (array coordinates) of the matching elements, or both, which is represented as a PDC data selection. Using this selection, a user can load the data from the matching objects into memory. The memory objects may have the same or different data structures from those in the query condition. This flexibility facilitates the most common data query use-cases in scientific applications. We explain more details on the query interface, query evaluation, and various query optimization techniques used in PDC-Query in the following sub-sections.

A. PDC Query interface

We show in Fig. 1 the functions to create and execute queries. `PDCquery_create` creates a query on a single object with the object ID, the operator type ($>, \geq, <, \leq, =$), the data type (float, double, int, unsigned int, long long, unsigned long long, etc.) of the value, and a value pointer. To combine multiple conditional expressions to form a more complex query, we provide `PDCquery_and` and `PDCquery_or`, which can be used to combine conditions either on the same object or on multiple objects with identical array dimensions. A user can construct complex queries by using these three primitive calls. Additionally, the user can specify a region as the spatial constraint of a query, where the region selection can be arbitrary and does not need to match any of the existing PDC internal region partitions.

The `PDCquery_get*` functions are used to evaluate and execute queries using the PDC-Query service and to retrieve the corresponding results to a user-provided memory buffer when the query is completed. `PDCquery_get_selection` is needed before calling `PDCquery_get_data*`, as we require the user to allocate sufficient space to store the data and is responsible to free it afterward. `PDCquery_get_data_batch` is a special case of `PDCquery_get_data` when the resulting data size is too large and cannot fit in memory at one time. The user can use this function to get and process a number of "batches" sequentially. `PDCquery_get_histogram` retrieves the global histogram of an object that is automatically generated by the PDC system at no additional cost. The "free" calls are not listed in the figure. PDC APIs also include querying on metadata and moving data, which were described in our previous work [5]. This paper focuses on the new API introduced above.

```
// Create a one-sided data query
pdcquery_t *PDCquery_create(pdcid_t obj_id, pdcquery_op_t op, pdctype_t type, void *value);
// Combine queries
pdcquery_t *PDCquery_and(pdcquery_t *query1, pdcquery_t *query2);
pdcquery_t *PDCquery_or(pdcquery_t *query1, pdcquery_t *query2);
// Set query region constraint
perr_t PDCquery_set_region(pdcquery_t *query, pdcregion_t *region);
// Query operations
perr_t PDCquery_tag(const char* name, uint32_t valsize, void *val, int *nobj, pdcid_t **obj_ids);
perr_t PDCquery_get_nhits(pdcquery_t *query, uint64_t *n);
perr_t PDCquery_get_selection(pdcquery_t *query, pdcselection_t *sel);
perr_t PDCquery_get_data(pdcid_t obj_id, pdcselection_t *sel, void *data);
perr_t PDCquery_get_data_batch(pdcid_t obj_id, pdcselection_t *sel, uint64_t batch_size, void *data);
pdchistogram_t *PDCquery_get_histogram(pdcid_t obj_id);
```

Fig. 1: PDC query API.

B. Data decomposition

The PDC system decomposes data and breaks a large object (GBs and above) into smaller regions, so that data operations can be easily parallelized. Each region shares the same metadata of the object and has additional metadata such as its offsets and sizes within the object. This approach also allows efficient access to subsets of an object through region selection, eliminates the need to access the entire object when only a small amount is actually needed. All the data processing in the PDC system is performed on these regions.

The selection of region size affects the overall query evaluation performance: a smaller region size allows for better chance to prune regions that have no matching result, and reduces the total amount of data that need to be read from storage. However, it may also result in a large number of regions, bringing additional overhead to store metadata and load them from storage. On the other hand, a region size too large may leads to reading more data than necessary and slows down the overall performance. To determine an efficient region size, we have used an empirical strategy of measuring performance with various region sizes ranging from $4MB$ to $128MB$ and with different types of queries. More details on these measurements are discussed in §VI.

C. Query processing

In Fig. 2, we illustrate the workflow of the PDC-Query service. The application can construct query conditions on one or more objects using the provided query API. Internally in PDC, we use a tree structure to store and represent the query conditions, which allows for chaining an unlimited number of conditions. After the client application finished constructing query conditions, it invokes the corresponding "get result" routines to obtain the number of hits and/or the locations of all matching data elements. The PDC client library automatically serializes the query conditions and broadcasts them to all available servers.

Upon the receipt of a query request, different regions of the queried object are assigned to the servers in a load-balanced fashion. Each server will then obtain the metadata of its assigned regions through the PDC's metadata management service. After the metadata distribution process, the PDC servers do not need to communicate with each other, thus

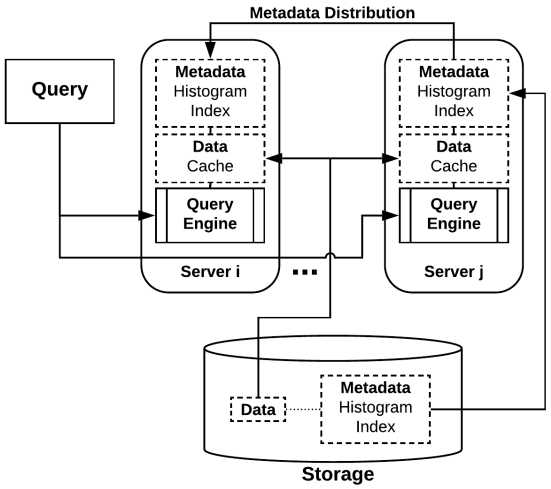

Fig. 2: An overview of PDC's data query service.

reducing the communication overhead.

Once a server receives all necessary metadata of the objects involved in a query, it starts the evaluation process. Each server goes through all the regions assigned to it and records the number of hits and their locations if needed. When the query conditions include multiple objects, they are processed sequentially with the order based on their estimated selectivity. We will provide more details on this in §III-D2. In the case of the intersection (AND) operator between condition statements, we only evaluate the already selected locations matching the first condition for the subsequent conditions, which improves efficiency. For the union (OR) operator, we combine the results from multiple objects and remove the duplicates with a merge sort. A special case for the intersection case is when one query condition has no hit, then there is no need to evaluate the remainder of the query conditions. Similarly for the union operation, if one part of the query selects all elements, we can return them immediately. The servers send the result back to the client after it finishes its query evaluation.

On the PDC client (application) side, after sending the query requests to the servers, a client can either block and wait for the query result or continue to other tasks when the servers are processing, as the communication between PDC clients and servers happens asynchronously. The client has a background thread that aggregates the results received from all servers

before storing them in the user's buffer.

D. Query evaluation strategies

We have implemented several strategies for query evaluation optimization, each can be activated by the user through the setting of an environment variable before running the PDC servers. The histogram only approach is selected by default.

1) ***Full scan****:* To evaluate a data query, a straightforward and baseline approach is to load all the data of the queried object into memory, iterate through all elements, and record the locations that satisfy the query condition. This approach is typically called a "full scan" operation. We have implemented this approach in the PDC system as a building block, and apply optimizations to improve the query evaluation performance based on it. An obvious drawback of the full scan operation is that it requires reading all the objects' data, even if only a few elements satisfy the query condition. Reading data could be costly for a large object even the data is stored in a parallel file system. Effective reduction of the amount of data accesses can significantly improve efficiency, and we explore several approaches in the following sub-sections to achieve it.

2) ***Global histograms****:* The first approach we explored to reduce data access is through the use of histograms. Histograms are often used in database systems to estimate selectivity for query optimization [14]. It provides a representation of the data distribution for the corresponding dataset. It divides values into a number of bins and each bin contains the number of occurrences of data elements in the dataset within its range. Two common binning methods are "equal-width" binning and "equal-height" binning. The "equal-width" histogram divides data into a fixed number of equal-width ranges. The corresponding height of each range represents the number of values falling into that range. On the other hand, "equal-height" histograms have the same number of elements in each bin.

In PDC-Query, a "local" histogram is automatically generated for each data region when data is either produced within PDC or imported from an outside dataset. Depending on the region size, we use 50 to 100 bins. The histogram is used primarily for two aspects: eliminate the need to load regions that do not contain the queried data, and estimate the selectivity of different objects for query optimization.

Using histogram for region elimination: Histograms contain the minimum and maximum value of the corresponding data, which we can use to quickly determine whether the region has any element that satisfies the query condition. While we only need the min and max values from a histogram for the region elimination, we do need all the information a histogram contains to estimate the number of hits satisfying the query condition.

Using histogram for selectivity estimation: When a query involves conditions on multiple objects, the execution order has a significant impact on the overall query evaluation time. For example, when a query has high selectivity on one object and low selectivity on another, executing the former query and getting the matching element locations first and only check elements in those locations for the subsequent objects significantly reduces the overall query execution time. As getting an exact selectivity is costly, we chose to use a histogram that can provide an approximate estimation at a very low cost. To achieve this, we go through the histogram and find all bins that overlap with the query condition, and aggregate their count. The upper bound of the number of hits includes all bins that are fully or partially overlap with the query condition, while the lower bound only counts the fully overlapping bins. Dividing the count by the total number of elements produces the upper and lower bound of the selectivity.

While a regular histogram can achieve the above two purposes, we found that further performance improvement can be achieved if we can merge the local histograms of different regions and obtain a "global" histogram of an entire object. As the metadata is cached in all servers after the metadata distribution, such a global histogram can be used multiple times with very low access latency when serving a series of queries.

3) ***Data reorganization with sorting****:* Range queries are commonly used to explore scientific data. For example, accelerator physics scientists often need to find the high energy particles in the Vector Particle-In-Cell (VPIC) project [15]. Similarly, scientists studying combustion search for array locations where the temperature is between two values. While the data is typically stored based on their spatial location, the resulting data of range queries are often scattered across many regions. In such cases, accessing data from different regions almost always lead to poor performance due to a large number of non-contiguous disk accesses.

When there is prior knowledge on how the data would be queried, sorting and reorganizing the data by value based on one or more objects speeds up the query evaluation process. For example, we can sort the particle data of the VPIC dataset (more details described in the Results section), based on their energy values as it is often the primary queried object. A query condition with high selectivity on the energy object would result in data clustered only in a few regions and thus lead to high efficiency.

The reorganization of data requires additional sorting time, and if the original data has to be kept without reorganization, additional storage space is required to maintain the sorted replica of the data. In PDC, we provide users the option to specify hints on how data should be organized: whether to be sorted and what objects the sort is based on.

4) ***Bitmap index****:* The previously mentioned approaches all require loading the data and then going through them in the evaluation process. An alternative method is to index the data elements and use it for query evaluation. Bitmap index offers efficient searching and data selection retrieval operations. It is especially useful for scientific data as they are generally write-once-read-many and do not require the costly bitmap update operation. We have used the Fastbit [16] indexing library combined with our proposed global histograms as an attempt to speed up the query evaluation process. We construct a bitmap for each region, with the data split into a number

of bins by Fastbit automatically. One representative key is selected in each bin and the value of these representative keys is used to map the original data into 0 or 1 based on these distinct values. The Word-Aligned Hybrid compression (WAH) method is used to reduce the index file size in Fastbit.

Querying with an index can speed up the query evaluation process, however, it also requires extra space to store the index itself as well as extra time to load them into memory. The index file size is expected to be a fraction of the total object size, and reading it is still expected to be faster than reading more data from the storage system for getting the number of hits and/or data selection. We used $precison = 2$ as the default value to construct the Fastbit index, which is sufficient for the queries evaluated in the results section.

E. Data retrieval

The PDC system supports different types of back-end storage systems, such as Lustre and GPFS file systems. The PDC internal data files are hidden from the user, and are not meant to be directly accessed outside the PDC system. Users can see and operate on objects with the PDC's object-centric interface. To speed up the data read performance when the actual data is requested by the query, PDC automatically distributes the data across the parallel file system's storage devices, and uses aggregation methods to merge small reads into bigger ones to reduce the data access contention. This approach offers better performance than setting Lustre stripe parameters of data files.

IV. Global Histogram for Parallel Querying

The generation of global histograms requires all the region histograms to either share the exact same bin boundaries, or their bin sizes are divisible and can be aligned. The former can be achieved by pre-determining the bin boundaries and use them for all histograms generated, however, determining the bin boundaries that can effectively represent the data can be very costly, as we need to scan all data elements. Additionally, in the distributed environment, it requires global communication for parallel processing, which is rather heavyweight and would significantly slow down the overall performance.

As the pre-determined bin boundary approach is impractical, we propose a new method (shown in Algorithm 1) to generate histograms with aligned and divisible bin sizes that can be merged into a global one. We first sample the data to obtain the approximate minimum and maximum values, and calculate the bin width of the pre-determined number of bins (lines 1-2). To better represent different regions that may have very different data distribution, we use non-uniform bin width for different region histogram, with their value rounded to a pre-defined set that are powers of 2 ($..., \pm0.125, \pm0.25, \pm0.5, \pm1, \pm2, \pm4, ...$) (line 3). With such an approach, different histograms may have different bin widths, but their sizes are all divisible to each other.

With the bin size determined, we still need to guarantee that all histograms must have aligned bin boundaries so that they can be merged. We choose to use natural numbers ($\mathbb{N}$) as the first bin boundary's value. As a result, all bin boundary values fall in the set of $\mathbb{N} \pm 2^n, n \in \mathbb{N}$ (lines 4-5). Using this approach does not guarantee the resulting histograms have the same bin numbers as specified, however, this is acceptable for our purposes as using the histogram for selectivity estimation does not require an exact number of bins. Once the bin width and boundaries are determined, we just need to go through the data elements and aggregate the counts (lines 6-18). The time complexity of this algorithm is O(N).

Merging the histograms that were generated using Algorithm 1 into a global one can be done with the following process: first identify the histogram with the largest bin width, which becomes the bin width for the resulting global histogram, and then iterate over each bin of all other histograms, and aggregate the bin count into the aggregated histogram. The merged histogram can have more bins than any of the existing ones if there are non-overlapping bins boundaries. The time complexity of merging histograms is also O(N).

Algorithm 1: Generate a histogram that can be merged into a global histogram

Data: An array S of data elements with $|S| = N$, a lower bound of the number of bins (N_{bin}).
Result: A histogram consisting an array of bin boundaries B ($|B| = 2 * N_{bin}^{'}$), a bin count array C ($|C| = N_{bin}^{'}$), and with $N_{bin}^{'} \geq N_{bin}$.

1 Random sample 10% of the data to get approximate $min(S)$ and $max(S)$
2 $S_{bin} = (max(S) - min(S))/N_{bin}$
/* Round down S_{bin} to power of 2 */
3 $S_{bin}^{'} = \lfloor S_{bin} \rfloor, \lfloor S_{bin} \rfloor \in \pm 2^x, x \in \mathbb{N}$
/* Adjust min and max to be natural numbers $\mathbb{N}$ */
4 $min(S)^{'} = \lfloor min(S) \rfloor + S_{bin}^{'}, \lfloor min(S) \rfloor \in \mathbb{N}$
5 $max(S)^{'} = \lceil max(S) \rceil - S_{bin}^{'}, \lceil max(S) \rceil \in \mathbb{N}$
/* Get the actual number of bins */
6 $N_{bin}^{'} = (max(S)^{'} - min(S)^{'})/S_{bin}^{'}$
/* Generate bin boundaries */
7 **for** $i = 0;\ i < N_{bin}^{'};\ i = i + 1$ **do**
8 $\quad B[i \times 2] = min(S)^{'} + i \times S_{bin}^{'}$
9 $\quad B[i \times 2 + 1] = min(S)^{'} + (i + 1) \times S_{bin}^{'}$
10 **end**
/* Get each bin's count */
11 **for** $i = 0;\ i < N;\ i = i + 1$ **do**
12 $\quad$ Find the bin (j) that includes $S[i]$
13 $\quad$ **if** $S[i] < min(S)^{'}$ **then**
14 $\quad\quad B[0] = min(S)^{'} = S[i]$
15 $\quad$ **else if** $S[i] > max(S)^{'}$ **then**
16 $\quad\quad B[2 \times N_{bin}^{'} - 1] = max(S)^{'} = S[i]$
17 $\quad C[j] + +$
18 **end**

V. Experimental Setup

We ran PDC-Query on Cori supercomputer, located at the National Energy Research Scientific Computing Center (NERSC). Cori is a Cray XC40 system with Intel Haswell and Intel KNL partitions. The Haswell partition, where we ran our experiments, contains 1630 Intel Xeon "Haswell" compute nodes, each consists of 32 cores and 128GB memory. Its Lustre storage system is shared by all Cori users. We ran one PDC server on each compute node in our tests that share the compute and memory resources with the user application, that is, the PDC server occupies one core on each compute node, and the user's application can run on the remaining 31 cores. We compare various configurations of PDC, including varying region sizes, whether to pre-load data, use of histograms, index, and data reorganization with sorting. We set a memory limit of $64GB$ (half the total amount available on a compute node) to be used by each PDC server, which is enough to hold the entire VPIC data for the full scan approach.

We used a $3.3TB$ particle data dataset, which is generated from a plasma physics code called VPIC [17], that simulates magnetic re-connection phenomenon in space weather. VPIC data structure uses 1-D arrays to represent each variable, there are ≈ 125 billion particles with 7 different properties including: Energy, x, y, z, Ux, Uy, and Uz, with each data object $\approx 466GB$ in size. We have constructed 21 different queries with single or multiple constraints to compare the performance for different approaches. For single variable queries, the query constraints range from $3.5 < Energy < 3.6$ (0.0004% selectivity) to $2.1 < Energy < 2.2$ (1.3025% selectivity). For multiple variable queries, it ranges from $Energy > 2.0$ AND $100 < x < 200$ AND $-90 < y < 0$ AND $0 < z < 66$ (0.0013% selectivity) to $Energy > 1.3$ AND $100 < x < 140$ AND $-100 < y < 0$AND $0 < z < 66$ (0.0442% selectivity)

In addition to the particle dataset, we also used the Baryon Oscillation Spectroscopic Survey (BOSS [18]) data, which maps the spatial distribution of galaxies and quasars in the early universe. Each BOSS data object is associated with rich metadata. We have obtained the BOSS data stored in the HDF5 format, which has 2448 files [19]. To measure the performance using the PDC system, we have converted all of the ≈ 25 million objects into our PDC system.

For all the results presented below, unless otherwise specified, we ran the experiments with 64 PDC server processes on 64 compute nodes (64 processes for HDF5 full scan and 64 PDC servers for all PDC experiments). We have measured the elapsed time, which is the end-to-end time from the client issues the query until it receives all the query results. We ran the experiments at least 5 times and reported numbers representing the best performance, which have the least interference from other users on shared system resources (network, storage system, etc.).

VI. Evaluation

We evaluate the performance of executing the 21 different queries and measured the time to get the number of hits along with the matching elements' locations (query time), and the time to load the actual data of matching elements into application memory (get data time). Query time includes the time to read the selected data or index and perform the query evaluation. We compare the PDC-Query's performance with a hand-optimized parallel code using HDF5 to read data stored in HDF5 files and to perform a full scan to obtain the query results (labeled as "HDF5-F" in the plots below). For PDC, we evaluated the performance with various configurations: 1) pre-load all the data of queried objects and a full scan (similar to the HDF5 approach and labeled "PDC-F"), 2) using histograms to reorder the query evaluation sequence and only read and evaluate the regions that have matching elements (labeled "PDC-H"), 3) using histograms and bitmap indexes to obtain the result element selection without the need to read the region's data (labeled "PDC-HI"), and 4) using the histogram and the sorted copy of the object's data to improve the data access and query evaluation efficiency (labeled "PDC-SH"). For each PDC-Query strategy, we also vary the region size from $4MB$ to $128MB$. The Fastbit index file takes $500-600GB$ (15% to 17% of the total data size) of storage space with different region sizes, and the sorted copy requires a full copy of the data, unless the original data can be deleted. We show the results of querying on a single object and on multiple objects, with both metadata and data query conditions, as well as scaling the number of PDC servers in the following sub-sections.

A. Query on a single object

In Fig. 3, we show a comparison of the performance of executing 15 different queries with varying selectivity numbers sequentially. We plot the selectivity on the x-axis and the query execution time on the y-axis.

For the HDF5 and PDC-Query full scan approaches (HDF5-F and PDC-F), we show their amortized time (i.e., [total read time / number of queries] + full scan time) as the evaluation of the queries need to read the entire data into memory once and scan through it to find the matching elements with the query condition. Both times are near constant values as a full scan has approximately the same cost for any query condition. PDC-F achieves up to $2X$ better performance over the HDF5-F in all cases because of the improvement from the initial data read, due to the different data distribution across storage devices and reduced access contention, as mentioned in Section III-E. A small amount of time increase is observed for PDC-F with lower selectivity due to the increasing amount of data that needs to be transferred back to the client through the network. We can also see an improved query performance with larger region sizes (compare PDC-F in different plots of Fig. 3), due to the larger contiguous reads of the regions. However, when the region size increases to more than $64MB$, the performance starts to decrease, as more data need to be loaded from the storage system even a small fraction of data is needed.

For query processing with optimizations (i.e., 'PDC-* Query' in Fig. 3), the most efficient approach for all region sizes is with a sorted copy of the data and with a global

(a) 4MB region size.

(b) 8MB region size.

(c) 16MB region size.

(d) 32MB region size.

(e) 64MB region size.

(f) 128MB region size.

Fig. 3: Single object (Energy) query performance comparison among different approaches and configurations using 64 processes on 64 nodes. HDF5-F: amortized time to process query using full scan with HDF5 for all 15 queries, PDC-F: amortized time to process query using full scan with PDC for 15 queries, PDC-H: PDC with **H**istogram only, PDC-HI: PDC with **H**istogram and Fastbit **I**ndex, PDC-SH: PDC with **S**orted data (sorted by the 'energy' object) and **H**istogram.

histogram ("PDC-SH"). This is mainly because the single object range query processing on sorted data is efficient because the result data are all contiguously stored. The second best approach is "PDC-HI" that uses the Fastbit index, which reads and reconstructs the index instead of the actual object data. "PDC-H" uses histogram only, and is at least twice as fast as either of the full scan approach. We have also observed a decrease in the query evaluation time when more data is selected, this is due to the caching mechanism provided by the PDC, as the queries are evaluated sequentially, an increasing number of the regions' data are cached in the PDC servers' memory and do not require storage access, thus reducing the overall cost for the evaluation process. PDC-SH is over $4X$ faster for a query with 1.3% selectivity, and over $1000X$ faster than the full scan approach for a selectivity of 0.0004%. For PDC-H, the speedup compared to PDC-F is between $2X$ and $3X$, and that for PDC-HI is between $4X$ and $14X$.

Besides getting the number of hits and the data selection, getting the actual values of a query result is often needed. We measured the "get data" performance (which is shown as stacked on top of "query time" in the Fig. 3 plots). For PDC-Query with histogram only (PDC-H) and with sorted data (PDC-SH), the time to get the data with the query selection is very efficient, as all the resulting elements' data are cached in memory during the query evaluation process, and can be directly transferred the data back to the client. PDC-SH takes a longer time due to the results are cached in fewer servers and takes longer time to send to the client. For PDC-HI, since the data is not read from the storage previously, we have to read the regions with matching elements before sending them to the client. As a result, although the query processing time using an index is much faster, the total time to get query results and the data may be similar or even longer than not using any indexes.

B. Query on multiple objects

To further evaluate the effectiveness of PDC-Query service for query conditions on multiple objects, we have constructed 6 queries with conditions on 4 objects, including energy, x, y, and z. The region size has a similar impact on performance to that on the single object queries. Due to the page limit, we show only the results using $32MB$ regions, i.e., with the best region size.

In Fig. 4, we illustrate the performance of different approaches for 6 queries on 4 different objects. We varied the query condition on different objects to demonstrate the effectiveness of our proposed optimizations. As more data (all 4 objects) needs to be read from storage, and fewer queries are evaluated, the amortized time for both approaches increased by a significant amount compared with the single object experiments.

The other three PDC-Query approaches also take a longer time than the single object queries with similar selectivity, which is due to the requirement to read and evaluate data of multiple objects. However, they are still much faster than full scan approaches.

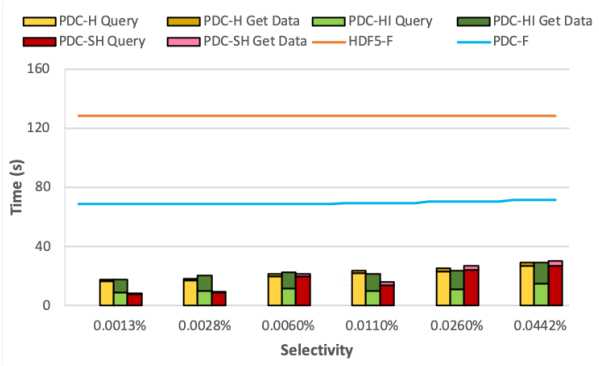

Fig. 4: Multi-object (energy, x, y, z) query performance comparison among different approaches using $32MB$ region size and running with 64 processes on 64 nodes.

Comparing the three optimized PDC-Query methods, we found them to exhibit different behavior of performance than in the single object queries. The sorted approach is not always the fastest to get the number of hits, especially for the last two queries, it is slower than the index approach and takes almost the same time as the histogram-only approach. This is because the last two queries select very few elements of the energy object, which is the primary sort key. In fact, the PDC query engine evaluates the condition of object "x" first, making the sorted reorganization less effective. As with the first two queries that are highly selective on the energy object, the sorted approach still shows the best performance. The index approach offers fast evaluation in all queries regardless of the selectivity on different objects, however, similar to the previous experiments, it takes a longer time to get the actual data as it needs to load actual data from the storage system.

C. Querying on both metadata and data objects

To demonstrate the ability and performance of evaluating queries on both metadata and data, we have imported the H5BOSS dataset (detailed description in the previous section) into the PDC system. H5BOSS has a large number (25 million) of relatively small objects (less than a few MBs each), and scientists are often interested in the data values of a small number of objects that are associated with specific metadata, such as the number of values that are within a range of objects that have a common metadata key-value pair.

We have constructed a set of queries on both metadata and data – e.g., the metadata query condition is "RADEG=153.17 AND DECDEG=23.06", which selects 1000 objects with Right Ascension equal to 153.17 degrees and Declination equals to 23.06 degrees. We vary the data query condition from "$0.0 < \text{flux} < 20$" (11% selectivity) to "$5.0 < \text{flux} < 20$" (65% selectivity) that finds the number of values of the specified range from the flux of fiber objects. Being able to support such queries significantly reduces the time and effort needed by scientists and allowing them to focus more on science than on managing data.

Figure 5 compares the query evaluation time between HDF5 (full scan) and PDC (with histogram only, and with histogram and Fastbit index). From the figure we can see that querying

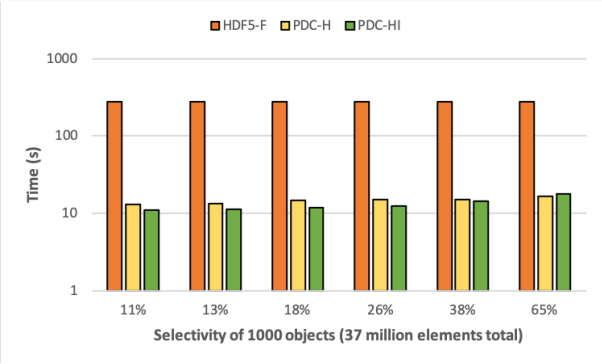

Fig. 5: Comparison of queries with both metadata (fixed selectivity on 1000 objects) and data constraint (varied selectivity from 11% to 65%) on the H5BOSS dataset.

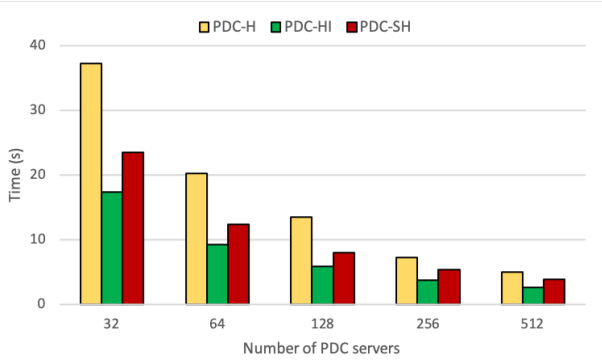

Fig. 6: Query time comparison for a multi-object query condition with 0.011% selectivity using different number of PDC servers.

metadata and data with PDC is much more efficient than the HDF5 approach that requires a traversal of all H5BOSS files. The multi-fold speed up comes mostly from the efficient metadata query service provided by PDC, as it can locate the 1000 objects instantly and start the data query process without the need to look at other irrelevant objects. Due to the small size of the BOSS objects, each object has one region only in PDC-Query, and the entire region is loaded from the storage system, thus the total query processing time does not vary significantly with different selectivity.

D. Scalability

To demonstrate the scalability of the PDC-Query service, we have measured the query performance with a different number of PDC servers. By increasing the number of servers, we increase the parallelism for the query evaluation as each server will process less amount of data. As we only measured the query time for one query, the results of the full scan approach are not included, as their data read time is hundreds of times slower than the query time. In Fig. 6, we show the time of a multi-object query evaluation (0.011% selectivity, using 32 to 512 PDC servers. From the plot, we observe that the query evaluation performance with all three optimizations improves with more servers, which demonstrates the stability of our parallel PDC query service.

VII. Discussion

In summary, we have the following observations from PDC-Query's evaluation. **(1)** When partitioning a large object into smaller regions for parallel processing, the region size influences the query evaluation performance heavily, and we found that region sizes of $32MB$ or $64MB$ for large objects perform well. **(2)** The global histogram-based querying introduced in this paper can effectively accelerate the parallel query planning process with evaluation ordering and hence reducing the number of regions that need to be examined. **(3)** Using an index provides superior query evaluation performance for both single- and multiple-object queries, while the data reorganization with sorting is the most effective when a query is only on that object or is highly selective among all objects' query conditions. **(4)** When the data of the query results need to be loaded to memory, using a strategy that reads and caches the data into memory during query processing offers better performance than using an index for query evaluation. **(5)** When the query condition includes both metadata and data constraint, an efficient metadata query evaluation can significantly accelerate the overall processing time.

VIII. Related Work

Finding a small amount of useful information in a massive scientific dataset through querying is a crucial task for scientific discovery. While this is novel to object-centric data management systems, several technologies have been exploring querying solutions on HPC systems.

"Object storage" is a generic term used to describe an abstract data container that consists of many byte-streams (or *objects*), each with related attributes. Several object store technologies in development, such as DAOS [2], MarFS [3], and RADOS [4], currently provide an object interface that allows basic data and metadata operations, and do not support data querying yet.

In the database community, various techniques have been proposed for decades to optimize the query evaluation process for different database management systems. Building an index using B-tree and its variants [20] have proven to be effective in various commercial DBMS such as Berkeley DB [21], PostgreSQL [12], and MongoDB [13], etc. Object databases that combine database capabilities with object-oriented programming language capabilities have been proposed and developed, such as Gemstone [22], IRO-DB [23], etc.. However, they are not primarily designed to handle the multi-dimensional array data that is commonly used in the scientific community, and often result in poor performance on HPC systems.

SciDB [10] has been developed as a DBMS to store and query array-structured data. However, SciDB requires converting data to its own format, and the data import process is time-consuming and requires extensive user involvement [11]. The result produced by these DBMS is often text-based and needs to be converted to another format for further analysis and/or visualization. Other indexing methods such as Fastbit [16], FastQuery [8], ISABELA [24], and PIQUE [25], have

demonstrated fast index construction, efficient index compression and high query performance.

Various indexing technologies can evaluate and return the query results efficiently, however, the results only include the number of hits and their array locations, while scientific applications often need to read the actual data from the storage system. As the resulting data elements are typically scattered in the datasets, reading them from the storage system can be costly and may require much more time than the query evaluation. Block index [26] is proposed to partition a dataset into fixed-size blocks and record their minimum and maximum values. To speed up the data read performance, each block with matching elements is read entirely to avoid small non-contiguous access. The PDC-query service and the block index share similar concepts to divide large data into smaller parts. However, we use the global histograms to further optimize querying performance for more complex multi-object queries, and our approach is designed for an object-centric data management system.

IX. Conclusions and Future Work

Most scientific discoveries rely on finding a small fraction of key information in massive amounts of data. Data querying is a crucial tool for efficient information retrieval that enhances scientific productivity. With the rapidly growing importance of object-centric data management, developing indexing and querying on data objects is of critical requirement. In this work, we presented PDC-Query, a new object-centric data indexing and querying service, that is highly efficient and scalable. We also proposed a novel indexing with global histograms to accelerate parallel querying on objects by considering object regions. Our evaluation shows these methods are up to 2X to 1000X faster than scanning the entire data.

Our future work aims at bringing query optimization techniques used by relational database management systems to object-centric data management, as well as other data reorganization methods that can provide better performance for multiple variable query conditions.

Acknowledgment

This work is supported 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: Proactive Data Containers, Program manager: Dr. Laura Biven). This research used resources of the National Energy Research Scientific Computing Center, which is 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] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, pp. 59:1–59:12.

[2] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for an Exascale Storage System," in *Supercomputing*, 2016, pp. 50:1–50:12.

[3] J. Inman, D. Bonnie, M. Broomfield, H.-B. Chen *et al.*, "MarFS, Version 1," LANL, Tech. Rep., 2015.

[4] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007, pp. 35–44.

[5] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu *et al.*, "Toward scalable and asynchronous object-centric data management for hpc," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2018, pp. 113–122.

[6] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-centric Metadata Management for High Performance Computing," in *CLUSTER*, 2017, pp. 359–369.

[7] B. Dong, S. Byna, and K. Wu, "Parallel query evaluation as a scientific data service," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 194–202.

[8] J. Chou, K. Wu *et al.*, "Fastquery: A parallel indexing system for scientific data," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 455–464.

[9] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, "Querying large scientific data sets with adaptable IO system ADIOS," in *Asian Conference on Supercomputing Frontiers*. Springer, 2018, pp. 51–69.

[10] P. G. Brown, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *SIGMOD*, 2010, pp. 963–968.

[11] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *SIGMOD*. ACM, 2014, pp. 385–396.

[12] PostgreSQL Global Development Group. PostgreSQL. Http://www.postgresql.org.

[13] MongoDB. MongoDB. Https://www.mongodb.com/.

[14] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1998, pp. 34–43.

[15] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012055.

[16] K. Wu, "Fastbit: an efficient indexing technology for accelerating data-intensive science," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 556.

[17] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[18] K. S. Dawson, D. J. Schlegel, C. P. Ahn, S. F. Anderson, and et al., "The Baryon Oscillation Spectroscopic Survey of SDSS-III," *Astronomical Journal*, vol. 145, p. 10, Jan. 2013.

[19] J. Liu, D. Bard, Q. Koziol, S. Bailey *et al.*, "Searching for millions of objects in the boss spectroscopic survey data with h5boss," in *Scientific Data Summit (NYSDS), 2017 New York*. IEEE, 2017, pp. 1–9.

[20] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[21] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 43–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268751

[22] P. Butterworth, A. Otis, and J. Stein, "The gemstone object database management system," *Communications of the ACM*, vol. 34, no. 10, pp. 64–78, 1991.

[23] G. Gardarin, S. Gannouni, B. Finance, P. Fankhauser *et al.*, "Iro-db-a distributed system federating object and relational databases," in *Object-Oriented Multidatabase SystemsA Solution for Advanced Applications, chapter 20*. Citeseer, 1995.

[24] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," in *European Conference on Parallel Processing*. Springer, 2011, pp. 366–379.

[25] D. A. Boyuka II, H. Tang, K. Bansal, X. Zou, S. Klasky, and N. F. Samatova, "The hyperdyadic index and generalized indexing and query with pique," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM, 2015, p. 20.

[26] T. Wu, J. Chou, S. Hao, B. Dong, S. Klasky, and K. Wu, "Optimizing the query performance of block index through data analysis and i/o modeling," in *Supercomputing*. ACM, 2017, p. 12.