Parallel Data Analysis Directly on Scientific File Formats

Spyros Blanas [1] Kesheng Wu [2] Surendra Byna [2] Bin Dong [2] Arie Shoshani [2]

[1] The Ohio State University
blanas.2@osu.edu

[2] Lawrence Berkeley National Laboratory
{kwu, sbyna, dbin, ashosani}@lbl.gov

ABSTRACT

Scientific experiments and large-scale simulations produce massive amounts of data. Many of these scientific datasets are arrays, and are stored in file formats such as HDF5 and NetCDF. Although scientific data management systems, such as SciDB, are designed to manipulate arrays, there are challenges in integrating these systems into existing analysis workflows. Major barriers include the expensive task of preparing and loading data before querying, and converting the final results to a format that is understood by the existing post-processing and visualization tools. As a consequence, integrating a data management system into an existing scientific data analysis workflow is time-consuming and requires extensive user involvement.

In this paper, we present the design of a new scientific data analysis system that efficiently processes queries directly over data stored in the HDF5 file format. This design choice eliminates the tedious and error-prone data loading process, and makes the query results readily available to the next processing steps of the analysis workflow. Our design leverages the increasing main memory capacities found in supercomputers through bitmap indexing and in-memory query execution. In addition, query processing over the HDF5 data format can be effortlessly parallelized to utilize the ample concurrency available in large-scale supercomputers and modern parallel file systems. We evaluate the performance of our system on a large supercomputing system and experiment with both a synthetic dataset and a real cosmology observation dataset. Our system frequently outperforms the relational database system that the cosmology team currently uses, and is more than 10× faster than Hive when processing data in parallel. Overall, by eliminating the data loading step, our query processing system is more effective in supporting *in situ* scientific analysis workflows.

1. INTRODUCTION

The volume of scientific datasets has been increasing rapidly. Scientific observations, experiments, and large-scale simulations in many domains, such as astronomy, environment, and physics, produce massive amounts of data. The size of these datasets typically ranges from hundreds of gigabytes to tens of petabytes. For example, the Intergovernmental Panel on Climate Change (IPCC) multi-model CMIP-5 archive, which is used for the AR-5 report [22], contains over 10 *petabytes* of climate model data. Scientific experiments, such as the LHC experiment routinely store many gigabytes of data per second for future analysis. As the resolution of scientific data is increasing rapidly due to novel measurement techniques for experimental data and computational advances for simulation data, the data volume is expected to grow even further in the near future.

Scientific data are often stored in data formats that support arrays. The Hierarchical Data Format version 5 (HDF5) [2] and the Network Common Data Form (NetCDF) [1] are two well-known scientific file formats for array data. These file formats are containers for collections of data objects and metadata pertaining to each object. Scientific data stored in these formats is accessed through high-level interfaces, and the library can transparently optimize the I/O depending on the particular storage environment. Applications that use these data format libraries can achieve the peak I/O bandwidth from parallel file systems on large supercomputers [6]. These file formats are well-accepted for scientific computing and are widely supported by visualization and post-processing tools.

The analysis of massive scientific datasets is critical for extracting valuable scientific information. Often, the most vital pieces of information for scientific insights consist of a small fraction of these massive datasets. Because these scientific file format libraries are optimized for storing and retrieving consecutive parts of the data, there are several inefficiencies in accessing individual elements. It is important for the scientific data analysis systems to support such selective data accesses efficiently.

Additionally, scientific file format libraries commonly lack sophisticated query interfaces. For example, searching for data that satisfy a given condition in an HDF5 file today requires accessing the entire dataset and sifting through the data for values that satisfy the condition. Moreover, users analyzing the data need to write custom code to perform this straightforward operation. In comparison, with a database management system (DBMS), searches can be expressed as declarative SQL queries. Existing high-level interfaces for scientific file formats do not support such declarative querying and management capabilities.

Scientific data management and analysis systems, such as SciDB [11], SciQL [39], and ArrayStore [30], have been re-

SIGMOD/PODS'14, June 22 - 27 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2612185 ...$15.00.

cently developed to manage and query scientific data stored as arrays. However, there are major challenges in integrating these systems into scientific data production and analysis workflows. In particular, in order to benefit from the querying capabilities and the storage optimizations of a scientific data management system, the data need to be prepared into a format that the system can load. Preparing and loading large scientific datasets is a tedious and error-prone task for a domain scientist to perform, and hurts scientific productivity during exploratory analysis. Moreover, after executing queries in a data management system, the scientists need to convert the results into a file format that is understood by other tools for further processing and visualization. As a result, the integration of data management systems in scientific data processing workflows has been scarce and the users frequently revert to inefficient and *ad hoc* methods for data analysis.

In this paper, we describe the design of a prototype system, called SDS/Q, that can process queries directly over data that are stored in the HDF5 file format[1]. SDS/Q stands for Scientific Data Services [18], and we describe the Query engine component. Our goal is to support data analysis directly on file formats that are familiar to scientific users. The choice of the HDF5 storage format is based on its popularity.

SDS/Q is capable of executing SQL queries on data stored in the HDF5 file format in an *in situ* fashion, which eliminates the need for expensive data preparation and loading processes. While query processing in database systems is a matured concept, developing the same level of support on array storage formats directly is a challenge. The capability to perform complex operations, such as `JOIN` operations on different HDF5 datasets increases the complexity further. To optimize sifting through massive amounts of data, we designed SDS/Q to use the parallelism available on large supercomputer systems, and we have used in-memory querying execution and bitmap indexing to take advantage of the increasing memory capacities.

We evaluate the performance of SDS/Q on a large supercomputing system and experiment with a real cosmology observations database, called the Palomar Transient Factory (PTF), which is a PostgreSQL database. The schema and the queries running on PostgreSQL were already optimized by a professional database administrator, so this choice gives us a realistic comparison baseline. Our tests show that our system frequently performs better than PostgreSQL without the laborious process of preparing and loading the original data into the data management system. In addition, SDS/Q can effortlessly speed up query processing by requesting more processors from the supercomputer, and outperforms Hive by more than 10× on the same cosmology dataset. This ease of parallelization is an important feature of SDS/Q for processing scientific data.

The following are the contributions of our work:

- We design and develop a prototype *in situ* relational query processing system for querying scientific data in the HDF5 file format. Our prototype system can effortlessly speed up query processing by leveraging the massive parallelism of modern supercomputer systems.
- We combine an in-memory query execution engine and bitmap indexing to significantly improve performance for highly selective queries over HDF5 data.
- We systematically explore the performance of the prototype system when processing data in the HDF5 file format.
- We demonstrate that *in situ* SQL query processing outperforms PostgreSQL when querying a real cosmology dataset, and is more than 10× faster than Hive when querying this dataset in parallel.

The remainder of the paper is structured as follows. First, we discuss related work on scientific data management in Section 2. Then, in Section 3, we describe the SDS/Q prototype. Section 4 follows with a description of the experimental setup and presents the performance evaluation. We conclude and briefly discuss future work in Section 5.

2. RELATED WORK

Several technologies have been proposed to make scientific data manipulation more elegant and efficient. Among these technologies, Array Query Language (AQL) [24], RasDaMan [5], ArrayDB [25], GLADE [16], Chiron [26], Relational Array Mapping (RAM) [34], ArrayStore [30], SciQL [39], MonetDB [21], and SciDB [11] have gained prominence in analyzing scientific data. Libkin et al. [24] propose a declarative query language, called Array Query Language (AQL), for multi-dimensional arrays. AQL treats arrays as functions from index sets to values rather than as collection types. Libkin et al. provide readers/writers for data exchange formats like NetCDF to tie their system to legacy scientific data. ArrayStore [30] is a storage manager for storing array data using regular and arbitrary chunking strategies. ArrayStore supports full scan, subsampling, join, and clustering operations on array data.

Scientific data management systems can operate on arrays and support declarative querying capabilities. RasDaMan [5] is a domain-independent array DBMS for multidimensional arrays. It provides a SQL-based array query language, called RasQL, for optimizing data storage, transfer, and query execution. ArrayDB [25] is a prototype array database system that provides Array Manipulation Language (AML). AML is customizable to support a wide-variety of domain-specific operations on arrays stored in a database. MonetDB [21] is a column-store DBMS supporting applications in data mining, OLAP and data warehousing. Relational Array Mapping (RAM) [34] and SciQL [39] are implemented on top of MonetDB to take advantage of the vertically-fragmented storage model and other optimizations for scientific data. SciQL provides a SQL-based declarative query language for scientific data stored either in tables or arrays. SciDB [11] is a shared-nothing parallel database system for processing multidimensional dense arrays. SciDB supports both the Array Functional Language (AFL) and the Array Query Language (AQL) for analyzing array data. Although these systems can process array data, scientific datasets stored in "legacy" file formats, such as HDF5 and NetCDF, have to be prepared and loaded into these data systems first in order to reap the benefits of their rich querying functionality and performance. In contrast, our solution provides *in situ* analysis support, and allows users to analyze data using SQL queries directly over the HDF5 scientific file format.

[1]The ability to query data directly in their native file format has been referred to as *in situ* processing in the data management community [3].

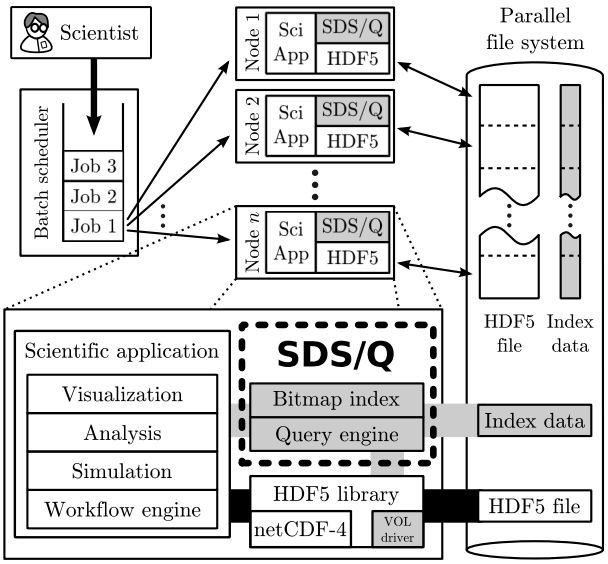


Figure 1: Overview of SDS/Q, the querying component of a prototype scientific data management system. The new components are highlighted in gray.

There have also been efforts to query data in their original file formats. Alagiannis et al. eliminate the data load step for PostgreSQL and run queries directly over comma-separated value files [3]. In this paper, we adapt the positional map Alagiannis et al. proposed and index HDF5 data (see Algorithm 2) and experimentally evaluate its suitability for SDS/Q in Section 4.4.3. Several recent efforts propose techniques to support efficient selections and aggregations over scientific data formats, and are important building blocks for advanced SDS/Q querying capabilities, such as joins between different datasets. Wang et al. [36] propose to automatically annotate HDF5 files with metadata to allow users to query data with a SQL-like language. FastBit [37], an open-source bitmap indexing technology, supports some features of *in situ* analysis as indexes can be queried and stored alongside the original data. FastQuery [17], implemented on top of FastBit, parallelizes FastBit's index generation and query processing operations, and provides a programming interface for executing simple lookups. DIRAQ [23] is a parallel data encoding and reorganization technique that supports range queries while data is being produced by scientific simulations. FlexQuery [40] is another proposed system to perform range queries to support visualization of data while the data is being generated. Su et al. [32] implemented user-defined subsetting and aggregation operations on NetCDF data. Finally, several MapReduce-based solutions have been proposed to query scientific data. SciHadoop [12] enables parallel processing through Hadoop for data stored in the NetCDF format. Wang et al. [35] propose a unified scientific data processing API to enable MapReduce-style processing directly on different scientific file formats.

3. A SCIENTIFIC DATA ANALYSIS SYSTEM

A scientist today uses a diverse set of tools for scientific computation and analysis, such as simulations, visualization tools, and custom scripts to process data. Scientific file format libraries, however, lack the sophisticated querying capabilities found in database management systems. We address this shortcoming by adding relational querying capabilities directly over data stored in the HDF5 file format.

Figure 1 shows an overview of SDS/Q, a prototype *in situ* scientific data analysis system. Compared to a database management system, SDS/Q faces three unique technical challenges. First, SDS/Q relies on the HDF5 library for fast data access. Section 3.1 presents how data are stored in and retrieved from the HDF5 file format. Second, SDS/Q does not have the opportunity to reorganize the original data because legacy applications may manipulate the HDF5 file directly. SDS/Q employs external bitmap indexing to improve performance for highly selective queries, which is described in Section 3.2. Third, SDS/Q needs to leverage the abundant parallelism of large-scale supercomputers which is exposed through a batch processing execution environment. Section 3.3 describes how the query processing is parallelized across and within nodes, presents how HDF5 data are ingested in parallel, and discusses the integration of the bitmap indexing capabilities in the SDS/Q prototype.

3.1 Storing scientific data in HDF5

The Hierarchical Data Format version 5 (HDF5) is a portable file format and library for storing scientific data. The HDF5 library defines an abstract data model, which includes files, data groups, datasets, metadata objects, datatypes, properties, and links between objects. The parallel implementation of the HDF5 library is designed to operate on large supercomputers. This implementation relies on the Message Passing Interface (MPI) [20] and on the I/O implementation of the MPI, known as MPI-IO [33]. The HDF5 library and the MPI-IO layers of the parallel I/O subsystem offer various performance tuning parameters to achieve peak I/O bandwidth on large-scale computing systems [6].

A key concept in HDF5 is the *dataset*, which is a multi-dimensional array of data elements. An HDF5 dataset can contain basic datatypes, such as integers, floats, or characters, as well as composite or user-defined datatypes. The elements of an HDF5 dataset are either stored as a continuous stream of bytes in a file, or in a *chunked* form. The elements within one chunk of an HDF5 dataset are stored contiguously, but chunks may be scattered within the HDF5 file. HDF5 datasets that represent variables from the same event or experiment can be structured as a group. Groups, in turn, can be contained in other groups and form a group hierarchy.

An HDF5 dataset may be stored on disk in a number of different physical layouts. The HDF5 *dataspace* abstraction decouples the logical data view at the application level from the physical representation of the data on disk. Through this abstraction, applications become more portable as they can control the in-memory representation of scientific data without implementing endianness or dimensionality transformations for common operations, such as sub-setting, subsampling, and scatter-gather access. In addition, analysis becomes more efficient, as applications can take advantage of transparent optimizations within the HDF5 library, such as compression, without any modifications.

Reading data from the disk into the memory requires a mapping between the source dataspace (the file dataspace, or *filespace*) and the destination dataspace (the memory dataspace, or *memspace*). The file and the memory dataspaces may have different shapes or sizes, and neither datas-

pace needs to be continuous. A mapping between dataspaces of the same shape but different size performs array sub-setting. A mapping between dataspaces that hold the same number of elements but have different shapes triggers an array transformation. Finally, mapping a non-continuous filespace onto a continuous memspace performs a *gather* operation. Once a mapping has been set up, the `H5Dread()` function of the HDF5 library loads the data from disk into memory.

3.2 Indexing HDF5 data

Indexing is an essential tool for accelerating highly-selective queries in relational database management systems. The same is true for scientific data management systems [29]. However, in order to work with scientific data files *in situ*, one mainly uses secondary indexes, as the original data records cannot usually be reorganized. Other characteristics of scientific workloads include that the data is frequently append-only and that the accesses are read-dominant. Hence, the performance of analytical queries is much more important than the performance of data manipulation queries. Bitmap indexing has been shown to be effective given the above scientific data characteristics [27]. The original bitmap index described by O'Neil [27] indexes each key value precisely and only supports equality encoding. Researchers have since extended bitmap indexing to support range encoding [14] and interval encoding [15]. Scientific datasets often have a large number of distinct values which leads to large bitmap indexes. In many scientific applications, and especially during exploratory analysis, the constants involved in queries are low precision numbers, such as 1-degree temperature increments. Binning techniques have been proposed to allow the user to control the number of bitmaps produced, and indirectly control the index size. Compression is another common technique to control the bitmap index sizes [38].

For this work we use FastBit, an open-source bitmap indexing library [37]. In addition to implementing many bitmap indexing techniques, FastBit also adopts techniques seen in data warehousing engines. For example, it organizes its data in a vertical organization and it implements cost-based optimization for dynamically selecting the most efficient execution strategy among the possible ways of resolving a query condition. Although FastBit is optimized for bitmap indexes that fit in memory, it can handle bitmap indexes that are larger than memory too. If a small part of an index file is needed to resolve the relevant query condition, FastBit only retrieves the necessary part of the index files. FastBit also monitors its own memory usage and could free the unused indexing data structures when the memory is needed for other operations. Most importantly, the FastBit software has a feature that makes it particularly suitable for *in situ* data analysis: its bitmap indexes are generated alongside the base data, without needing to reorganize the base files.

Probing the FastBit index on every lookup, however, is prohibitively expensive because many optimizations in the FastBit implementation aim to maximize throughput for bulk processing. Although these optimizations benefit certain types of scientific data analysis, one needs to optimize for latency to achieve good performance for highly selective queries. To improve the efficiency of point lookups in SDS/Q, we have separated the index lookup operation into three discrete actions: `prepare()`, `refine()` and `perform()`. The first, `prepare()` is called once per query and permits FastBit to optimize for *static* conditions, that is, conditions that are known or can be inferred from the query plan during initialization. In our existing prototype, these are join conditions and user-defined predicates that can be pushed down to the index. Once this initial optimization has been completed, the query execution engine invokes the `refine()` function to further restrict the index condition (for example, to look for a particular key value or key range). Because `refine()` may be invoked multiple times per lookup, we only allow the index condition to be passed in programmatically, as parsing a SQL statement would be prohibitively expensive. Finally, the engine invokes the `perform()` function to do the lookup. FastBit can choose to optimize the execution strategy again prior to retrieving the data, if the optimization is deemed worthwhile.

3.2.1 Automatic index invalidation by in situ updates

SDS/Q maintains scientific data in their original format for *in situ* data analysis, and augments the raw scientific data with indexing information for efficiency. By storing indexing information separately, legacy applications that directly update or append scientific data through the HDF5 library would render indexing information stale. As a consequence, analysis tasks that use indexes may miss newer data and return wrong results. This would severely limit the effectiveness of indexing for certain classes of applications. To ensure the consistency of the index data in the presence of updates, SDS/Q needs to detect that the original dataset has been modified or appended to, and act on the knowledge that the index data are now stale.

We leverage the Virtual Object Layer (VOL) abstraction of the HDF5 library to intercept writes to HDF5 objects and to generate notifications when a dataset changes during *in situ* processing. VOL allows users to customize how HDF5 metadata structures and scientific data are represented on disk. A VOL plugin acts as an intermediary between the HDF5 API and the actual driver that performs the I/O. Using VOL, we can intercept HDF5 I/O requests initiated by legacy applications and notify SDS/Q to invalidate all index data for the dataset in question. In the future, we plan to avoid the expensive index regeneration step and update index data directly.

3.3 Analyzing scientific data

A key component of SDS/Q is a query execution engine that ingests and analyzes data stored in the HDF5 format. The query engine is written in C++, and intermediate results that are produced during query execution are stored in memory. The engine accepts queries in the form of a physical execution plan, which is a tree of relational operators. The HDF5 scan operator (described in detail in Section 3.3.3) retrieves elements from different HDF5 datasets and produces relational tuples. An index scan operator can retrieve data from FastBit indexes. For additional processing, the query engine currently supports a number of widely-used relational operators such as expression evaluation, projection, aggregation, and joins. Aggregations and joins on composite attributes are also supported. We have implemented both hash-based and sort-based aggregation and join operators, as the database research community is currently engaged in an ongoing debate about the merits of hash-based and sort-based methods for in-memory query execution [4].

The execution model is based on the iterator concept, which is pull-based, and is similar to the Volcano system [19]. All operators in our query engine have the same interface, which consists of: a `start()` function for initialization and operation-specific parameter passing (such as the key range for performing an index lookup), a `next()` function for retrieving data, and a `stop()` function for termination. Previous research has shown that retrieving one tuple at a time is prohibitively expensive [10], so the `next()` call returns an array of tuples to amortize the cost of function calls. The pull-based model allows operators to be pipelined arbitrarily and effortlessly to evaluate complex query plans in our prototype system.

3.3.1 Parallelizing across nodes

A scientist analyzes large volumes of data by submitting a parallel job request to the supercomputer's batch processing system. The request includes the number of nodes that need to be allocated before this job starts. When the analysis task is scheduled for execution, the `mpirun` utility is called to spawn one SDS/Q process per node. The scan operators at the leaf level of the query tree partition the input using the `MPI_comm_size()` and `MPI_comm_rank()` functions to retrieve the total number of processes and the unique identifier of this process, respectively. Each node then retrieves and processes a partition of the data, and synchronizes with other nodes in the supercomputer using MPI primitives such as `MPI_Barrier()`.

If intermediate results need to be shuffled across nodes to complete the query, SDS/Q leverages the shared disk architecture of the supercomputer and shuffles data by writing to and reading from the parallel file system. For example, each SDS/Q process participating in a broadcast hash join algorithm produces data in a staging directory in the parallel file system, and then every process builds a hash table by reading every file in the staging directory. Although disk I/O is more expensive than native MPI communication primitives (such as `MPI_Bcast()` and `MPI_Alltoall()`), storing intermediate data in the parallel file system greatly simplifies fault tolerance and resource elasticity for our prototype system.

3.3.2 Parallelizing within a node

At the individual node level, the single SDS/Q process spawns multiple threads to fully utilize all the CPU cores of the compute node. We encapsulate thread creation and management in a specialized operator, the *parallelize* operator. The main data structure for thread synchronization is a multi-producer, single-consumer queue: Multiple threads produce intermediate results concurrently for all operators rooted under the *parallelize* operator in the query tree, but these intermediate results are consumed sequentially by the operator above the *parallelize* operator. The `start()` method spawns the desired number of threads, and returns to the caller when all spawned threads have propagated the `start()` call to the subtree. The producer threads start working and push intermediate results to the queue. The `next()` method pops a result from the queue and returns it to the caller. The `stop()` method of the *parallelize* operator propagates the `stop()` call to the entire subtree, and signals all threads to terminate.

By using multiple worker threads, and not multiple processes per node, all threads can efficiently exchange data, synchronize and share data structures through the common process address space and not rely on expensive inter-process communication. For example, during a hash join, all threads can synergistically share a single hash table, synchronize using spinlocks, and access any hash bucket directly. (This hash join variant has also been shown to use working memory judiciously and perform well in practice [7].) If intermediate results need to be shuffled between threads during query processing (for example, for partitioning), threads exchange pointers and read the data directly.

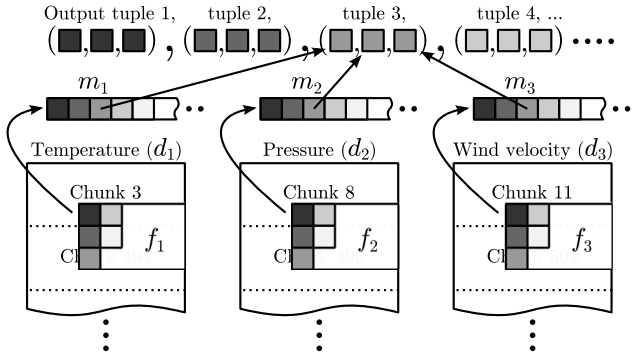


Figure 2: The HDF5 scan operation. Chunks are array regions that are stored contiguously on disk. For every HDF5 dataset d_i, each SDS/Q process operates on a unique subset of the array (the filespace f_i). Every array subset is linearized in memory in the one-dimensional memspace m_i, which becomes one column of the final output.

Algorithm 1 The HDF5 scan operator

```
1: function START(HDF5 datasets {d_1, ..., d_n}, int count)
2:     for each HDF5 dataset d_i do
3:         open dataset d_i, verify data type and shape
4:         create filespace f_i to cover this node's d_i partition
5:         create memspace m_i to hold count elements
6:         map m_i on f_i at element 0
7:     end for
8: end function
9: function NEXT()
10:    for each filespace f_i do
11:        call H5Dread() to populate memspace m_i from f_i
12:        copy memspace m_i into column i of output buffer
13:        advance mapping of m_i on f_i by count elements
14:        if less than count elements remain in filespace f_i then
15:            shrink memspace m_i
16:        end if
17:    end for
18: end function
```

3.3.3 Ingesting data from the HDF5 library

The query engine ingests HDF5 data through the HDF5 scan operator. The HDF5 scan operator reads a set of user-defined HDF5 datasets (each a multi-dimensional array) and "stitches" elements into a relational tuple. The HDF5 library transparently optimizes the data access pattern at the MPI-IO driver layer, while the query execution engine remains oblivious to where the data is physically stored.

The HDF5 scan operation is described in Algorithm 1. The operation is initialized by calling the `start()` method and providing two parameters: (1) a list of HDF5 datasets to access ($\{d_1, d_2, \ldots, d_n\}$), and (2) the desired output buffer

Algorithm 2 Positional map join

```
1: function START(Positional index idx, HDF5 datasets {d_1, ..., d_n}, Join key J_R from R, int count)
2:     for each HDF5 dataset d_i do
3:         open dataset d_i, verify data type and shape
4:         create empty sparse filespace f_i
5:     end for
6:     load idx in memory
7:     create empty hash table HT
8:     for each tuple r ∈ R do
9:         pos ← idx.lookup(π_{J_R}(r))
10:        if join key π_{J_R}(r) found in idx then
11:            expand all filespaces {f_i} to cover element at pos
12:            store r in HT under join key π_{J_R}(r)
13:        end if
14:    end for
15:    for each filespace f_i do
16:        create memspace m_i to hold count elements
17:        map m_i on f_i at element 0
18:    end for
19:    allocate buffer B_S to hold count elements
20: end function
21: function NEXT(Join key J_S from S)
22:    for each dataset d_i do
23:        call H5Dread() to populate memspace m_i from f_i
24:        copy memspace m_i into column i of buffer B_S
25:        advance mapping of m_i on f_i by count elements
26:    end for
27:    for each tuple s ∈ B_S do
28:        for each r in HT such that π_{J_S}(s) = π_{J_R}(r) do
29:            copy s ⋈ r to output buffer
30:        end for
31:    end for
32: end function
```

size (*count*). For example, consider a hypothetical atmospheric dataset shown in Figure 2. The user opens the two-dimensional HDF5 datasets "temperature" (d_1), "pressure" (d_2) and "wind velocity" (d_3). Every process first computes partitions of the dataset and constructs the two-dimensional filespaces $\{f_1, f_2, f_3\}$, which define the array region that will be analyzed by this process. Each process then creates the one-dimensional memspaces $\{m_1, m_2, m_3\}$ to hold *count* elements. (The user defines how the multi-dimensional filespace f_i is linearized in the one-dimensional memspace m_i.) Finally, the HDF5 scan operator maps each memspace m_i to the beginning of the filespace f_i before returning to the caller of the `start()` function.

When the `next()` method is invoked, the HDF5 scan operator iterates through each filespace f_i in sequence and calls `H5Dread()` to retrieve elements from the filespace f_i into memspace m_i. This way, the k-th tuple in the output buffer is the tuple $(m_1[k], m_2[k], \ldots, m_n[k])$. Before returning the filled output buffer to the caller, the operator checks if each memspace m_i needs to be resized because insufficient elements remain in the filespace f_i.

We have experimentally observed that optimally sizing the output buffer of the HDF5 scan operator represents a trade-off between two antagonistic factors. The parallel filesystem and the HDF5 library are optimized for bulk transfers and have high fixed overheads for each read. We have found that transferring less than 4MB per `H5Dread()` call is very inefficient. Amortizing these fixed costs over larger buffer sizes increases performance. On the other hand, buffer sizes greater than 256MB result in large intermediate results. This increases the memory management overhead, and causes final results to arrive unpredictably and in batches, instead of continuously. Very large buffer sizes, therefore, eliminate many of the benefits of continuous pipelined query execution [13]. We have experimentally found that a 32MB output buffer balances these two factors, and we use this output buffer size for our experiments in Section 4.

Algorithm 3 Semi-join with FastBit index

```
1: function START(FastBit index idx, Predicate p on S, Join key J_R from R, int count)
2:     idx.prepare(p)
3:     create empty predicate q(·)
4:     create empty hash table HT
5:     for each tuple r ∈ R do
6:         store r in HT under join key π_{J_R}(r)
7:         q(·) ← q(·) ∨ (π_{J_R}(·) = π_{J_R}(r))
8:     end for
9:     idx.refine(q)
10:    allocate buffer B_S to hold count elements
11: end function
12: function NEXT(Join key J_S from S)
13:    B_S ← idx.perform(count)
14:    for each tuple s ∈ B_S do
15:        for each r in HT such that π_{J_S}(s) = π_{J_R}(r) do
16:            copy s ⋈ r to output buffer
17:        end for
18:    end for
19: end function
```

3.3.4 Positional map join

In addition to index scans, SDS/Q can use indexes to accelerate join processing for *in situ* data processing. Alagiannis et al. have proposed maintaining *positional maps* to remember the locations of attributes in text files and avoid the expensive parsing and tokenization overheads [3]. We extend the idea of maintaining positional information to quickly navigate through HDF5 data, and we exploit this information for join processing. In particular, we use positional information from a separate index file which stores (key, position) pairs, and use this information to create a sparse filespace f_i over the elements of interest in an HDF5 dataset. Mapping a sparse filespace f_i onto a continuous memspace m_i triggers the HDF5 *gather* operation, and the selected data elements are selectively retrieved from disk without loading the entire dataset.

Algorithm 2 describes the positional map join algorithm. R is the table that is streamed in from the next operator in the query tree, and S is the table that the positional index has been built on. The algorithm first creates an empty *sparse* filespace f_i for each dataset of interest and loads the entire positional map in memory (lines 2–7). The index is probed with the join key of each input tuple, and if a matching position (*pos*) is found in the index, the input tuple is stored in a hash table and the position is added to the filespace f_i (lines 8–14). When the `next()` method is called, *count* elements at the selected positions in f_i are retrieved from the HDF5 file and stored in buffer B_S (lines 22–26). Finally, a hash join is performed between B_S and the hash table (lines 27–31) and the results are returned to the caller.

3.3.5 FastBit index semi-join

We now describe an efficient semi-join based index join over the FastBit index (Algorithm 3). We assume that R is the table that is streamed in from the operator below the FastBit semi-join operator, and S is the table that is stored

in the FastBit index. The FastBit index semi-join operator accepts: (1) the FastBit index file (*idx*), (2) an optional selection $\sigma(\cdot)$ to directly apply on the indexed table S, (3) the join key projection $\pi^j(\cdot)$ and the columns of interest $\pi(\cdot)$ of the streamed input table R, and (4) the number of elements that the output buffer will return (*count*). The optional filtering condition $\sigma(\cdot)$ is passed to the initialization function `prepare()` to allow FastBit to perform initial optimizations for static information in the query plan. As the input R is consumed, the algorithm stores the input $\pi(R)$ in a hash table, and adds the join keys $\pi^j(R)$ to the σ^j predicate. When the entire build input has been processed, the σ^j predicate is passed to the `refine()` method to further restrict the index condition before performing the index lookup. The algorithm's `next()` method populates the B_S buffer with *count* tuples from FastBit and joins B_S with the hash table.

4. EXPERIMENTAL SETUP AND RESULTS

In this section we evaluate the performance of the SDS/Q prototype for *in situ* analysis of scientific data. Since our goal is to provide SQL query processing capabilities directly on scientific file formats, we looked for real scientific datasets that are already queried using SQL. We have chosen a PostgreSQL database that contains cosmological observation data from the Palomar Transient Factory (PTF) sky survey. The purpose of our experiments is to run the workload the scientists currently run on PostgreSQL, and compare the query response time to running the same SQL queries on our SDS/Q system, which runs directly on top of HDF5. The advantage of taking this experimental approach is that the relational schema has already been designed by the application scientists, and PostgreSQL has been tuned and optimized by an expert database administrator. This makes our comparisons more realistic.

We conduct our experimental evaluation on the Carver supercomputer at the National Energy Research Scientific Computing Center (NERSC). We describe the NERSC infrastructure in more detail in Section 4.1, and provide the details of our experimental methodology in Section 4.2. We first evaluate the performance of reading data from the HDF5 file format library through controlled experiments with a synthetic workload in Section 4.3. We complete the evaluation of our SDS/Q prototype using the PTF observation database in Section 4.4.

4.1 Computational Platform

NERSC is a high-performance computing facility that serves a large number of scientific teams around the world. We use the Carver system at NERSC for all our experiments, which is an IBM iDataPlex cluster with 1,202 compute nodes. The majority of these compute nodes have dual quad-core Intel Xeon X5550 ("Nehalem") 2.67 GHz processors, for a total of eight cores per node, and 24 GB of memory. The nodes have no local disk; the root file system resides in RAM, and user data are stored in a parallel file system (GPFS).

Scientific applications run on the NERSC infrastructure as a series of batch jobs. The user submits a batch script to the scheduler that specifies (1) an executable to be run, (2) the number of nodes and processing cores that will be needed to run the job, and (3) a wall-clock time limit, after which the job will be terminated. When the requested resources become available, the job starts running and the scheduler begins charging the user (in CPU hours) based on the elapsed wall-clock time and the number of processing cores allocated to the job.

4.2 Experimental methodology

Evaluating system performance on large-scale shared infrastructure poses some unique challenges, as ten-thousand-core parallel systems such as the Carver system are rarely idle. Although the batch queuing system guarantees that a number of compute nodes will be exclusively reserved to a particular task, the parallel file system and the network interconnect offer no performance isolation guarantees. Scientific workloads, in particular, commonly exhibit sudden bursts of I/O activity, such as when a large-scale simulation completes and materializes all data to disk. We have observed that disk performance is volatile, and sudden throughput swings of more than 200% over a period of a few minutes are common. Another challenge is isolating the effect of caching that may happen in multiple layers of the parallel file system and is opaque to user applications. Although requesting different nodes from the batch scheduler for different iterations of the same task bypasses caching at the compute node level, the underlying layers of the parallel file system infrastructure are still shared.

In order to discount the effect of caching and disk contention, all experimental results we present in this paper have been obtained by repeating each experiment every four hours over at least a three day period. As the Carver system has a backlog of analysis jobs, a cool-down period of a few hours ensures that the data touched in a particular experiment will most likely have been evicted from all caches. We report *median* response times when presenting the performance results to reduce the effect of unpredictable background I/O activity from other analysis tasks.

The PTF cosmology workload we used in this study is a PostgreSQL workload and is described in detail in Section 4.4. We use PostgreSQL version 9.3.1 for evaluation, and we report the response time returned from the `\timing` command. We have increased the working memory per query (option `work_mem`) to ensure that intermediate data are not spilled to disk during query execution, and we have also disabled synchronous logging and increased the checkpointing frequency to improve load performance. For the parallel processing performance evaluation, we use Hive (version 0.12) for query processing and Hadoop (version 0.20) as the underlying execution engine, and we schedule one map task per requested processor. All SDS/Q experiments use version 1.8.9 of the HDF5 library and the MPI-IO driver to optimize and parallelize data accesses.

4.3 Evaluation of the HDF5 read performance through a synthetic workload

We now evaluate the efficiency of *in situ* data analysis over the HDF5 array format through a synthetic workload. We consider two access patterns. Section 4.3.1 describes the performance of the full scan operation, where the query engine retrieves all elements of one or more arrays. Section 4.3.2 presents performance results for point lookups, where individual elements are retrieved from specific array locations.

Consider a hypothetical climatology research application that manipulates different observed variables, such as temperature, pressure, etc. We generate synthetic data for evaluation that contain one billion observations and twelve variables $v_1, \ldots, v_{12}$. (Each variable is a random double-precision

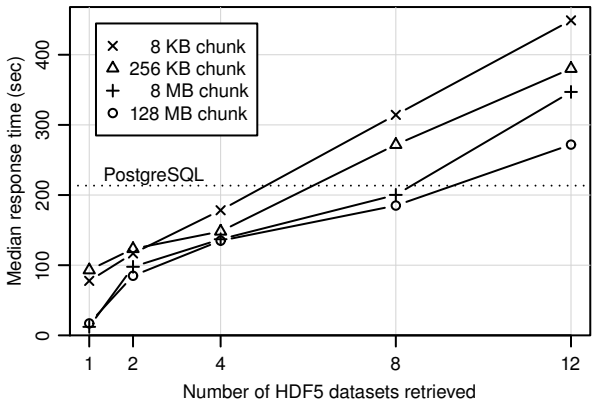


Figure 3: Median response time (in seconds) as a function of the number of HDF5 datasets that are retrieved from the HDF5 file for a full scan.

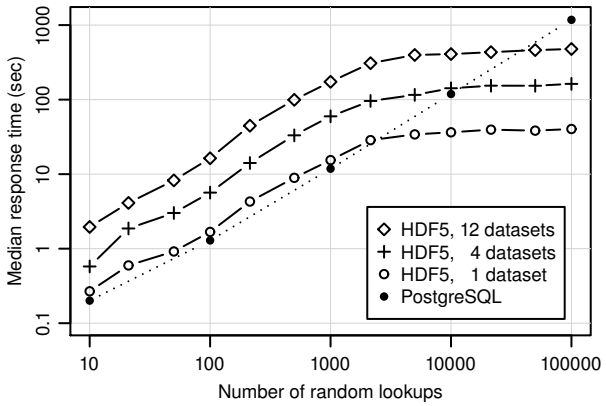


Figure 4: Median response time (in seconds) as a function of the selectivity of the query.

floating point number.) We store this synthetic dataset in two forms: an HDF5 file and a PostgreSQL database. The HDF5 file contains twelve different HDF5 datasets, one for each variable, and each HDF5 dataset is a one-dimensional array that contains one billion eight-byte floating point numbers. Under the relational data model, we represent the same data as a single table with one billion rows and thirteen attributes (id, $v_1, \ldots, v_{12}$): The first attribute id is a four-byte observation identifier, followed by twelve eight-byte attributes $v_1, \ldots, v_{12}$ (one attribute for each variable). Thus, the PostgreSQL table is 100 bytes wide.

4.3.1 Full scan performance

Suppose a scientist wants to combine information from different variables of this hypothetical climatology data collection into a new array and then compute a simple summary statistic over the resulting array. We evaluate the performance of this type of data analysis through a query that retrieves each of $v_1[i], v_2[i], \ldots, v_n[i]$, where $n \leq 12$. For PostgreSQL, the SQL query we run is:

```
SELECT AVG(v1 + v2 + ··· + vn) FROM R;
```

Figure 3 shows the median response time of this analysis (in seconds) on the vertical axis, and the horizontal axis is the number of variables (n) being retrieved. PostgreSQL (the dotted line) scans and aggregates the entire table in about 210 seconds. Because different variables are stored contiguously in a tuple, the disk access pattern is identical regardless of the number of variables being retrieved. Each solid line represents a scan over an HDF5 file with a different chunk size. For HDF5, each variable has been vertically decomposed into a separate HDF5 dataset, and is physically stored in a different location in the HDF5 file. Similar to column-store database systems [10, 31], accessing multiple HDF5 datasets requires reading from different regions of the HDF5 file. When comparing data accesses on HDF5 files with different chunk sizes, we observe that full scan performance benefits from a large chunk size. Based on this performance result, we fix the HDF5 chunk size to 128MB for the remainder of the experimental evaluation.

4.3.2 Point lookup performance

Suppose that the scientist exploring the climatology data decides to inspect specific locations, instead of scan through the entire collection of data. This task is highly selective and the response time of this query critically depends on retrieving data items from specific array locations.

For PostgreSQL, we assume that the data have already been loaded in the relational database system and an index has already been built on the id attribute. In order to isolate the point lookup performance, we force PostgreSQL to use an index by setting the `enable_indexscan` parameter to `on`, and disabling all other scan types. We then retrieve n variables at k random locations $T = (t_1, \ldots, t_k)$ by issuing this SQL query:

```
SELECT AVG(v1 + v2 + ··· + vn) FROM R WHERE id IN T;
```

We also perform the same analysis directly on the HDF5 data file by specifying a sparse filespace that contains k random elements of interest. We then retrieve the data at these locations from n arrays, where each array is stored as a separate HDF5 dataset.

Figure 4 shows the median response time (in seconds) on the vertical axis, as the number of random point lookups (k) grows on the horizontal axis. The performance of PostgreSQL is not affected by the number of variables inspected (n), because data from different variables are stored in separate columns of the same tuple and retrieving these variables does not incur additional I/O. We therefore show a single line for PostgreSQL for clarity. In comparison, retrieving each HDF5 variable requires an additional random lookup, which causes response time to degrade proportionally to the number of variables being retrieved.

When few random lookups happen, the response time for accessing data in HDF5 is slower than PostgreSQL. As the number of random lookups grows, the response time for accessing HDF5 data increases less rapidly after 2,000 point lookups, and becomes faster than PostgreSQL after tens of thousands of lookups. This happens because the first read in a particular HDF5 chunk is significantly more expensive than subsequent accesses. When there is a single lookup per HDF5 chunk, the dominating cost is the access penalty associated with retrieving chunk metadata to compute the appropriate read location. When many random array locations are accessed, the number of read requests per chunk increases and the chunk metadata access cost is amortized over more lookups. As a result, query response time increases less rapidly.

Table	Cardinality	Columns
candidate	672,912,156	48
rb_classifier	672,906,737	9
subtraction	1,039,758	51

Table 1: Our snapshot of the PTF database.

Query	Matching σ_s time interval	Tuples in subtraction that match σ_s	Cardinality of the result
Q1	hour	226	5,878
Q2	night	2,876	42,530
Q3	week	21,026	339,634

Table 2: Number of tuples that match the filter condition, and cardinalities of the final answers.

4.3.3 Summary of results from the synthetic dataset

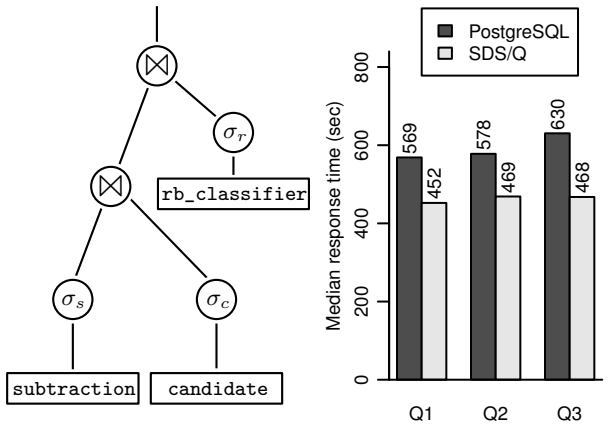
We find that the performance of a sequential data scan with the HDF5 library is comparable to the performance of PostgreSQL, if the HDF5 chunk size is hundreds of megabytes. A full scan over a single HDF5 dataset can be significantly faster because of the vertically-partitioned nature of the HDF5 file format. Performance drops when multiple HDF5 datasets are retrieved, because these accesses will be scattered within the HDF5 file. When comparing the point lookup performance of the HDF5 library with PostgreSQL, we find that accessing individual elements from multiple HDF5 datasets can be one order of magnitude slower than one B-tree lookup in PostgreSQL, because the HDF5 accesses have no locality.

4.4 Evaluating the SDS/Q prototype using the Palomar Transient Factory workload

The Palomar Transient Factory is an automated survey of the sky for transient astronomical events, such as supernovae. Over the past 15 years, observations of supernovae have been the basis for scientific breakthroughs such as the accelerating universe [28]. Because the phenomena are transient, a critical component of the survey is the automated transient detection pipeline that enables the early detection of these events. A delay of a few hours in the data processing pipeline may mean that astronomers will need to wait for the next night to observe an event of interest.

The data processing pipeline starts on the site of the wide-field survey camera. The first step is image subtraction, where event candidates are isolated and extracted. The candidates are then classified through a complex machine learning pipeline [9] which detects whether the candidate is real or an image artifact, and if it is real, what is the transient type of the candidate (supernova, variable star, etc.). After this initial processing, each candidate event is described with 47 different variables, and each image subtraction with 50 variables. The classifier has produced a list of matches between candidate events and image subtractions (an M-to-N relationship) and 7 generated variables containing confidence scores about each match.

The data are then loaded into a PostgreSQL database for processing. The database schema consists of three tables. The `candidate` table contains one row for each candidate event, the `subtraction` table contains one row for each image subtraction, and the `rb_classifier` table has one row per potential match between a candidate event and an image



(a) PTF query plan. (b) Query response time.

Figure 5: Query performance for the three PTF queries when performing a full scan.

subtraction. Every variable is stored as a separate column, and an `id` field is prepended to differentiate each observation and combine information through joins between the tables. We have obtained a nightly snapshot of the database used by the PTF team for near-real-time transient event detection for evaluating our SDS/Q prototype. The snapshot we experiment with is a PostgreSQL database of about 180GB of raw data, with another 200GB of space used by indexes to speed up query processing. The cardinalities of the three tables, and the size of each tuple are shown in Table 1.

The workhorse query for this detection workload is a three-way join. The query seeks to find events in the `candidate` table exceeding certain size and brightness thresholds, that have been observed in images in the `subtraction` table over some time interval of interest, and are high-confidence matches (based on the scores in the `rb_classifier` table.) The final result returns 18 columns for further processing and visualization by scripts written by the domain experts. PostgreSQL evaluates this query using the query plan shown in Figure 5(a). By varying the time interval of interest, we create three queries with different selectivities from this query template. Each query corresponds to three different scientific use cases. Q1 looks for matches over the time span of one hour and is extremely selective, as it returns only 0.02% of the tuples in the `subtraction` table. This is an example of a near-real-time query that runs periodically during the night. Q2 matches an entire night of observations, and returns 0.3% of the tuples in the `subtraction` table. Finally, Q3 is looking for matches over last week's image archive. The selectivity of the condition on the `subtraction` table is about 2%. Table 2 shows how many tuples match the filter condition, and the cardinality of the answer.

4.4.1 Query performance without indexing

If an index has not been constructed yet, PostgreSQL evaluates all PTF queries by scanning the entire input relations, and uses the hash join as the join algorithm of choice. We execute the same query plan in SDS/Q to evaluate the performance of *in situ* data processing in the HDF5 file format.

Figure 5(b) shows the median response time (in seconds) on the vertical axis for each of the three queries. We find that the response time of SDS/Q is similar to that of PostgreSQL for all three queries, and SDS/Q completes all queries about

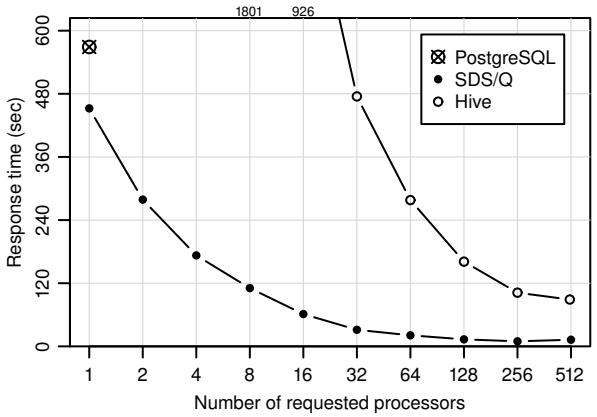


Figure 6: PTF Q1 response time as more processors are requested from the job scheduler.

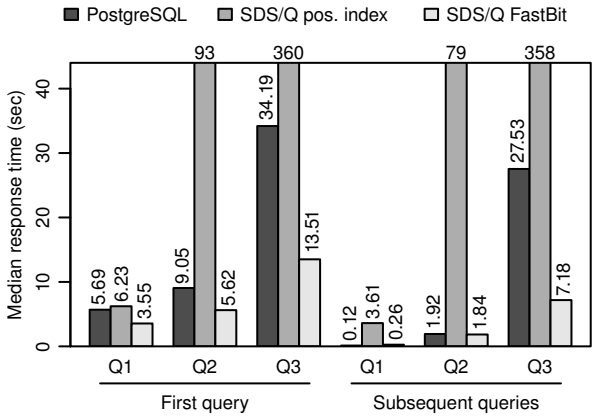


Figure 7: Median response time of the three PTF queries when accessing data through an index.

25% faster. SDS/Q outperforms PostgreSQL because the query retrieves HDF5 datasets selectively from each table, as only about 10 out of the approximately 50 attributes of the `candidate` and `rb_classifier` table are accessed. (We have systematically explored this effect in Section 4.3.1.) The performance improvement for SDS/Q is because of the vertically-partitioned layout of the HDF5 datasets that results in less I/O for the three PTF queries.

4.4.2 Parallel processing of native HDF5 data

All experiments so far have presented single-threaded performance. A major advantage of native scientific data formats such as HDF5 is that they have been designed for parallel processing on modern supercomputers. In this section, we evaluate how the response time for PTF Q1 improves (on the same data) if one requests more than one processing core when the batch job is submitted for execution.

Figure 6 shows the response time of the PTF Q1 query as more processors are requested when the job is submitted for execution. We request n cores to be allocated on $\lceil \frac{n}{8} \rceil$ compute nodes. PostgreSQL has not been designed for a parallel computing environment, therefore we only show response time for a single core. We instead compare the performance of SDS/Q with the Apache Hadoop data processing stack. As the PTF queries generate small intermediate results (cf. Table 2), we have rewritten the query to force each join to use the efficient map-only broadcast join, instead of the expensive repartition join [8]. Analyzing the PTF dataset with Hive using 8 processors (one node) takes 30 minutes, but performance improves almost proportionally with the number of processors: the analysis takes less than two minutes when using 256 cores. The poor performance of Hive is rooted in the fact that the Apache Hadoop stack has not been designed for a high-performance environment like the Carver supercomputer, where the parallel file system can offer up to 80GB/sec of disk read throughput. As indicated by the near-optimal speedup when allocating more processors, the Hive query is a CPU-bound task in this environment.

SDS/Q parallelizes the PTF Q1 query to use all n cores automatically in the HDF5 scan operator (see Algorithm 1), and uses the broadcast hash join to create identical hash tables in every node to compute the join result. The performance of SDS/Q improves significantly from parallel data processing, and when using 16 cores (two nodes) the response time drops to less than one minute. There are limited performance improvements when using more than 64 cores. This is caused by two factors. First, there are high fixed overheads to initiate a new job. Given the small size of the original database, these fixed overheads become significant very quickly at scale. (For example, at 128 cores each spawned task is processing less than 2GB of data.) The second factor is skew. The references on the `candidate` and `rb_classifier` tables exhibit high locality for any chosen time range on the `subtraction` table. This causes a few cores to encounter many matches that propagate across the two joins, while the majority of the processors find no matches after the first join and terminate quickly. In summary, we find that SDS/Q can significantly improve query processing performance when more processors are requested, and SDS/Q completes the PTF Q1 query $10\times - 15\times$ faster than Hive on the Carver supercomputer.

4.4.3 Using an index for highly selective queries

We now turn our attention to how indexing can improve the response time of highly selective queries. PostgreSQL relies on an index for the range lookup on `subtraction`, and on index-based joins for the `rb_classifier` and `candidate` tables. We adopt the same query plan for SDS/Q.

Figure 7 shows the response time for all PTF queries when accessing data through an index. We ran a given query multiple times, in sequence, and report the response time of the "cold" first query on the left, and the "warm" subsequent queries on the right. PostgreSQL relies on a B-tree index, and SDS/Q uses either a positional index (Algorithm 2) or FastBit (Algorithm 3).

Performance is poor for all queries that use the positional index. As described in detail in Algorithm 2, this operation consists of two steps: it first retrieves offsets from the positional index, and then retrieves specific elements from the HDF5 file at these offsets. As shown in prior work [3], retrieving the offsets from the positional index is fast. The performance bottleneck is retrieving elements at specific offsets in the HDF5 file, which we have explored systematically in Section 4.3.2 (cf. Figure 4). Aside from response time, another important consideration is the amount of working memory that a join requires [7]. The FastBit index semi-join and positional map join have different memory footprints.

System	Time (minutes)	
	Load	Index
PostgreSQL	212.4	99.6
SDS/Q	0	11.3
Hive	0	N/A

Table 3: Data preparation time (in minutes) of different systems for the PTF database.

As FastBit is forming an intermediate result table, it may access a number of different bitmap indexes for each retrieved variable, which requires more working memory. Therefore, positional indexing may still be appropriate when the available working memory is limited.

Both PostgreSQL and SDS/Q with FastBit return results nearly instantaneously for the most selective PTF query (Q1). As the query becomes less selective (Q3), the query response time from SDS/Q is 3.8× faster than PostgreSQL. PostgreSQL benefits from high selectivity as it stores each tuple contiguously: all variables can be retrieved with only one I/O request. In comparison, SDS/Q relies on FastBit for both indexing and data access. FastBit partitions data vertically during indexing, hence SDS/Q needs to perform multiple I/O requests to retrieve all data. As the query becomes less selective, SDS/Q can identify all matching tuples at once from the bitmap indexes on `rb_classifier` and `candidate` (see Algorithm 3). In comparison, PostgreSQL has to perform multiple B-tree traversals to retrieve all data.

To summarize, we find that SDS/Q with FastBit bitmap indexing delivers performance that is comparable to that of PostgreSQL for extremely selective queries. As the queries become less selective, the bitmap index favors SDS/Q, where it outperforms PostgreSQL by nearly 4×. Positional indexing performs poorly due to the inherent cost of performing random point lookups over HDF5 data.

4.4.4 The user perspective: end-to-end time

The user of a scientific data analysis system wants to understand the data produced by some simulation or observed during an experiment. For the case of the PTF dataset, the astronomer has the choice of loading the data in a relational database system, which takes significant time (see Table 3). Alternatively the astronomer can choose to process the data *in situ* using SDS/Q or Hadoop [12], leaving data in a file format that is understood by the analysis scripts and visualization tools she is using already.

Figure 8 shows the time it takes for the astronomer to gain the desired insight from the PTF database on the vertical axis, as a function of the number of Q1 queries that need to be completed on the horizontal axis. Using four Carver compute nodes, the scientist can get an answer to her first question in about 30 seconds by running SDS/Q on all processing cores. In comparison, the first answer from Hive needs approximately eight minutes. PostgreSQL would respond after the time-consuming data load process is complete, which takes four hours. If the scientist desires to ask multiple questions over the same dataset, the indexing capabilities of SDS/Q can produce the final answer faster than loading and indexing the data in PostgreSQL, or performing full scans in parallel. Due to the cumbersome data load process of the PTF database, the sophisticated querying capabilities of a relational database only outperform SDS/Q after many thousands of queries. SDS/Q allows scientists

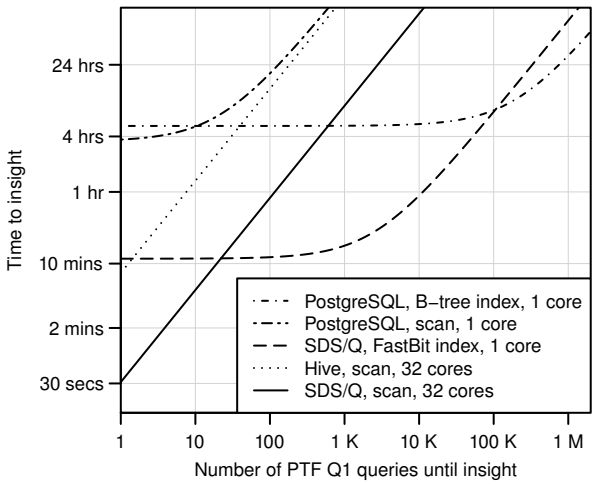


Figure 8: Time to insight as a function of the number of the PTF Q1 queries completed. The result for the first query includes the data preparation time.

to take advantage of the ample parallelism of a supercomputer through a parallel relational query processing engine and a fast bitmap index without modifying their existing data analysis workflow.

5. CONCLUSIONS

We propose an *in situ* relational parallel query processing system that operates directly on the popular HDF5 scientific file format. This prototype system, named SDS/Q, provides a relational query interface that directly accesses data stored in the HDF5 format without first loading the data. By removing the need to convert and load the data prior to querying, we are able to answer the first query of the Palomar Transient Factory (PTF) cosmology workload more than 40× faster than loading and querying the relational database system that the scientists use today.

In addition, SDS/Q is able to easily take advantage of the parallelism available in a supercomputing system and effortlessly speed up query processing by simply requesting more CPU cores when the batch job is submitted for execution. Given that parallel database systems are often too expensive for large scientific applications, this ease of parallelization is an important feature of SDS/Q for processing scientific data. In a test with the same query from the PTF cosmology dataset, SDS/Q responds 10× faster than Apache Hive when running on 512 CPU cores.

We also demonstrate that SDS/Q can use bitmap indexes to significantly reduce query response times. The bitmap indexes are external indexes and do not alter the base data stored in the HDF5 files. On sample queries from the same PTF workload, these bitmap indexes are able to reduce the query processing time from hundreds of seconds to a few seconds or less.

We plan to focus our future work in reducing the time to generate the bitmap indexes. In the current prototype, the bitmap indexes are built with intermediate data files which are generated from the HDF5 data. We are working to accelerate the index build phase by directly reading data from the HDF5 file. This will reduce the time needed to answer the first query using a bitmap index. We also plan to extend

our Scientific Data Services to support operations that are important to scientific applications that manipulate large scientific datasets, such as transparent transposition, and reorganization based on query patterns. Finally, query optimization on shared-disk supercomputers has unique challenges and opportunities, and we see this as a promising avenue for future work.

Source code

The source code for the core SDS/Q components is available:
- Bitmap index: `https://sdm.lbl.gov/fastbit/`
- Query engine: `https://github.com/sblanas/pythia/`

Acknowledgements

We would like to acknowledge the insightful comments and suggestions of three anonymous reviewers that greatly improved this paper. In addition, we thank Avrilia Floratou and Yushu Yao for assisting with the Hive setup, and Peter Nugent for answering questions about the PTF workload. This work was supported in part by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and used resources of The National Energy Research Scientific Computing Center (NERSC).

6. REFERENCES

[1] NetCDF. `http://www.unidata.ucar.edu/software/netcdf`.
[2] The HDF5 Format. `http://www.hdfgroup.org/HDF5/`.
[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. In *SIGMOD*, pages 241–252, 2012.
[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *ACM SIGMOD*, 1998.
[6] B. Behzad, H. V. T. Luu, J. Huchette, et al. Taming parallel I/O complexity with auto-tuning. In *SC*, 2013.
[7] S. Blanas and J. M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *SoCC*, 2013.
[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *ACM SIGMOD*, 2010.
[9] J. S. Bloom, J. W. Richards, P. E. Nugent, et al. Automating discovery and classification of transients and variable stars in the synoptic survey era. *arXiv preprint arXiv:1106.5491*, 2011.
[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.
[11] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *ACM SIGMOD*, 2010.
[12] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: array-based query processing in Hadoop. In *SC*, 2011.
[13] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.
[14] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, 1998.
[15] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, 1999.
[16] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big data analytics made easy. In *SIGMOD*, pages 697–700, 2012.
[17] J. Chou, K. Wu, and Prabhat. FastQuery: A general indexing and querying system for scientific data. In *SSDBM*, pages 573–574, 2011.
[18] B. Dong, S. Byna, and K. Wu. SDS: A framework for scientific data services. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 27–32, 2013.
[19] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.
[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22(6):789–828, Sept. 1996.
[21] S. Idreos, F. Groffen, N. Nes, et al. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
[22] IPCC 2013. *Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, in press.
[23] S. Lakshminarasimhan, D. A. Boyuka, et al. Scalable in situ scientific data encoding for analytical query processing. In *HPDC*, pages 1–12, 2013.
[24] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD*, 1996.
[25] A. P. Marathe and K. Salem. Query processing techniques for arrays. *The VLDB Journal*, 11(1):68–91, Aug. 2002.
[26] E. Ogasawara, D. Jonas, et al. Chiron: A parallel engine for algebraic scientific workflows. *Journal of Concurrency and Computation: Practice and Experience*, 25(16), 2013.
[27] P. E. O'Neil. Model 204 architecture and performance. In *HPTS*, pages 40–59, 1989.
[28] S. Perlmutter. Nobel Lecture: Measuring the acceleration of the cosmic expansion using supernovae. *Reviews of Modern Physics*, 84:1127–1149, July 2012.
[29] A. Shoshani and D. Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2009.
[30] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A storage manager for complex parallel array processing. In *ACM SIGMOD*, pages 253–264, 2011.
[31] M. Stonebraker, D. J. Abadi, A. Batkin, et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.
[32] Y. Su and G. Agrawal. Supporting user-defined subsetting and aggregation over parallel NetCDF datasets. In *IEEE/ACM CCGRID*, pages 212–219, 2012.
[33] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *IOPADS*, pages 23–32, 1999.
[34] A. R. van Ballegooij. RAM: A multidimensional array DBMS. In *EDBT*, pages 154–165, 2004.
[35] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A novel MapReduce-like framework for multiple scientific data formats. In *CCGRID*, pages 443–450, 2012.
[36] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over HDF5. In *CCGRID*, 2013.
[37] K. Wu. FastBit: An efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16:556–560, 2005.
[38] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31:1–38, 2006.
[39] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array data processing inside an RDBMS. In *ACM SIGMOD*, 2013.
[40] H. Zou, M. Slawinska, K. Schwan, et al. FlexQuery: An online in-situ query system for interactive remote visual data exploration at large scale. In *IEEE Cluster*, 2013.