

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-852637

IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems

I. Yildirim, H. Devarajan, A. Kougkas, X. H. Sun, K. Mohror

August 2, 2023

8TH INTERNATIONAL PARALLEL DATA SYSTEMS WORKSHOP
Denver, CO, United States
November 12, 2023 through November 12, 2023

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.



IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems

Izzet Yildirim
Illinois Institute of Technology
Chicago, IL, USA
iyildirim@hawk.iit.edu

Hariharan Devarajan
Lawrence Livermore National Laboratory
Livermore, CA, USA
hariharandev1@llnl.gov

Anthony Kougkas
Illinois Institute of Technology
Chicago, IL, USA
akougkas@iit.edu

Xian-He Sun
Illinois Institute of Technology
Chicago, IL, USA
sun@iit.edu

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, CA, USA
kathryn@llnl.gov

ABSTRACT

I/O analysis is an essential task for improving the performance of scientific applications on high-performance computing (HPC) systems. However, current analysis tools, which often use data drilling techniques (iterative exploration for deeper insights), treat every query independently and do not optimize column data for data-slicing (extracting specific data subsets), resulting in subpar querying performance. In this paper, we designed IOMax, a tool for efficient data drilling analysis on large-scale I/O traces. IOMax utilizes a novel query optimization technique to improve the query performance by 8.6x while reducing the memory footprint required for analysis by 11x. Additionally, it employs data transformation techniques to improve data-slicing performance by up to 11.4x. In conclusion, IOMax optimizes I/O analysis for scientific workflows on the Lassen supercomputer, resulting in up to 7x improvement.

KEYWORDS

HPC, I/O Performance, Data Drilling, Out-of-Core Analysis

ACM Reference Format:
Izzet Yildirim, Hariharan Devarajan, Anthony Kougkas, Xian-He Sun, and Kathryn Mohror. 2023. IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3624062.3624191

1 INTRODUCTION

With the increasing data-intensive nature of scientific workloads, optimizing I/O efficiency has become essential for maximizing scientific productivity on HPC systems. Despite the theoretical potential, a significant gap exists between the achievable and actual I/O performance of most scientific workloads on HPC systems [8, 13, 25]. As a result, tuning I/O performance has become a routine task for application developers on modern HPC systems.

One popular methodology for tuning I/O performance involves gathering I/O traces and examining patterns to detect anomalies [11, 12, 24]. Several tools currently exist to enhance the I/O performance of workloads by utilizing data drilling techniques to identify potential I/O bottlenecks [11, 16, 18], working with trace data that can fit in memory. Recently, traces from scientific workloads have reached terabytes in size, necessitating out-of-core I/O analysis (analysis of data too large to fit in memory by means of distributed computing) to understand their behavior, as demonstrated by Devarajan et al. [5]. As all the above work illustrates, data drilling techniques in I/O analysis are important in improving I/O performance for large-scale scientific workloads.

However, data drilling is a complex task that faces two major challenges. Firstly, data drilling often involves accessing the same data multiple times by drilling in or up the dataset. Existing analysis tools treat each query independently, resulting in poor performance due to the lack of global query optimizations [1]. Secondly, as part of the data drilling process, records within the dataset are often sliced and grouped based on dataset indices. Current analysis tools utilize the columns in the trace format for this purpose, which are efficient for trace production but inefficient for analysis purposes. For example, indexing on string columns like file names can be 11.4x slower than indexing on integer columns (Figure 6). These challenges highlight the need for an improved methodology to perform data drilling on large-scale I/O traces.

In this work, we present IOMax, a query optimization tool designed to enhance the efficiency of data drilling on large-scale I/O traces. IOMax incorporates three novel techniques to achieve this goal. First, the most important technique, is the query reduction that involves minimizing redundant I/O costs by intelligently constructing an execution plan and reducing the number of I/O operations needed to process a query. Second is the in-memory caching, where we decrease overall query time by storing the aggregate view of the dataset in distributed memory. Finally, IOMax analyzes the structure and format of the I/O traces and performs datatype conversions and encodings to enhance data drilling performance. The contributions of this work can be summarized as follows:

(1) **Design of IOMax**, a tool that improves data drilling analysis performance by up to 7x on large-scale I/O traces.
(2) **Designing a query engine** that builds a novel cache-aware query optimization algorithm to improve data drilling performance.
(3) **Developing a dataset validator and transformer** that detects and optimizes dataset inefficiencies for I/O analysis.

2 BACKGROUND & RELATED WORK

In this section, we provide an overview of data drilling, I/O analysis in HPC systems, and query optimizations in databases, highlighting the challenges and existing methods in these areas.



This work is licensed under a Creative Commons Attribution-NonCommercial International 4.0 License.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624191

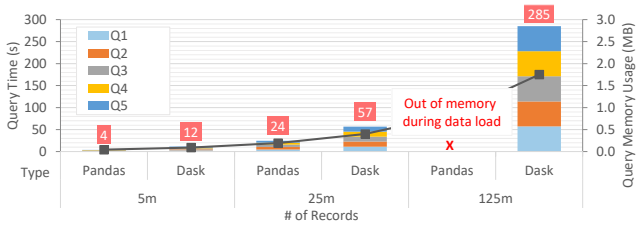


Figure 1: Performing out-of-core data drilling queries against datasets of different sizes in a memory-restricted environment demonstrates the need for a query and dataset optimization for distributed analysis.

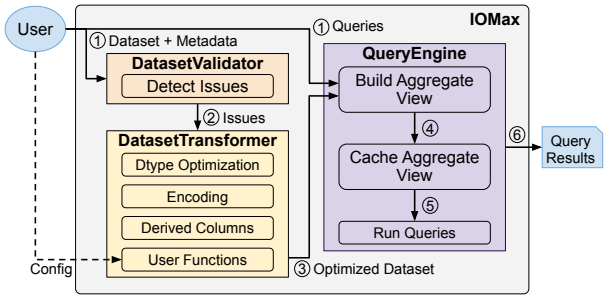


Figure 2: High-level design of IOMax showcasing the dataset optimization process and the query execution flow from user inputs to query results.

2.1 Data Drilling & Optimizations in Databases

Data drilling is a process of exploring data at different levels of detail through an iterative analysis. This iterative exploration helps uncover patterns, relationships, and anomalies that may not be apparent at a higher level. However, working with large datasets involves an inherent requirement for aggregation and join operations that necessitates multiple passes through the same data. To address this challenge, numerous methods have been proposed to optimize query execution plans in databases and data warehousing, all aimed at minimizing redundant work [1, 10, 23]. In our work, we build upon these existing methods by tailoring them to the unstructured data format of I/O traces. The primary reason why existing algorithms cannot be applied is that I/O analysis involves selection of specific columns to discover and gain insights. Traditional database systems designed to work with structured data are inefficient for this purpose.

2.2 I/O Analysis in HPC Systems

Currently, I/O analysis involves the use of multiple tools to examine the individual components of I/O systems. These tools include performance analysis, measurement, and visualization tools. Of those, Darshan [3] is one of the most widely used I/O profiling tool and a couple of companion tool was developed to analyze its traces, such as PyDarshan [18], DXT Explorer [2], and VaniDL [4]. Similarly, Recorder-viz [19] provides visualization for Recorder [22], an I/O tracing tool.

Studies such as UMAMI [12], TOKIO [11], and IOMiner [24] utilize aforementioned tools to detect I/O problems. Typically, such analysis is conducted through data drilling. It involves either running queries directly on the generated logs or manually iterating through them. However, both of these approaches have proven to be highly inefficient for data analysis (Figures 3-5). Although there is a study [5] that closely relates to our work; it lacks query, caching, and format optimizations, which we incorporate in our approach.

3 MOTIVATION

I/O analysis on large-scale I/O traces entails several design considerations that the existing tools may fall short in addressing. First, the tool should support out-of-core analysis, as the size of scientific workloads' I/O traces continues to grow, surpassing the available memory capacity. Second, it should minimize redundant I/O costs by reducing the number of I/O operations required for each query. Additionally, the tool should reduce the memory footprint of the analysis to ensure efficiency and scalability. Finally, it should optimize inefficiencies present in I/O traces, such as inefficient datatype usage and slow indexing.

Figure 1 illustrates the impact of limited memory resources on performing data drilling on large-scale I/O traces and the need for a query and dataset optimization. The figure shows the query times (y1-axis) and query memory usage (y2-axis) of five data drilling queries against different datasets sizes. In the figure, an in-memory analysis approach (Pandas) fails to load the largest dataset due to exceeding the available memory, underscoring the need for out-of-core analysis. Another notable observation is the similar query times and memory usage for individual queries, indicating the repetitive effort involved in treating each query independently. Therefore, there is a need for a solution that performs query optimizations for out-of-core data drilling analysis.

4 IOMAX

The IOMax tool is aimed at enhancing the efficiency of data drilling analysis on large-scale I/O traces. It is implemented as a Python library and specifically designed for I/O analysis purposes. While the methodology is generalizable, the tool currently specializes in enhancing I/O analysis. IOMax provides users with three main components: `QueryEngine`, `DatasetValidator`, and `DatasetTransformer`. The `QueryEngine` constructs an execution plan that leverages query reduction and in-memory caching techniques to minimize redundant I/O costs. The `DatasetValidator` identifies trace-format optimizations that can improve data-slicing performance on I/O traces. Lastly, the `DatasetTransformer` automatically applies the trace-format optimizations suggested by the `DatasetValidator`.

Figure 2 shows the high-level design of IOMax. The tool expects three main inputs from the user: `Dataset`, `Metadata`, and `Queries`. The `Dataset` is the raw I/O traces. The `Metadata` includes the indices and columns that need to be optimized. The `Queries` represent a set of queries for which the user wants to optimize. Initially, the `Dataset` and the `Metadata` are passed to the `DatasetValidator` to detect issues (①). If any issues are detected, both the `Dataset` and the detected issues are then passed to the `DatasetTransformer` (②). The `DatasetTransformer` performs several optimizations, including datatype correction, encoding, deriving columns, and applying user-defined optimizations. Next, the optimized `DataSet` and the `Queries` are forwarded to the `QueryEngine` (③), which in

turn, builds the aggregate view, caches it, and executes the query set against it (④-⑤). Finally, the user receives the results (⑥).

4.1 Query Engine

As discussed earlier, we argue that running queries directly on the generated logs or manually iterating through them are highly inefficient and unnecessary. IOMax introduces the `QueryEngine`, which is responsible of building a cache-aware query execution plan to address this problem. The creation of the execution plan comprises three steps:

4.1.1 Building Aggregate View. The aggregate view serves as an intermediary representation of the raw dataset, with which all the queries can be resolved. The algorithm to create the aggregate view is outlined in Algorithm 1. The input and output are described in the algorithm. Initially, we identify the keys (corresponding columns) in the query set (line 2). For example, the keys of the query in Algorithm 2, line 11, are `io_op` and `duration`. We then combine these keys with the indices provided to the algorithm (line 3). These indices are selected from the dataset columns and represent the resulting aggregate view's indices. Given our focus on I/O analysis in this work, typical indices include `duration`, `file_name`, and `proc_name`. Next, we determine the aggregation methods for each key in the combined set of keys (line 4). For instance, the I/O operation (`io_op`) column is aggregated using the `count` method, while the `duration` column is aggregated using the `sum` method. Finally, we group and aggregate the raw dataset using the determined aggregation methods and the combined set of keys to create the aggregate view (line 5). This approach ensures that the aggregate view includes only the necessary indices and columns, resulting in significantly faster query resolution and a substantial memory footprint reduction.

Algorithm 1: Building & Caching Aggregate View

Input: A dataset *DS*, a set of indices *IX*, a set of queries *QS*
Output: An aggregate view

1 **begin**
2 $QSKeys \leftarrow$ Find keys in the query set *QS*
3 $Keys \leftarrow$ Union *IX* and *QSKeys* into a single set
4 $AggM \leftarrow$ Determine aggregation method for each *Keys*
5 $AggView \leftarrow$ Group-aggregate *DS* by *Keys* using *AggM*
6 Cache *AggView* to improve performance
7 **return** *AggView*
8 **end**

4.1.2 Caching Aggregate View. When dealing with large datasets or complex analytical operations, cache optimization becomes crucial for optimizing query performance. One such complex operation is data drilling, which typically involves multiple iterations of queries, aggregations, or filtering on large datasets. By caching intermediate results, subsequent iterations can benefit from the cached data, avoiding redundant computations and reducing overall processing time. We utilize in-memory caching as our primary cache optimization technique, as it involves storing frequently accessed data in a fast and easily accessible location.

Column	Misinferred	Transformed	Reduction
I/O Category	32-bit Integer	8-bit Integer	~4x
Access Pattern	32-bit Integer	Boolean	~4x

Table 1: Memory footprint reduction for commonly misinferred datatypes.

Column	Original	Transformed	Reduction
File Name	String	64-bit Integer	10-20x
Process Name	String	64-bit Integer	10-20x
I/O Function	String	Category	60-80x

Table 2: Memory footprint reduction for string columns.

4.1.3 Query Resolving. Once the aggregate view is built and cached, the final step in the creation of the execution plan is the query-resolving process. This process involves two steps: First, we parse the queries to identify the requested columns and operations. For example, the requested columns in Algorithm 2, line 11, are `io_ops_per_sec`, `io_op` and `duration`; along with the operations `count` and `sum`. At this stage, it is unnecessary to validate the requested columns since we initially built the aggregate view with these columns in mind. Second, we create an execution plan which allows us to execute these operations in a single request, thereby avoiding redundant work and improving overall query performance.

4.2 Dataset Validator and Transformer

Raw I/O traces have a specific format that is not immediately suitable for analysis or data drilling. This is expected because the primary focus of I/O tracing tools is to capture and record events accurately rather than optimizing them for analysis purposes. However, when working with large-scale I/O traces, overlooking datatype inference can result in significant memory consumption penalties. For instance, if a categorical data column in an I/O trace is mistakenly inferred as a 32-bit integer, it would consume 4x more memory than necessary (Table 1).

IOMax introduces the `DatasetValidator` to detect issues related to datatype inference, binary encoding, and string encoding within the I/O traces. The validator compares the inferred datatypes of the columns with the expected datatypes and raises issues if any inconsistencies are found. In Table 1, you can see the original and transformed datatypes of commonly misinferred columns. Additionally, the validator verifies the encoding scheme used for strings and identifies malformed or incorrectly encoded strings. It also detects characters that may potentially cause issues during analysis.

Once the issues are identified, they are then passed to the `DatasetTransformer` along with the `Dataset`. The transformer corrects misinferred datatypes by converting them into the expected datatypes. Table 1 provides examples of commonly misinferred datatypes and the resulting memory reduction achieved by correcting them. The `Boolean` columns, such as the *Access Pattern* column in Table 1, are also binary-encoded for dimensionality reduction. In Table 2, we provide examples for string columns and show the observed memory reductions (in the I/O traces of the workflows that will be discussed in Section 5.1.2). Memory reduction for string columns depends on the number of unique values they contain, as their lengths can vary. When a string

column has a relatively small set of unique values compared to the total number of records, we use categorical encoding, as it results in the most significant memory reduction. In I/O traces, the *I/O Function* column typically has 10-20 unique function names, such as `open` and `close`, which makes it ideal for categorical encoding.

In addition, as we focus on I/O analysis, we give special attention to string columns such as *File Name* and *Process Name*. These columns inherently contain hierarchical information within each value. For example, the *File Name* column includes the complete file path, allowing extraction of file directories as well. In this study, we employ a custom hashing algorithm designed to process a string value, such as a file path, and calculate a unique hash value by considering specific components of that value. The resulting hash value serves as an identifier for the string value, allowing for efficient indexing, comparison, and sorting operations. These operations are instrumental in data-slicing.

4.3 APIs and Usage

In this section, we provide the description of the Python API of IOMax and its usage. At present, there are two main APIs that need to be discussed: the *Dataset Optimization API* and the *Query API*.

4.3.1 Dataset Optimization API. The Dataset Optimization API allows users to optimize their I/O traces for efficient analysis. Algorithm 2 demonstrates the usage of this API in lines 4 to 9. The process begins by importing the IOMax tool (line 2) and loading the dataset using the `load_traces()` function (line 4). Next, users determine the indices and the column types for optimization (line 6-7). The dataset is then optimized using the `ds.optimize()` method, with the indices and column types as inputs (line 9). This optimization step enhances the dataset's performance and efficiency for subsequent analysis tasks.

4.3.2 Query API. The Query API enables users to define and execute queries on the optimized dataset. Algorithm 2 demonstrates the usage of this API in lines 10 to 15. Users first define their queries using the illustrated query syntax (lines 11-12). These queries are stored in a dictionary for easy access (line 13). To execute the queries, the `ds.query()` method is utilized, taking the query dictionary as input (line 15). This triggers the execution of the queries and retrieves the results. In the provided algorithm, the result of a specific query (`q1`) is accessed using the query name as the key (line 17).

Algorithm 2: Usage of IOMax

```
1  # Import IOMax
2  import iomax
3  # Load I/O traces
4  ds = iomax.load_traces(log_dir)
5  # Determine indices & column types
6  indices = ["duration", "file_name", "proc_name"]
7  column_types = {"duration": float}
8  # Optimize dataset
9  ds.optimize(indices, column_types)
10 # Prepare queries
11 q1 = "io_ops_per_sec = count(io_op) / sum(duration)"
12 q2 = "metadata_ratio = count(metadata_op) / count(io_op)"
13 queries = dict(q1=q1, q2=q2)
14 # Execute queries
15 results = ds.query(queries)
16 # Read individual query results
17 result = results["q1"]
```

Workload	# of Records	# of Files	# of Processes
1000 Genomes	715,248,240	21,268,291	2,712
Montage	12,346,353	19,680	11,488
HACC	162,587	2,562	1,281
CM1	27,463	775	1,280

Table 3: The numbers of records, files, and processes of scientific workflows used in the evaluation.

5 EVALUATION

5.1 Setup and Software

5.1.1 Hardware. We run the experiments on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [7]. A 23-petaflop IBM Power9 system consists of 795 nodes connected with a Mellanox 100 Gb/s EDR InfiniBand network, and a 24 PiB IBM Spectrum Scale file system (also known as GPFS). Individual Lassen nodes consist of two IBM POWER9 CPUs (IBM AC922 servers) with 256GB of system memory.

5.1.2 Workloads. We utilize micro-benchmarks to demonstrate performance metrics related to the designed components. Furthermore, we incorporate four scientific HPC workflows: 1000 Genomes [21] (a workflow aids in identifying mutational overlaps to uncover potential disease-related mutations), Montage [9] (a mosaics-building tool extensively employed in astrophysics), HACC [6] (a cosmology workload), and CM1 [20] (an atmospheric-simulation workload). Table 3 shows the numbers of records, files, and processes of the workflows.

5.1.3 Tools. We use Pandas [14], most widely used Python data analysis library, and Dask [17], a Python parallel computing library for analytics, as our main analysis tools. We utilize Dask for our out-of-core analysis requirements, as it partitions large datasets into manageable chunks, enabling processing on available CPU cores or distributed clusters. One notable benefit is that the Dask `DataFrame` API is almost identical to the Pandas `DataFrame` API. This similarity allows us to perform data manipulations using familiar Pandas functions and syntax. As a result, we can execute identical queries using both Pandas and Dask, ensuring consistency in our analyses.

We use Recorder [22] as our tracing tool, as it provides a fine-grained format of I/O events, which is better suited for our query requirements compared to the aggregated statistical format of Darshan. However, since raw traces are not efficient for analysis purposes [5] and are incompatible with both Pandas and Dask, we first convert them into the Parquet file format [15], a column-oriented data file format designed for efficient data analysis. This conversion process is carried out by the `DatasetTransformer`, written in C, as part of our dataset optimizations.

Dataset	# of Records	File Size	Memory Size
5m	5 million	1.2 GB	2.4 GB
25m	25 million	5.9 GB	12 GB
125m	125 million	30 GB	60 GB

Table 4: The file and memory sizes of the datasets used in the Data Reduction evaluations.

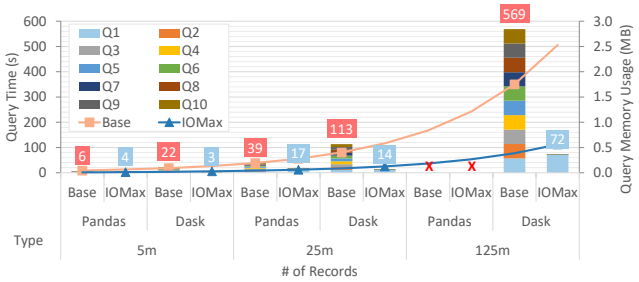


Figure 3: Comparison of query times and memory footprints between unoptimized (*Base*) and optimized (*IOMax*) query sets. Query times are stacked, while memory footprints are summed.

5.2 Data Reduction

To assess the effectiveness of our `QueryEngine` to address the main considerations mentioned in the motivation section, which are supporting out-of-core analysis, minimizing redundant I/O cost, and reducing analysis memory usage, we conducted an experiment using a micro-benchmark. In this study, we utilize Pandas' `groupby` and `agg` methods to achieve this. The `groupby` method groups the dataset using specific columns (in our case, keys), and the `agg` method allows us to specify a particular aggregation method for each key. This benchmark executes ten queries across various datasets with 5 million, 25 million, and 125 million records. The datasets were derived from real-world Recorder traces, and the queries were specifically designed to emulate real-world I/O analysis scenarios, such as determining I/O operations per second or transfer sizes within a specified time range as used by data drilling analysis [5, 11]. File and memory sizes of the datasets are shown in Table 4.

To accurately replicate the conditions of out-of-core analysis, we executed the benchmark on a compute node with a memory restriction of 20GB. The results are shown in Figure 3. The x-axis depicts different dataset scales for the unoptimized (*Base*) and optimized (*IOMax*) query sets, which were run using Pandas and Dask. In the optimized version, we create the aggregate view in addition to the initial query (*Q1*) to facilitate the execution of subsequent queries. On the other hand, the unoptimized version ran all queries sequentially, similar to existing toolsets. The y1-axis shows the total time taken for query execution in seconds. The y2-axis represents the total memory footprint of the query sets in gigabytes.

The findings indicate that the execution time of the unoptimized queries increases linearly, approximately 6x per dataset scale. Due to the memory restriction, the 125 million record dataset with a memory footprint of 60GB cannot fit in memory, and as expected, both the unoptimized and optimized query sets using Pandas fail. Dask manages to successfully complete the execution of an unoptimized query set by loading and processing data in smaller, manageable chunks that fit into memory. Additionally, we observe

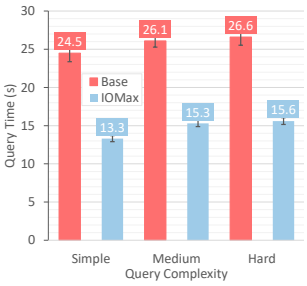


Figure 4: Cache-optimized query engine performs 1.7x faster.

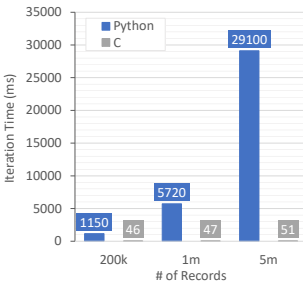


Figure 5: Python scales linearly, while C exhibits a constant iteration time.

that the optimized version has a memory footprint that is 8.5x smaller than the unoptimized version.

5.3 Cache Optimization

To assess the effectiveness of our cache optimization in the `QueryEngine`, we conducted an experiment using the I/O traces of the 1000 Genomes workflow. These traces contain more than 715 million I/O events from 3 thousand processes. During the experiment, we ran queries with different levels of complexity on both the unoptimized (*Base*) and cache-optimized (*IOMax*) query engines. In this study, the *Simple* queries are single aggregations over single columns (e.g. `sum(duration)`), the *Medium* queries are multiple aggregations over single columns (e.g. `sum(duration)` and `max(duration)`), and the *Hard* queries are multiple aggregations over multiple columns. It's important to note that the dataset index remains constant across all query complexities and is based on the *Process ID* (`proc_id`) column.

The results are presented in Figure 4, where the x-axis represents the query complexities and the y-axis represents the total time taken for query execution in seconds. The findings demonstrate that the cache-optimized query engine performs 1.7x faster than the unoptimized version. Additionally, the cache-optimized query engine exhibits greater consistency with a standard deviation of 400ms, in contrast to the unoptimized version which has a standard deviation of 1.13s.

5.4 Iterative Queries

Performing I/O analysis and data drilling typically involves iterative operations to manipulate data and identify patterns. One common task is determining whether files in an I/O trace are accessed sequentially or randomly. This involves tracking the offsets of `seek` operations and other related operations, usually done by iterating over the data. Python is commonly used for these iterative operations as they are part of the analysis process. In our design, the `DatasetTransformer` serves as a preprocessor responsible for handling such data manipulations. To evaluate the performance of both approaches, we conducted an experiment using a micro-benchmark. The benchmark involved iterating through various datasets, including 200 thousand, 1 million, and 5 million records. All datasets were created using the same approach outlined in Section 5.2.

The results are shown in Figure 5. The figure illustrates dataset scales on the x-axis and the corresponding total iteration time in milliseconds on the y-axis. The findings reveal a linear increase in Python's iteration time, approximately 5x per scale. This can be

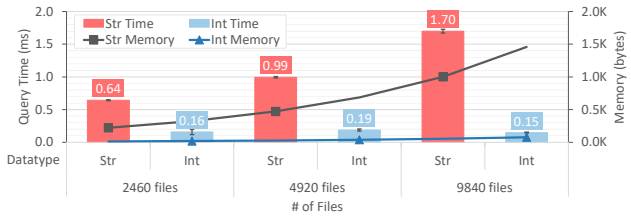


Figure 6: String indices, such as file names, slower and more memory-intensive than integer indices.

attributed to Python's inherent overhead of high-level abstractions and the limitation of performing vectorized operations when iterating over individual records. In contrast, C exhibits a constant iteration time, averaging around 50ms. Therefore, we conclude that for large-scale I/O traces, iterative operations should be avoided using Python and should instead be performed during the preprocessing stage.

5.5 Datatype Performance

To evaluate the effectiveness of our data transformation, we performed an experiment involving range queries on string (`Str`) and integer (`Int`) indices using the I/O traces from the Montage workflow. These traces contain over 12 million I/O events of 19680 unique files. The experiment aimed to identify the top 1/8, 1/4, and 1/2 of the most accessed files by executing range queries on two different indices. We leveraged the `loc` method of Pandas' `DataFrame` API to accomplish this. The `loc` method is to access a group of rows by labels. In our case, the labels are `file_names` for string indices and `file_ids` for integer indices. The `file_ids` are created using the hashing algorithm described in Section 4.2.

The experimental results are reported in Figure 6. In the figure, the x-axis shows the subset of files being accessed using both string and integer indices. The y1-axis shows the total time taken to locate the files in seconds. The y2-axis represents the total memory footprint of the indices in bytes. The results show that accessing subsets with string indices exhibits a linear increase in access time relative to the number of files, resulting in an 11.4x slower performance compared to integer indices. On the other hand, accessing the subset of files through integer indices shows a nearly constant access time, averaging around 15µs. This is mainly due to the efficient representation and memory access patterns of integer indices (and the cache optimization techniques employed by Pandas for integer indexing). Furthermore, string indices incur a memory footprint 13.1x larger than that of integer indices due to their variable-length nature.

5.6 Scientific Workflows

We evaluated the effectiveness of our methodology by conducting a similar evaluation to the Data Reduction evaluation described in Section 5.2. However, in this case, we used real-world scientific workflows as outlined in Table 3. Specifically, we utilized CM1 and HACC to showcase the in-memory performance of our tool, and 1000 Genomes and Montage to demonstrate its out-of-core performance. Prior to the evaluation, we optimized these datasets following the process described in Section 4.2.

The results are presented in Figures 7 and 8. As shown in the figures, IOMax significantly improves the performance of both

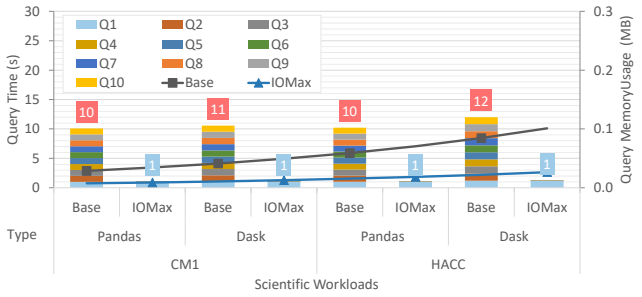


Figure 7: IOMax achieves up to 10x improvement in in-memory data drilling analysis of real-world scientific workflow I/O traces.

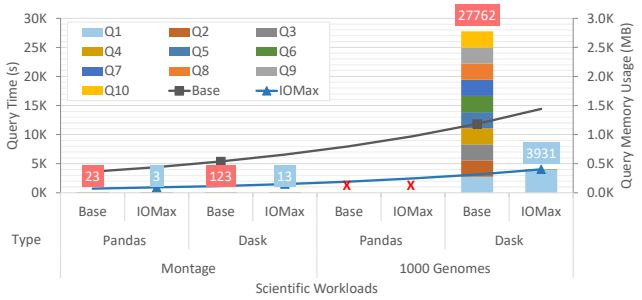


Figure 8: IOMax achieves up to 7x improvement in out-of-core data drilling analysis of real-world scientific workflow I/O traces.

in-memory and out-of-core analyses, achieving up to 10x and 7x improvements respectively. Furthermore, it reduces the memory footprint of queries by 5x and 3x respectively. The results validate IOMax's effectiveness in performance improvements and memory footprint reduction for real-world scientific workflows.

6 CONCLUSION

In this work, we addressed the limitations of current analysis tools, which often lack optimization for data-slicing and treat queries independently, by introducing IOMax, a tool that improves data drilling analysis performance on large-scale I/O traces. IOMax's novel query engine, cache-aware execution plan, and dataset optimizations achieved up to 8.6x performance improvement and an 11x reduction in memory usage. Furthermore, our transformer improves data-slicing performance by 11.4x. Testing on the Lassen supercomputer showcased the potential of IOMax, demonstrating up to 7x improvement in I/O analysis of real-world scientific workflows.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program (LLNL-CONF-852637). Also, the material is based upon work supported by the National Science Foundation under Grant no. NSF OAC-2104013, OCI-1835764, and CSR-1814872.

REFERENCES

[1] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. *In Proc. of Int. Conf. on Very Large Data Bases* (1995), 358–369.

[2] Jean Luca Bez, Houjun Tang, Bing Xie, David Williams-Young, Rob Latham, Rob Ross, Sarp Oral, and Suren Byna. 2021. I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*. IEEE, St. Louis, MO, USA, 15–22. https://doi.org/10.1109/PDSW54622.2021.00008

[3] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 Characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, New Orleans, LA, USA, 1–10. https://doi.org/10.1109/CLUSTR.2009.5289150

[4] Hariharan Devarajan. 2020. VaniDL. https://github.com/argonne-lcf/vanidl

[5] Hariharan Devarajan and Kathryn Mohror. 2022. Extracting and characterizing I/O behavior of HPC workloads. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Heidelberg, Germany, 243–255. https://doi.org/10.1109/CLUSTER51413.2022.00037

[6] Katrin Heitmann, Thomas D. Uram, Hal Finkel, Nicholas Frontiere, Salman Habib, Adrian Pope, Esteban Rangel, Joseph Hollowed, Danila Korytov, Patricia Larsen, Benjamin S. Allen, Kyle Chard, and Ian Foster. 2019. HACC Cosmological Simulations: First Data Release. *The Astrophysical Journal Supplement Series* 244, 1 (Sept. 2019), 17. https://doi.org/10.3847/1538-4365/ab3724

[7] HPC @ LLNL. 2023. Lassen. https://hpc.llnl.gov/hardware/compute-platforms/lassen

[8] Mihailo Isakov, Eliakin Del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. 2020. HPC I/O Throughput Bottleneck Analysis with Explainable Local Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–13. https://doi.org/10.1109/SC41405.2020.00037

[9] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John C. Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei Hui Su, Thomas A. Prince, and Roy Williams. 2009. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering* 4, 2 (2009), 73. https://doi.org/10.1504/IJCSE.2009.026999

[10] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *Comput. Surveys* 16, 2 (June 1984), 111–152. https://doi.org/10.1145/356924.356928

[11] Glenn K Lockwood, Nicholas J Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. 2018. TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis. *Proceedings of the 2018 Cray User Group* (2018).

[12] Glenn K. Lockwood, Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. 2017. UMAMI: a recipe for generating meaningful metrics through holistic I/O performance analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems - PDSW-DISCS '17*. ACM Press, Denver, Colorado, 55–60. https://doi.org/10.1145/3149393.3149395

[13] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Portland Oregon USA, 33–44. https://doi.org/10.1145/2749246.2749269

[14] Open-source. 2008. Pandas. https://pandas.pydata.org/

[15] Open-source. 2013. Apache Parquet. https://parquet.apache.org/

[16] Open-source. 2015. Darshan-util. https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html

[17] Open-source. 2015. Dask. https://www.dask.org/

[18] Open-source. 2020. PyDarshan. https://www.mcs.anl.gov/research/projects/darshan/docs/pydarshan/index.html

[19] Open-source. 2021. Recorder-viz. https://github.com/wangvsa/recorder-viz

[20] Hafizur Rahman, Michel M. Verstraete, and Bernard Pinty. 1993. Coupled surface-atmosphere reflectance (CSAR) model: 1. Model description and inversion on synthetic data. *Journal of Geophysical Research* 98, D11 (1993), 20779. https://doi.org/10.1029/93JD02071

[21] The 1000 Genomes Project Consortium. 2010. A map of human genome variation from population-scale sequencing. *Nature* 467, 7319 (Oct. 2010), 1061–1073. https://doi.org/10.1038/nature09534

[22] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New Orleans, LA, USA, 1–8. https://doi.org/10.1109/IPDPSW50202.2020.00176

[23] Min Wang and Bala Iyer. 1997. Efficient Roll-Up and Drill-Down Analysis in Relational Databases. *SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery* (1997), 39–43.

[24] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. 2018. IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Belfast, 466–476. https://doi.org/10.1109/CLUSTER.2018.00062

[25] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. 2012. Characterizing output bottlenecks in a supercomputer. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Salt Lake City, UT, 1–11. https://doi.org/10.1109/SC.2012.28