Implementation, evaluation and analysis of Block index for ADIOS

Tzuhsien Wu*, Jerry Chou*, Norbert Podhorszki‡, Yuan Tian‡, Junmin Gu†, and Kesheng Wu†

* National Tsing Hua University, Taiwan
† Lawrence Berkeley National Laboratory, USA
‡Oak Ridge National Laboratory, USA

***Abstract*—Scientific discoveries are increasingly relying on analysis of massive amounts of data. The ability to directly access the most relevant data records through query, without shifting through all of them becomes essential. However, scientific datasets are commonly stored on parallel file systems and IO systems that are optimized for reading/writing large chunks of data, and many scientific datasets have spatial-temporal data similarity, such that the records with similar values often locate in a close proximity of each other. Therefore, our previous work started to investigate the benefit of using block range index technique on HDF5 format files. In this work, we implemented our block index technique into the ADIOS I/O system by logically dividing the ADIOS storage blocks into smaller partitions, and recording the minmax values from each partitions as indexes. We studied how the data organizations and IO optimization techniques in ADIOS will impact the query performance of block index using a 3-dimensional S3D dataset. Comparing to the existing minmax method in ADIOS, which only records the minmax values per storage block, our evaluations showed that we can achieve up to 3x speedup in query time, and cause negligible overhead even in the worst case scenario. Therefore, our study suggests block index could be an effective technique for query, and its performance can be further improved for ADIOS.**

Keywords*—*Scientific data, Indexing, Query analysis, IO systems

I. Introduction

Scientific applications can easily produce massive amount of data in the order of terabytes, but only few data records contain important events or critical information for scientific discovery. Hence, instead of reading the entire datasets, indexing techniques are applied to accelerate query and reduce the time for retrieving data records, such as variants of B-tree [5] used in DBMS, bitmap indexes [18], and inverted indexes. While these indexing techniques can deliver efficient query performance to locate individual records, the computation complexity and storage requirement of these indexes is known to be expensive [11], [12].

On the other hand, scientific datasets are commonly stored and managed by parallel file systems and IO libraries, such as Lustre, HDF5, NetCDF and ADIOS [10]. To achieve greater scalability and higher throughput, all these systems are optimized for reading/writing large chunks of data block (i.e. a set of contiguous records). Hence, accessing and retrieving individual records cannot perform efficiently. Furthermore, recent studies [1] have also shown that data layout and file organization can also have significant impacts to query performance and data retrieval time. Therefore, the performance characteristics and behaviors of IO systems should be considered into the design of indexing methods as well.

Driven by the aforementioned problems, our previous work [19] has exploit the idea of applying block range index technique [8] for scientific datasets. Our implementation using HDF5 library on Lustre file system showed that the indexing technique can achieve comparable or better query performance than FastBit [18] and SciDB [6], but only need much smaller index size and index build time. In this work, we want to further investigate the technique with other I/O systems. Therefore, we implemented the block index technique into the ADIOS I/O system [10], and demonstrated the query performance improvement from using our technique in the I/O system. We chose ADIOS for the integration and evaluation of our method for three important reasons. Firstly, ADIOS is a state of the art componentization of the IO system, and it has been used by several scientific simulations [13], including GTC, GTS, S3D, etc. It provides users the flexibility to switch between different IO implementations through a XML configuration file. Therefore, through this work, we can make the block index technique become one of the query optimization options for the community in the near future. Secondly, ADIOS has implemented a minmax method that simply records the min, max value from each writeblock (i.e., more details in Section II). However, the size of writeblock is determined by the size of data of each process when the file is created, and can be extremely big ($\sim$ 2 to 5 GB) for optimizing contiguous data read/write. Therefore, we want to investigate how much query performance still can be gained from using our method comparing to the existing implementation. Finally, ADIOS is a I/O system with the capability of supporting in-situ processing and analysis. Therefore, we believe our index technique is well suited for the system, and this work allows us to evaluate our technique for the in-situ applications in the future.

Our contributions are summarized as below:

- Implement block index method into the ADIOS I/O system, and evaluate our implementation on supercomputers.
- Compare our index technique to the existing minmax method in ADIOS, and show our technique can achieve achieves up to 3x speedup in query time.
- Analyze the data characteristics of several variables from a real multi-dimensional S3D dataset. Our finding confirms that scientific datasets, like S3D, are

suitable for using block index technique because they often exhibit data distribution patterns where interesting records locate in closer proximities.

- Identify a couple of performance problems that can be further optimized in our implementation. One of them is to balance the I/O requests among readers. The other is to improve IO throughput by merging blocks in close proximity.

The rest of paper is structured as follows. Section II briefly introduces ADIOS. Section III describes the implementation of our index technique in ADIOS. Section IV presents our experimental results, and Section V is the related work. Finally, the paper is concluded in Section VI.

II. ADIOS

ADIOS(Adaptive IO system) is a state of the art componentization of the IO system. It takes the implementation of the IO layer away from the application scientist, and provides users the flexibility to switch between different IO implementations through a simple XML configuration file. Due to its simplicity and portability, it has been used by several scientific simulation codes, and many IO optimization modules have been developed for ADIOS.

In ADIOS, a global array is consisted of many individual, contiguous blocks written out by many writers. These blocks are called the *writeblock*, and its size is determined by the size of data of each process when the file is created. Therefore, the size of writeblock can be extremely big (2 to 5 GB), and it is optimized for reading/writing large chunks of data block.

Besides having various read/write methods, ADIOS only supports query API as an alternated way of creating the selection of data reads. A query is an AND/OR tree of simple variable-relation-value expressions, like "$x > 1.2$ and $y < 10$". The API returns a list of points(i.e., records) that satisfy the expression as the result of a query evaluation. The list of points can then be directly used in the read functions.

Because reading a list of points that scattered across dataset one-by-one can easily cause significantly lower IO bandwidth. Therefore, if a set of points needs to be read out from the same writeblock, ADIOS will determine a contiguous bounding box (i.e. compact ranges in all dimensions of a variable) that includes all the points within the writeblock, and read out the bounding box from file by a single IO request.

ADIOS also implemented a minmax method on writeblock to accelerate query evaluation(i.e., but it has not been included into the latest public release). The minmax method simply records the maximum and minimum values from a writeblock when the variable is created. The metadata is stored along with data file, and is loaded into memory once the data file is opened. During query evaluation, minmax method examines the range of each writeblock based on the metadata in the memory, and the IDs of writeblocks which contain hit records are returned to user. To get the individual selected records from each block, user will need to read the entire data block based on the block ID, and then filter the results by evaluating the query constraints. Although this method needs to read more data, and adds an additional post-processing filtering step, its overall query time can still be faster than the traditional approach which evaluates query directly, and then retrieves the selected data individually, like we have shown in our previous study [19].

III. IMPLEMENTATION

As mentioned in the previous section, the minmax method is tightly coupled with the writeblock data structure in ADIOS. But writeblock is a static setting of block size that is determined by the amount of data written from process during file creation, and optimized for large chunk sequential IO. However, data access through query selection creates small random scattered IO. As a result, minmax method ends up retrieving too many data records which causes longer query time. To address the issue, our block index implementation extends the minmax method as follows.

For index procedure, we logically divide a writeblock into smaller fixed-size partitions, and records the minmax values of each partition. It is noted that creating logical partitions on writeblocks is different from simply using a smaller writeblocks size setting. This because using a smaller writeblocks size will create more physical writeblocks, and the IO throughput will drop significantly when more write blocks are read from storage systems. On the other hand, using logical partition can maintain the same number of writeblocks on storage systems, and the IO requests on the same writeblock can be merged by ADIOS to minimize IO contentions.

In our current implementation, the indexes are stored in a separate index file. Since the index file is relatively small comparing to the amount of data that needs to be retrieved from query selection, loading the index file has limited impact in our query performance evaluation. But in the future, we will also implement it as the metadata (a list of minmax values) on writeblocks, so that the indexes can be directly loaded from memory as well.

For query procedure, we perform domain decomposition to the dataset, and distribute the corresponding partitions to processes for query evaluation in parallel. Each process independently loads the minmax values of its partitions from the index file, and identifies the partitions with hits by comparing to the range of query constraints. A partition is the same as a bounding box data selection in ADIOS, and it can be specified by a pair of coordinates of opposite corners of the bounding box. If contiguous partitions are selected, we can directly merge them into a single bounding box. Hence, at the end, our query API returns a list of coordinates that describes the list of partitions with hits. Then the returned coordinates can be used by processes independently to retrieve the data records from their own subset of data using the ADIOS read API.

As shown by our evaluation study, creating block index with a size smaller than the data storage size, such as writeblocks, can provide several benefits. (1) The amount of retrieving data is reduced, and so is the query time. (2) If the block size is too small and cause inefficient IO, logical blocks can always be merged together into a larger data chunk for reading. In other words, the data read size can be dynamically adjusted according to the query selection pattern. (3) Having selected data scattered in more data partitions allows us to take advantage of higher IO parallelism in reads, and balanced the

load among processes more evenly. Therefore, throughout our implementation and evaluation, we analyze the performance impact of these design decisions, and show the performance of ADIOS minmax method can be improved substantially.

IV. Experimental Evaluations

A. Experimental setup

We conducted our experiments on a large super computing system called Edison at NERSC. We evaluated the query performance of our implementation using a S3D dataset. The dataset was generated from a turbulent combustion simulation, and written into files through ADIOS. The dataset contains three variables: "*temp*", "*wvel*", and "*pressure*". Each variable is a three-dimensional double precision array with the dimension length of $1100 \times 1080 \times 1408$. The variables were written to files using a writeblock size of $275 \times 270 \times 352$ (i.e. roughly 200 MB per writeblock). That means a variable is split on all dimensions, and stored into 64 writeblocks. The data in a writeblock is then serialized along the higher dimension first, and stored on disks. Lustre is the parallel file system for the supercomputer, and the file is stored using the maximum stripe count 36 and default stripe size 1MB.

In the experiments, we compare our implementation to the minmax method in ADIOS by reporting the query response time. For both methods, the query response time includes the time of scanning the index and retrieving the data blocks for query evaluation, so the values of selected records are returned to the applications. In this paper, we only show the query evaluation results of "*temp*" in Section IV-C∼Section IV-C due to page limits. But other variables also had similar results.

B. Data locality of S3D dataset

Data locality can significantly affect the performance of block index. For a dataset with good locality, data records with similar values are also in close locations in the file. If the locations of selected records are closer, a partition can contain more selected records, so block index can retrieve them in a single read without

First of all, we want to show that most of the real scientific datasets, including S3D, is suitable of using block index because they have stronger data locality, which means the query selected data often locates in a close proximity and can be retrieved together efficiently without accessing too much redundant records. In the experiments, we evaluate data locality by collecting the distances (i.e., in terms of number of records) between every two neighboring hit records according to their serialization order on storage devices. Then we plot the cumulative distribution function of these distances in Figure 1 from different variable and query selectivity. As shown, distances between more than 99.9% of selected records are just 1, which means those hits are contiguously located in the dataset and can be retrieved together efficiently. However, these records may not align to the boundary of a writeblock. Therefore, as shown by our evaluation results in the rest of the section, block index can still reduce the amount of retrieval data significantly. We also observed that the selected records become more scattered when the query selectivity increases. Therefore, in Section IV-D, we found block index can take advantage of locality of S3D dataset, and the performance

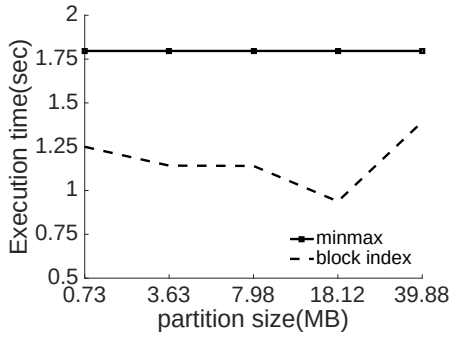


Fig. 2. Execution time of block index and minmax under different partition size settings.

TABLE I. IO statistics from varied partition size settings

Partition size	read requests	bytes read	I/O throughput
0.73MB	1298	941.17MB	753.76MB/s
3.63MB	266	964.38MB	852.84MB/s
7.98MB	124	989.03MB	867.29MB/s
18.12MB	59	1069.52MB	1141.22MB/s
39.88MB	30	1196.41MB	864.65MB/s
minmax	11	2193.42MB	1222.17MB/s

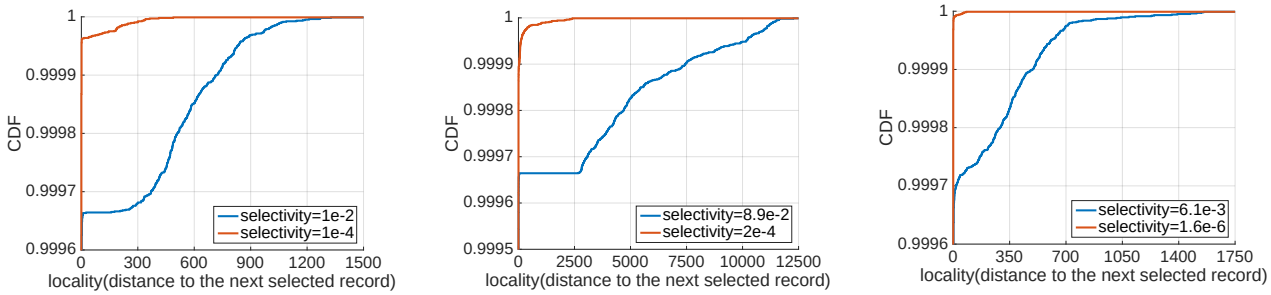
improvement is more obvious when the query selectivity is smaller.

C. Performance under varied partition size

Here, we investigate how the size of partition can affect the performance of block index. Figure 2 shows the execution time of block index comparing to minmax under different partition size settings. We conduct the experiment using "temp" variable with 1e-2 query selectivity. As shown by the figure, the best partition size setting in this case is 18.12MB which gives us the minimal execution time. To understand the reason for achieving the best performance, we analyze the IO statistics as shown in Table I. From the IO measurements, we can observe the performance is a result of tradeoff between the read size and IO throughput. When the partition size is increased, both the number of read requests and the bytes read can be reduced for saving data retrieving time. But at the same time, the IO throughput is lower when the partition size is smaller due to the constant latency overhead from each request. In comparing, minmax method has the highest IO throughput, but its read bytes is more than twice the block index. Therefore, overall, block index is still faster than minmax method.

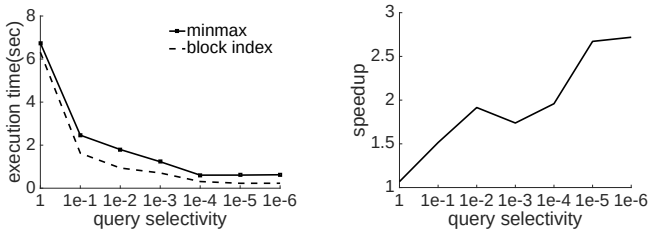
D. Performance under varied query selectivity

Figure 3(a) compares the query performance of block index and minmax under different query selectivity settings using the "temp" variable. The partition size of block index is 18.3MB, which is the best setting as suggested by the analysis results in Section IV-C. We can see that the execution time of block index is less than or similar to minmax under all cases, and the speedup improves from 1.07x to 2.8x as the query selectivity decreases. As observed from IO statistics summarized in Table II, we can see the performance of block index is due to less amount of read bytes than minmax method. When the query selectivity is 1E-2, minmax reads over 2GB of data, while block index only reads half of it around 1GB of data. But, when the query selectivity is reduced to 1E-6, block



(a) "temp" variable. (b) "wvel" variable. (c) "pressure" variable.

Fig. 1. CDF of data locality of S3D dataset under different query selectivity. Data locality represents the distance to the next selected record.



(a) Execution time. (b) speedup of block index.

Fig. 3. Execution time and speedup of block index and minmax under different query selectivity. Speedup is calculated as the execution time of minmax divided by the execution time of block index.

index reads less than 1/5 of the data of minmax method. This is due to the following two reasons.

First reason is that the minimum read bytes of minmax is bounded by the size of writeblock which is around 200MB, that is almost 10 times larger than the partition size (18.3MB) used by the block index in this experiment. So block index still can read less data when the query selectivity is smaller than 1E-4. The second reason is as shown by the data locality study in Section IV-B, the selected records are more scattered when the query selectivity is higher. Therefore, block index needs to retrieve more partitions that include more redundant data.

We also found that even in the worst case when all the data must be retrieve under 100% query selectivity, block index still can have a similar performance as the minmax method with negligible overhead. As shown in Table II, the number of read requests from block index is more than 10 times to the minmax method, but surprisingly this didn't cause significant performance impact to the block index method. This is because writeblocks are only logically partitioned by our block index method, not physically. As mentioned in Section II, ADIOS still can perform IO optimization internally, such as buffering, to reduce IO contentions. As a result, block index did not suffer from the increasing number of IO requests. To verify our observation, we did an experiment to test the query performance by setting the writeblock size to 18.3MB, and found the data retrieval time becomes 2∼5 times slower. Therefore, using a smaller writeblock size is different from building a block index with smaller partition size in ADIOS.

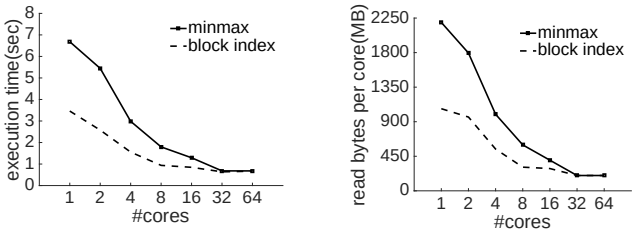
TABLE II. IO STATISTICS FROM VARIED QUERY SELECTIVITY SETTINGS

block index			
query selectivity	read requests	bytes read	I/O throughput
1	704	12761.72MB	2021.24MB/s
1E-1	70	1268.92MB	781.62MB/s
1E-2	59	1069.52MB	1141.22MB/s
1E-3	25	453.19MB	641.08MB/s
1E-4	5	90.64MB	292.99MB/s
1E-5	3	54.38MB	240.92MB/s
1E-6	2	36.25MB	168.83MB/s
minmax			
query selectivity	read requests	bytes read	I/O throughput
1	64	12761.72MB	1892.07MB/s
1E-1	12	2392.82MB	971.48MB/s
1E-2	11	2193.42MB	1222.17MB/s
1E-3	4	797.61MB	648.61MB/s
1E-4	1	199.40MB	329.10MB/s
1E-5	1	199.40MB	327.55MB/s
1E-6	1	199.40MB	318.97MB/s

E. Scalability & Load balancing issue

Finally, we evaluate the performance impact of using varied number of cores for query processing. We conduct the experiment using "temp" variable with partition size of 18.3MB for the block index, and the query selectivity of 1E-2. As expected, Figure 4 (a) shows the query execution time for both methods reduces as more cores are given. However, we found the improvement of block index is dimmish as the scale increases. After further investigation, we found the reason is because the load was not balanced among processes. This is because in the current ADIOS implementation, there is a fixed assignment between the writeblocks and the processes. The data from a writeblock can only be read from its assigned process. As a result, when the query selected data scattered unevenly among writeblocks, some processes might read more data than the others and become the performance bottleneck.

As shown in Figure 4 (b), the read bytes of the process with the highest load didn't decrease proportionally to the number of cores. That implies the load didn't evenly balanced among processes when more cores are added. This is more likely to happen in scientific datasets, because of the data locality behavior we observed in Section IV-B. Due to this reason, although the total read bytes of block index is always just half of the minmax method, the maximum read bytes of a single process eventually became the same and so did the query performance. This results indicate load balancing is crucial to the query performance of block index. Since not all the applications using ADIOS require a fixed assignment between



(a) Execution time. (b) Maximum read bytes per core.

Fig. 4. Execution time and maximum read bytes per core of block index comparing to minmax.

writeblocks and processes, we suggest to redistribute the data that needs to be retrieved among processes for achieving better performance for both methods. We also expect this load balancing method can benefit the block index more than minmax method. This is because the size of a block partition is much smaller than the size of a writeblock, and therefore the load can be balanced more evenly.

V. Related Work

A variety of indexing techniques are available in popular database systems [16], many of which are variations of the B-Tree [5]. The B-Tree data structure is designed to update quickly as the underlying data records are modified. But scientific data is mostly accessed in read-only manner, and thus bitmap index is a more appropriate indexing structure [15], [17]. A number of different strategies have been proposed to reduce the sizes of bitmap indexes and improve their overall effectiveness. A state-of-art bitmap index technique is FastBit [18], and its parallel implementation FastQuery [4] has shown the capability of indexing and analyzing of trillion particle datasets [2]. However, building these complex data structures is time consuming and the size of indexes is too large to fit in memory. Therefore, these indexing techniques usually happens at data load steps of database systems [1], and the indexes are stored in files for repeated usage.

Recently, there were several attempts that design new indexing technique to reduce the index size based on data compression and reorganization. ISABELA [12] adapts B-splines curve fitting to compress data, and guarantees a user-specified point-by-point compression error bound by storing the relative errors between estimated and actual values. DIRAQ [11] exploits the redundancy of significant bits in the floating-point encoding among similar values, and uses a compressed inverted index to reduce index size. However, data query is only a part of a data analysis workflow. Hence reorganizing and encoded data may have severe impacts to other processing steps.

Finally, our implementation of the block index is a variation of the block range index(BRIN) technique proposed by Herrera in 2013 [8]. This technique is intended to enable very fast scanning of extremely large tables. Since the implementation of this technique is tightly coupled to the underlying storage and IO systems, only ProstgreSQL has announced this feature in their products [8]. Other vendors have only described similar features, such as the storage index of Oracle [14] and Hive [9], the zone maps of Netezza [7]. Other recent studies [19], [3] have shown such technique is suitable for scientific datasets, because of the data locality from the spatial-temporal data similarity in these datasets.

VI. Conclusions

In this work, we implemented the block index technique into the ADIOS I/O system, and evaluated its query performance on a 3-dimensional S3D dataset. From our study, we have the following findings. First, we observed the query performance of minmax method in ADIOS is limited by the size of writeblock which is optimized for contiguous data read/write not for random query access. By building a block index that logically partitions a writeblock, we showed that the query time can be improved due to less data reading, and more flexible read size. Second, we analyzed the query performance under various settings, including query selectivity, computing scales, and block size. The results showed that block index can consistently deliver better performance, and the improvement is even greater under smaller query selectivity. Finally, we found scientific datasets, like S3D, often exhibit stronger data locality, such that interesting records locate in a closer proximity to each other. Hence, using block index technique can effectively reduce the amount of retrieval data in query.

In the future, we would like to further improve the query performance in ADIOS by balancing the data retrieval workload across readers. Also we will conduct more performance analysis and modeling of IO systems, so that we can design the algorithms to decide the proper block size and request merging condition. Finally, we will implement the indexing steps into the ADIOS system as well, and evaluate the overhead and benefit of our technique for in-situ processing and data analytic.

References

[1] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *ACM SIGMOD*, SIGMOD '14, pages 385–396, 2014.

[2] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC*, page 59, 2012.

[3] C. Chen, X. Huang, H. Fu, and G. Yang. The chunk-locality index: An efficient query method for climate datasets. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2104–2110, May 2012.

[4] J. Chou, K. Wu, O. Rübel, M. Howison, J. Qiang, Prabhat, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani. Parallel index and query for large scale data analysis. In *SC*, 2011.

[5] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[6] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-oriented DBMS. *Proc. VLDB Endow.*, 2(2):1534–1537, Aug. 2009.

[7] G. S. Davidson, K. W. Boyack, R. A. Zacharski, S. C. Helmerich, and J. R. Cowie. Data-centric computing with the netezza architecture. Technical Report SAND2006-3640, Sandia National Laboratory, 2006.

[8] A. Herrera. Minmax indexes. pg hackers.

[9] Apache hive. https://hive.apache.org/.

[10] ADIOS. http://www.nccs.gov/user-support/center-projects/adios/.

[11] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova. Scalable in situ scientific data encoding for analytical query processing. In *HPDC*, pages 1–12, 2013.

[12] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data. In *Euro-Par*, pages 366–379, 2011.

[13] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[14] A. Nanda. Smart scans meet storage indexes. *Oracle Magazine*, 2011.

[15] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59. Springer-Verlag, Sept. 1987.

[16] P. O'Neil and E. O'Neil. *Database: principles, programming, and performance*. Morgan Kaugmann, 2nd edition, 2000.

[17] A. Shoshani and D. Rotem, editors. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2010.

[18] K. Wu, S. Ahern, et al. FastBit: Interactively searching massive data. In *SciDAC*, 2009.

[19] T.-H. Wu, H. Shyng, J. Chou, B. Dong, and K. Wu. Indexing blocks to reduce space and time requirements for searching large data files. In *CCGrid*, pages 398–402, 2016.