Towards an Empirical Evaluation of Scientific Data Indexing and Querying

Thaylon Guedes[1], Vítor Silva[2], José Camata[4], Marcos V. N. Bedo[3], Marta Mattoso[2], Daniel De Oliveira[1]

[1] Computing Institute, Universidade Federal Fluminense, Brazil
thaylongs@id.uff.br, danielcmo@ic.uff.br
[2] Computer Science Department, Universidade Federal do Rio de Janeiro, Brazil
{silva, marta}@cos.ufrj.br
[3] Fluminense Northwest Institute, Universidade Federal Fluminense, Brazil
marcosbedo@id.uff.br
[4] Computer Science Department, Universidade Federal de Juiz de Fora, Brazil
camata@nacad.ufrj.br

Abstract. Computational simulations usually produce large amounts of data on a regular time-step basis. Heterogeneous simulation outputs are stored in different file formats and on distinct storage devices. Therefore, the main challenges for accessing simulation data are related to *time-to-query*, which is the effort spent for setting all data into a common framework, the issuing of a high-level query statement, and obtaining the result set. The simulation data loading into DataBase Management Systems (DBMS) are either unpractical, as they demand a prohibitive time for data preparation, or unfeasible, as data files are still needed in their original form (scientific applications still need to read and write contents to those files). In this article, we discuss the complementary approaches of *adaptive querying* and *raw data file indexing* for accessing simulation results stored in multiple sources (*e.g.*, raw data files) without data loading. In particular, we review (i) NoDB PostgresRAW routines for adaptive query processing, and (ii) FastBit methods for raw data file indexing and querying. We examine the behavior of both strategies regarding a real case study of computational fluid dynamics simulation in the domain of sediment deposition. In this experimental evaluation, we measured the elapsed time for index construction and query processing regarding six distinct query categories over 62 time steps, which sums up to different 372 queries on 44,160 files (12.2 GB) produced by the computational simulation. Results show that FastBit is faster than PostgresRAW for query execution in all but low-selectivity query scenarios. In a complementary manner, results also show PostgresRAW outperforms FastBit whenever users are interested in reducing time-to-query rather than the query execution time itself.

Categories and Subject Descriptors: H.2 [**Database Management**]: Systems; H.2 [**Database Management**]: Database Applications; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

Keywords: computational fluid dynamics, dataflow management, scientific data indexing, scientific data querying

1. INTRODUCTION

The volume of data consumed and produced by simulation programs is increasing at a very fast pace due to the several advances in computational simulations [Wang and Zhai 2016]. Such data represent the intermediate and final results produced by complex simulations and are usually stored in files with different formats depending on the domain (*e.g.*, `XDMF` and `HDF5`) [Clarke and Mark 2007]. In this context, and for the sake of simplicity, we call *scientific data* the data produced by scientific simulations and represented in their raw format. The exploration and analysis of scientific data requires accessing the content of raw data files, extracting regions of interest in files, loading them into an external database, and indexing all contents before high-level queries can be efficiently performed.

Most of the existing approaches that aim at querying and analyzing scientific big data stores information in its original form, *e.g.*, semi- and non-structured file formats [Blanas et al. 2014]. Such approaches access and extract domain-specific data from files employing lexical and syntactical analyzers. More specifically, users define the attributes of interest so that their corresponding domain data are extracted whenever they are present in the experiment files. Scientific data usually relies on indexing methods for (i) reducing data redundancy and the overhead of data loading, and (ii) speeding up the execution of queries on (parts of) inspected data. Remarkable examples of scientific data management systems are DBMS-driven solutions like SciDB [Cudre-Mauroux et al. 2009] and PostgresRAW [Alagiannis et al. 2012a; 2012b], and file-driven tools as FastBit [Wu 2005] and FastQuery [Chou et al. 2011]. Such solutions are designed for handling isolated raw files, *i.e.*, one file has no relationship with others. Consequently, they rely on high-level SQL statements for providing support for simulations that produce a series of files related to each other.

In this article, we discuss existing DBMS-driven and file-driven solutions regarding their characteristics for the analysis of simulation results stored into multiple data sources, *e.g.*, raw data files. Such solutions were divided into two approaches: *adaptive querying*, which improves the query performance by using on-line self-tuning whenever neither statistics nor predictable costs are available [Deshpande et al. 2007], and *raw data file indexing*, which optimizes data accesses in raw files through indexes. In particular, we review (i) PostgresRAW routines for adaptive query processing, and (ii) FastBit for file indexing and querying. We examine the behavior of both strategies regarding a case study of simulation in the domain of Computational Fluid Dynamics (CFD) for sediment deposition. In this experimental evaluation, we measured the time spent for index construction and query processing regarding six distinct query categories over 62 time steps, which sums up to different 372 queries on 44,160 files with 12.2 GB coming out of the computational simulation. Therefore, our experimental evaluation contributes with the following main results:

(1) FastBit is faster than PostgresRAW for query processing in all but low-selectivity queries, considering that indexes were previously constructed; and
(2) PostgresRAW is more responsive to first submitted queries because it demands a much lower *time-to-query* (*i.e.,* elapsed time to generate the index and perform the query) in comparison to competitor FastBit.

This article is an extension of work originally reported in the Proceedings of the Brazilian Symposium on Databases [Guedes et al. 2017] held in Uberlândia, MG - Brazil, on October 2017. The remainder of this article is structured as follows. Section 2 presents existing approaches for scientific data indexing. Section 3 discusses the case study based on a CFD simulation as well as the volume and the nature of the manipulated data. Section 4 presents the empirical evaluation performed on the examined techniques, considering the simulation on the CFD domain for analyzing sedimentation deposits. Finally, Section 5 discusses the results and concludes the paper.

2. BACKGROUND AND RELATED WORK

Analyzing scientific data can be carried out with and without data loading [Blanas et al. 2014]. In the first case, *file indexes* upon loaded data are viable options for search optimization. Several paradigms can be used for indexing and querying scientific data, but two of them stand out as the most versatile strategies regarding performance and data loading adaptability: *(i) bitmap* and *(ii) positional* indexing [Alagiannis et al. 2012a]. Bitmap indexing techniques rely on the discretization of active domains of scientific data attributes in such a way *boolean* comparisons can be used for the matching of entries that satisfy the query criteria. For instance, if the active domain of a given attribute X from a scientific data file is composed of five distinct values, then *bitmap* based on identity criteria requires five columns for pairing any identity-based comparison to the data attribute.

Additionally, bitmaps are useful for order-based comparisons whenever the active domain is large enough. FastBit [Wu 2005] and FastQuery [Chou et al. 2011] are examples of scientific data management systems that apply bitmap indexing techniques. In particular, FastBit enables the parameterization of the index structure itself regarding the codification type, the compression algorithm, and the discrete bitmap precision.

Analogously, positional indexing relies on the dynamic construction of a map of entries to the physical location of the attributes in the raw data files [Alagiannis et al. 2012a]. Positional indexes require less memory space in comparison to bitmaps as they do not employ redundant entries (as in the case of 0's values in bitmaps). Moreover, positional indexes can point to more complex structures than single raw data files, such as trees or meshes. Positional indexing can be used alongside with attribute extraction for *in situ* data analysis - a query strategy known as NoDB philosophy [Karpathiotakis et al. 2014]. PostgresRAW is an example of a self-tuning NoDB engine that extracts the active attribute domain to a modified PostgreSQL layer and enables the use of standard SQL for accessing domain data (*i.e.,* the content of data files produced by the simulation). PostgresRaw avoids modifying file structures and heavily relies on adaptive query processing, which demands the use of statistics and *caching* mechanisms for accessing, extracting, and indexing the contents of data sources. Accordingly, only the domain attributes retrieved by user-provided queries are effectively indexed, whereas the information itself is kept into files with different structures. Table I presents a comparison between FastBit and PostgresRAW solutions regarding their indexing techniques, high-level query processing support, scientific data loading requirements, and time complexity for fetching a raw data file.

Table I: Comparison of PostgresRAW and FastBit regarding their main features.

Approach	**Strategy**	**Indexing**	**Language**	**Data Loading**	**File Data Access**
PostgresRAW	Adaptive querying	Positional	Standard SQL	Not required	Linear
FastBit	Raw data file indexing	Bitmap	SQL Compatible[1]	Required	Constant

We highlight both bitmap and positional indexing approaches are designed for *isolated* files, *i.e.*, they do not enable the efficient query processing of *data element flows* [Silva et al. 2017]. Therefore, the straightforward use of such methods for querying scientific data whose output is a *flow* rather than a single *isolated file* would demand rewriting the simulation programs, or the complete modification of their data structures layer to be stored into DBMS, which is unviable in most of the cases. As an alternative, data can be kept in their original format and replicated to scientific data management and data analysis tools, which must be somehow coupled with FastBit or PostgresRaw, for instance.

In the next section, we discuss the use of the two prominent yet competing solutions, PostgresRaw and FastBit, for the query and analysis of results of a real turbidity currents simulation [Camata et al. 2018], which is kept as a series of related files stored as multiple raw data files. Accordingly, we evaluate queries that associate the contents of multiple files as scientific data are stored in several files. Our goal is to perform an empirical assessment of the capabilities of the two aforementioned solutions in a real-world scenario.

3. CASE STUDY: COMPUTATIONAL FLUID DYNAMICS

Our case study focuses on simulations in the CFD field regarding the specific problem of analyzing sediment deposition over sedimentary basins. Such case study is particularly suitable for the evaluation of adaptive query processing *versus* raw data file indexing as it involves the generation of a large volume of scientific data stored in specific file formats of the domain (`XDMF` and `HDF5`). Moreover, users need to access the contents of these files to support specific domain data analyses, such as the validation of scientific hypotheses. We ran the experiments by using the libMesh-sedimentation solver [Camata

[1]`https://sdm.lbl.gov/fastbit/doc/ibisCommandLine.html`

et al. 2018]. This application is built from libMesh[2] to simulate turbidity currents, typically found in geological processes. Within this context, sediments are transported due to the fluid movement and follow the rules of Navier-Stokes equations for incompressible fluid combined to the advection-dominated transport equation (sediment concentration).

In our simulation, the intermediate and final results of the meshes are stored in scientific data files, whereas other data of interest are kept in main memory (*e.g.*, variables of the solver code). Accordingly, libMesh-sedimentation was integrated with ParaView Catalyst [Ayachit et al. 2015] so that in-memory data can be extracted from the simulated mesh. Such scientific data files from in-memory data are stored into comma-separated values (`CSV`) files that include information such as the direction of fluid movement, the concentration of suspended and deposited sediments, the fluid pressure, and the coordinates of the mesh points. It is worth mentioning we choose plain `CSV` data files because they are a common format among the evaluated solutions and enable us to avoid the effective use of a binary data structure.

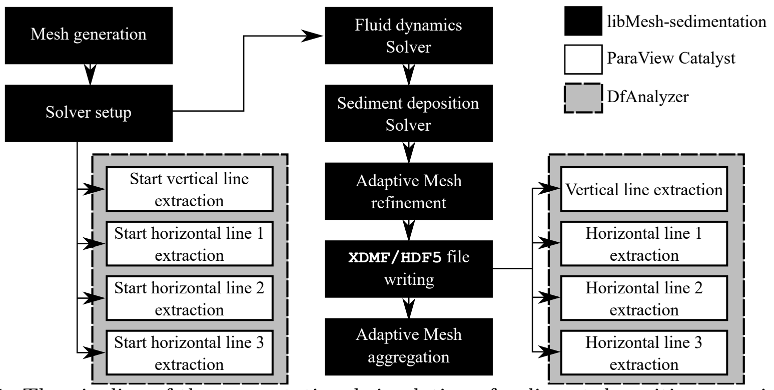
Mesh generation
Solver setup
Fluid dynamics Solver
Sediment deposition Solver
Adaptive Mesh refinement
XDMF/HDF5 file writing
Adaptive Mesh aggregation
libMesh-sedimentation
ParaView Catalyst
DfAnalyzer
Start vertical line extraction
Start horizontal line 1 extraction
Start horizontal line 2 extraction
Start horizontal line 3 extraction
Vertical line extraction
Horizontal line 1 extraction
Horizontal line 2 extraction
Horizontal line 3 extraction

Fig. 1: The pipeline of the computational simulation of sediment deposition experiment.

Additionally, we coupled the duo libMesh-sedimentation and ParaView Catalyst to the DfAnalyzer tool [Silva et al. 2018], which enables (i) runtime query processing on both scientific data coming out of libMesh and in-memory data extracted using ParaView Catalyst; and (ii) provenance query processing considering the dataflow paths generated by the computational simulation. Therefore, DfAnalyzer tool enables the dataflow monitoring in computational simulations through access, extraction, and indexing of scientific data, along with the capture of provenance data at runtime. Moreover, DfAnalyzer provides a complementary aspect to simulation programs regarding the finding of the variables of interest and/or files that need to have their content accessed, extracted and, eventually, indexed. DfAnalyzer is further designed to follow a W3C PROV compliant data model, named as PROV-Df [Silva et al. 2017]. Figure 1 shows the data transformation pipeline involved in libMesh-sedimentation (black squares) as well as the data transformations that use ParaView Catalyst (white squares) for raw data extraction. Such pipeline is related to the generation of both scientific data, which is straightforwardly stored into files, and in-memory information, which stores extracted data of in-memory structures to `CSV` files. Initially, a mesh is generated for the simulation and it is fed to the libMesh solver.

At this point, ParaView Catalyst extracts strategic simulation data of the solver data structures (horizontal lines) and persists them to files. For the remaining iterative part of the solver, libMesh generates `XDMF` and `HDF5` files that contain the simulation results, *i.e.*, the output mesh, whereas ParaView Catalyst keeps track of the mesh allocated in memory and stores them into `HDF5` files. Finally, the resulting aggregated mesh is persisted into a single `XDMF` file with references to the `HDF5` files. The provenance aspect of this simulation is carried out by DfAnalyzer (gray squares). Whenever information is gathered by either libMesh or Catalyst, DfAnalyzer is triggered such that it becomes

[2]`http://libmesh.github.io`

aware of data evolution during the simulation. In the next section, we discuss the processes of accessing, indexing and querying data extracted under the pipeline presented in Figure 1. More specifically, we considered all contents of the meshes generated by this simulation in the experimental evaluation of this paper (*i.e.*, output data of the data transformation `XDMF/HDF5` file writing).

4. ADAPTIVE QUERYING *VS.* RAW DATA FILE INDEXING

Previously in Section 2, we classified the approaches for accessing, extracting, indexing, and querying scientific data in two major categories: *adaptive querying* and *raw data file indexing*. We also highlighted that PostgresRAW and FastBit are representative solutions for those two categories. In this comparative analysis, we employ both PostgresRAW and FastBit. Our goal is to evaluate these two solutions regarding the following criteria:

(1) The elapsed time of query processing for each solution, *i.e.*, the time required for performing a query regardless of the data preparation, and
(2) The *time-to-query* demanded for each solution, *i.e.*, the joint time spent from data preparation to result set generation.

For this comparative analysis, we executed the turbidity currents simulation detailed in Section 3 in the Lobo Carneiro (LoboC) cluster[3] and used 960 cores in this execution. All computational simulation (including results, in-memory strategic scientific data and provenance data) generated approximately 47GB of data distributed across 97,163 files. Aiming at reducing the overall amount of raw data stored, we applied the ParaView Catalyst tool with default parameters for data selection and extracted 44,160 files from the overall results of the execution. Extracted data include 62 time steps (*i.e.,* solver iterations) divided into 480 files per *odd* time steps and 960 files per *even* time steps, on average. We have this variation on the number of files with extracted data due to the Adaptive Mesh Refinement (AMR) algorithm performed by libMesh-sedimentation solver at runtime, which implies the modification of the number of tetrahedrons in its output mesh. Figure 2 details the accumulated dataset size (in MB) for each time step.

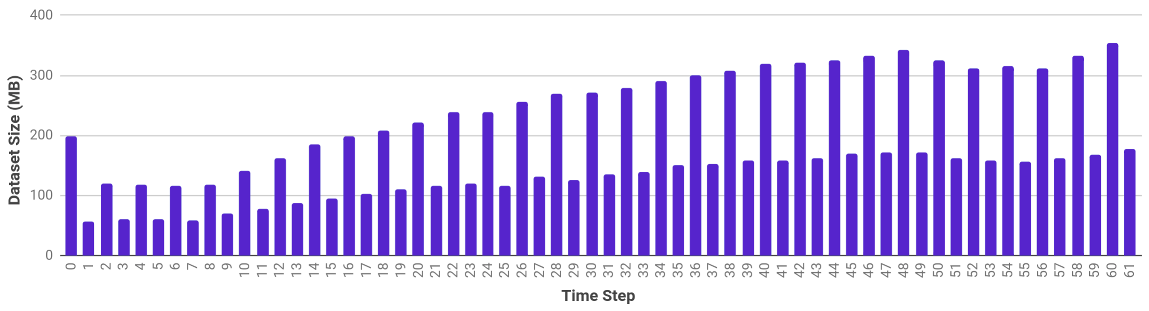

Fig. 2: Accumulated dataset size for each time step of the turbidity currents simulation.

In the next sections, we describe the setup and enhancements we made on both PostgresRAW and FastBit for a comparison between these solutions. All measures were taken after a series of 10 repetitions and all executions ran on the same hardware infrastructure in a Virtual Machine (VM) in the Google Cloud Platform[4]. We set a single VM to use 8 cores and 30GB RAM, whereas data were stored and retrieved from an SDD device. Also, we used Ubuntu 16.04.4 LTS OS, FastBit version 2.0.3[5] and PostgresRAW available in the last commit of its public repository[6] as of April 25, 2018.

[3]`http://www.nacad.ufrj.br/en`
[4]`https://cloud.google.com/`
[5]`https://code.lbl.gov/projects/fastbit/`
[6]`https://github.com/HBPMedical/PostgresRAW`

4.1 Setting up PostgresRAW

PostgresRAW manages open files by using an array in which every position corresponds to a file pointer, and this array is instantiated for each ODBC-like request. By default, PostgresRAW is configured for handling an array of fixed 50 positions, which is insufficient for the evaluated computational simulation. However, when the array size is increased, either statically or dynamically, it incurs into a substantial delay for each database connection. The cause of such delay is related to the strategy for fetching file pointers from the collection rather than the array structure itself.

After analyzing the PostgresRAW source code where it reads the file that is used to load data to the vector, we discovered that PostgresRAW searches the entire array twice to find an empty position to be associated to a new file or an entry whose identifier equals to the file identifier to be updated. The structure of each array has four attributes (file location, table name, delimiter character, and a flag for header ignoring) as follows:

```
filename-ID  = ‘FILE_LOCATION’
relation-ID  = ‘TABLE_NAME’
delimiter-ID = ‘,’
header-ID    = ‘TRUE | FALSE’
```

Therefore, we had to modify the data loading algorithm of PostgresRAW to use the file identifier as the array index. Whenever there is memory to be spent in pointers, the array is of fixed size and query complexity is bounded to $O(1)$. Otherwise, the array is a sparse implementation and query complexity becomes $O(log(\#\text{ of entries}))$. We also modified the PostgresRAW core for addressing the limitation of not handling more than 5 tables at the same time. Once again, the original source code was limited by a fixed array that kept query and table metadata in the main memory, which required the array to be converted into a dynamic collection with a proper reallocation method.

4.2 Setting up FastBit

Unlike PostgresRAW that relies on adaptive querying for generating its positional indexes, FastBit enables the user to define the parameters for indexing and tuning *a priori*. In particular, FastBit can be set towards either *binning* or *encoding* parameterizations. The *Binning* parameter determines how the bitmap is produced and, in our tests we employed the *default* value, *i.e.,* parameter $precision = 2$. On the other hand, *encoding* parameterization requires finding the *trade-off* of two settings (space and performance), which are usually achieved by several experimental evaluations that scales to the size of managed data. Accordingly, in our experiments, we used the *binning* parametrization.

4.3 Comparison of indexing generation and storage routines

FastBit generates bitmap indexes for raw data files, whereas PostgresRAW generates initial positional references to the raw data file entries. In both cases, such index generation ran *before* data querying. Accordingly, we measured the elapsed time and disk storage demanded by the two competitors. Table II presents the comparison of the elapsed time demanded for index creation upon data extracted from `XDMF` and `HDF5` files. PostgresRAW required a much smaller (in up to 2 orders of magnitude, on average) index construction time in comparison to FastBit. Unlike FastBit that relies on a single *one-pass* routine for the construction of its index (it indexes all values in a raw data file), PostgresRAW uses an in-memory incremental function for adding entries into its positional index. Thus, it is expected the first queries submitted to PostgresRAW to take longer than in FastBit as they also require the update of the positional.

Table II also presents the disk space required for each compared approach. Overall results indicate disk usage follows a similar behavior of that observed on index generation, *i.e.*, PostgresRAW has required up to 30 times less disk space for index storage in comparison to FastBit. Although PostgresRAW index size is expected to increase whenever queries on new attributes/columns arrive, the index size result reinforces FastBit heavily relies on redundant strategy for representing the entries on raw data files. FastBit's strategy seems to scale linearly to the number of entries $\times$ files. On the other hand, as queries are submitted to PostgresRAW, the disk usage tends to increase over time since PostgreRAW generates indexes on the fly.

Table II: Comparison of elapsed time and disk space required for the generation of FastBit and PostgresRAW indexes.

Measurements		Indexing solutions	
		FastBit	**PostgresRAW**
Elapsed time for index construction (s)	Mean	20,396.07	264.49
	Standard deviation	1,406.65	0.73
Disk usage (GB)		18.00	0.53

4.4 Comparison of raw data file querying

In this section, we use the indexes generated in the previous section and pose a group of representative queries upon the raw data files we obtained after the CFD simulation. These queries are based on the typical on-line queries users make on CFD simulation data. The overall idea is the submission of six high-level SQL-based queries to the competing solutions, whereas each query is related to every time step of the turbidity currents simulation. Therefore, we submit $6 \times 62 = 372$ queries (one query for every time step), executed each 10 times, and measured the elapsed time of each solution required to return the result set. Table III describes the domain queries in terms of their selectivity and the motivation for each SQL-based statement. Query selectivity is defined as a floating-point value between zero and one that represents the percentage of rows to be returned. Based on such a definition we classified queries into three types: low-selective, medium-selective and high-selective queries. High-selective queries are the ones that return up to 1% of the original tuples. Medium-selective queries are the ones that return from 1% up to 50% the original set of tuples. Finally, low-selective queries those that return more than 50% of the tuples.

Table III: Six domain queries to be applied over each time step of the CFD simulation.

Label	Selectivity	Description
`Q1`	Low	Retrieves data from multiple columns and files at same time.
`Q2`	High	Retrieves extreme values within multiple columns and files. `Q2` uses the SQL aggregate function `MAX`, whereas `Q3` applies the SQL aggregate function `MIN`.
`Q3`		
`Q4`	Low	Retrieves data from a single column from multiple files.
`Q5`	Medium	Filters data from a single column of multiple files.
`Q6`	Medium	Filters data from multiple columns and files.

Query `Q1` selects all data from all files that represent a given time step, *i.e.*, it joins the content within all of the time steps files into a single output. Queries `Q2` and `Q3` apply the SQL aggregate function about the maximum and minimum values of specific attributes from multiple files. Query `Q4` selects all data of a single attribute from multiple files, whereas query `Q5` filters such values by using an order-based predicate with medium selectivity. Finally, query `Q6` filters all attributes after a Cartesian product where a second column is filtered by a predicate with a medium selectivity condition.

Figure 3 presents the elapsed time regarding the execution of each query. We accumulated the elapsed times of the queries regarding the interval of time steps from 0 to 61 in steps of 10. Therefore, the first portion of each stacked graph corresponds to the CFD time steps 0 to 9, and so on. The last portion of stacked bars includes the time step interval of 60 to 61. Figure 3 shows the overall results for `Q1`, where PostgresRAW was 37.92% faster than FastBit, as FastBit needs to read all column files and matches the rows to produce the results for this query in particular. For all other queries, however, FastBit performed better than PostgresRAW. In particular, FastBit was up to 92.79%,

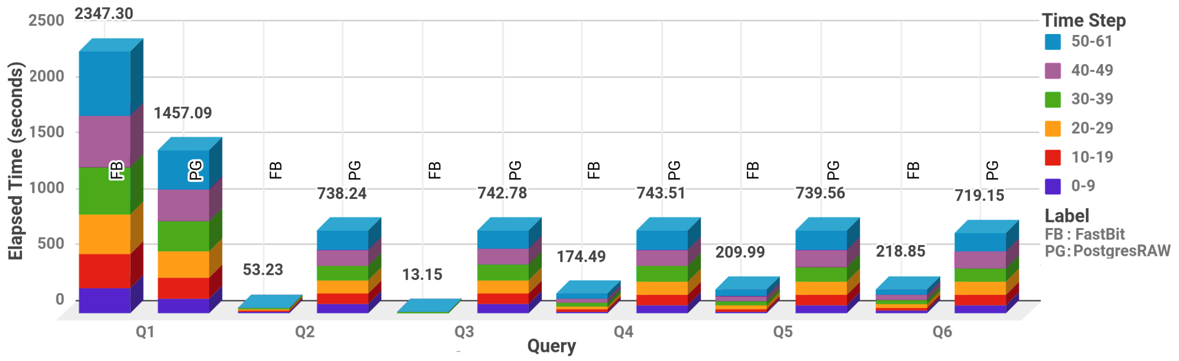

Fig. 3: Elapsed time of the query processing after data indexing.

98.23%, 76.53%, 71.61%, and 69.57% better than PostgresRAW regarding queries Q2, Q3, Q4, Q5, and Q6, respectively. Moreover, PostgresRAW query execution time showed nearly the same behavior for the all queries regardless of query selectivity.

On the other hand, FastBit is sensitive to the interval of time steps and also to the query selectivity. For instance, the sum of the elapsed time of queries Q2, Q3, and Q4 is smaller than the sum of the elapsed time of Q5 and Q6. Besides query selectivity, queries Q5 and Q6 require Cartesian products and also retrieve more attributes than the previous ones. FastBit execution on queries Q5 and Q6 indicates that the larger the portion (and number) of indexes FastBit needs to scan, the higher the accumulated query cost because it also involves the retrieval of index blocks themselves that are stored into the secondary memory file system.

We also evaluated the performance of FastBit and PostgresRAW regarding their elapsed times for running queries on each time step. Figure 4 and Figure 5 shows the comparison for each time step regarding query Q3. Such a query is representative because it traverses an entire common column between the multiple raw data files. Bottom of Figure 4 and Figure 5 present the performance of both indexes for querying the raw data files that store the information about the *odd* time steps, whereas the top of Figure 4 and Figure 5 shows the behavior of the competitors regarding the *even* time steps, respectively. Overall results indicate that FastBit is able to retrieve data in constant time for query Q3 as it employed a fixed-size indexing structure that enables a single scan of the entire set of values from the columns. On the other hand, PostgresRAW presents an elapsed time that increases according to the dataset size, which highlights its dynamic indexing structure is sensitive to data cardinality.

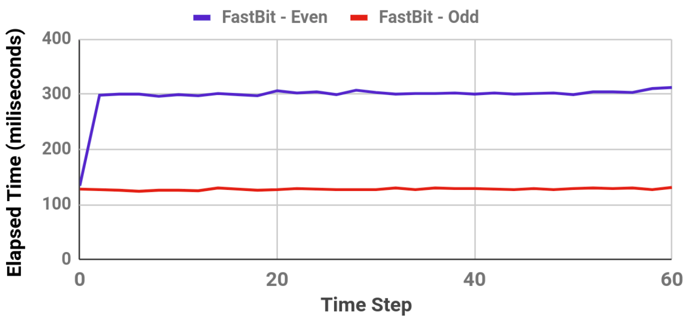

Fig. 4: FastBit Query performance for Query Q3

4.5 Comparison of *time-to-query*

In our last experiment, we measured the average *time-to-query* for each solution to query the scientific data of the CFD simulation. In this scenario, we suppose users want to perform a *single* query over the

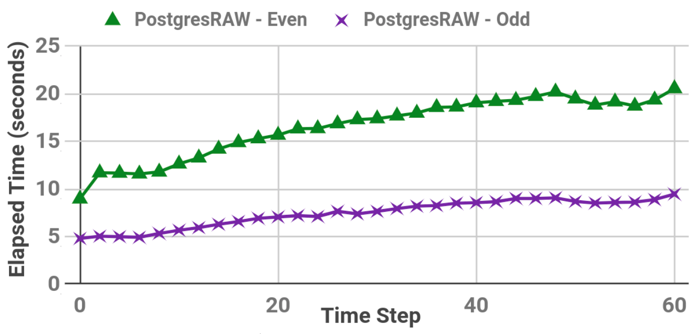

Fig. 5: PostgresRAW Query performance for Query `Q3`

time steps instead of the six sequential queries of Section 4.4. Therefore, the elapsed time demanded for raw data file indexing is added to the data querying time.

Figure 6 presents the *time-to-query* required for the six different searches on the CFD simulation files regarding all 62 time steps. Overall results indicate PostgresRAW is more suitable than FastBit in this context because it demanded less time to reach the results. In particular, PostgresRAW was up to 92.43%, 95.10%, 95.06%, 95.10%, 95.13%, and 95.23% better than FastBit regarding queries `Q1`, `Q2`, `Q3`, `Q4`, `Q5`, and `Q6`, respectively. The lowest gain of PostgresRAW over FastBit was on query `Q1` that is a column-oriented query followed by a union-all of multiple files. Besides the query itself, such a gain is also related to the index construction procedures, *i.e.,* while the entire values were scanned to construct the bitmap indexes, PostgresRAW did only performed a mapping of the queried columns.

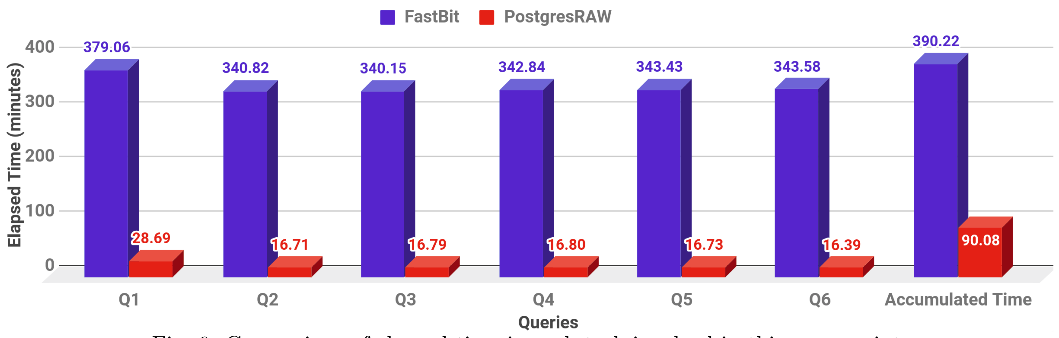

Fig. 6: Comparison of elapsed time in each task involved in this manuscript.

Although FastBit query execution time *after* index construction is faster than the query execution time on PostgresRAW, the elapsed time spent on index construction takes longer to pay off. In a scientific data querying scenario such as the CFD simulation, even for the accumulative elapsed time demanded by the execution of all six queries, PostgresRAW has required a much lower time-to-query in comparison to FastBit (Figure 6 - Accumulated time).

5. CONCLUSION

The loading of all scientific data makes a DBMS inviable for the handling of huge data volume generated by computational simulations. In addition, most of produced data is not queried over time, so it is impractical to load data to the database that is not going to be queried. Such data must be extracted and indexed to allow for querying capabilities of existing tools. In this article, we present a comparative analysis of representative and alternative raw data file querying solutions, *i.e.*, FastBit and PostgresRAW. The former stands for the *raw data file indexing* tuning approaches, whereas the latter represents the *adaptive searching* paradigm. In our analysis, we considered a real-world computation simulation and compared the competing approaches in terms of their indexing construction costs, query processing costs, and data storage requirements. All experiments were performed on either LoboC cluster or Google cloud platform, and we integrate libMesh-sedimentation, ParaView Catalyst and DfAnalyzer tools to ensure the capture of simulation, in-memory and provenance data.

Global results of our empirical comparison indicate PostgresRAW presents a better performance for data query whenever we consider the time-to-query metric. Likewise, PostgresRAW demanded smaller disk storage for its incremental indexing method, while FastBit approach has demanded 30× more space. As a matter of *trade-off*, FastBit outruns PostgresRAW after data indexing. However, when we take into account the overall *time-to-query* PostgresRAW was more efficient than FastBit.

Acknowledgments. Authors thank CAPES, CNPq and FAPERJ for their financial support.

REFERENCES

Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. ACM, New York, NY, USA, pp. 241–252, 2012a.

Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. Nodb in action: Adaptive query processing on raw data. *Proc. VLDB Endow.* 5 (12): 1942–1945, Aug., 2012b.

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., and Mauldin, J. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization.* ISAV2015. ACM, New York, NY, USA, pp. 25–29, 2015.

Blanas, S., Wu, K., Byna, S., Dong, B., and Shoshani, A. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. ACM, New York, NY, USA, pp. 385–396, 2014.

Camata, J. J., Silva, V., Valduriez, P., Mattoso, M., and Coutinho, A. L. In situ visualization and data analysis for turbidity currents simulation. *Computers and Geosciences* vol. 110, pp. 23 – 31, 2018.

Chou, J., Howison, M., Austin, B., Wu, K., Qiang, J., Bethel, E. W., Shoshani, A., Rübel, O., Prabhat, and Ryne, R. D. Parallel index and query for large scale data analysis. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '11. ACM, New York, NY, USA, pp. 30:1–30:11, 2011.

Clarke, J. A. and Mark, E. R. Enhancements to the extensible data model and format (xdmf). In *DoD High Performance Computing Modernization Program Users Group Conference.* DoD HPCMP '07, vol. 1. IEEE, Pittsburgh, PA, USA, pp. 322–327, 2007.

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

Deshpande, A., Ives, Z., and Raman, V. Adaptive query processing. *Foundations and Trends® in Databases* 1 (1): 1–140, 2007.

Guedes, T., Sousa, V. S., Camata, J. J., Mattoso, M., and de Oliveira, D. Análise de dados científicos: uma análise comparativa de dados de simulações computacionais. In *XXXII Simpósio Brasileiro de Banco de Dados - Short Papers, Uberlandia, MG, Brazil, October 4-7, 2017.* pp. 222–227, 2017.

Karpathiotakis, M., Branco, M., Alagiannis, I., and Ailamaki, A. Adaptive query processing on raw data. *Proc. VLDB Endow.* 7 (12): 1119–1130, Aug., 2014.

Silva, V., de Oliveira, D., Valduriez, P., and Mattoso, M. Dfanalyzer: Runtime dataflow analysis of scientific applications using provenance. *Proc. VLDB Endow.* 11 (12): 2082–2085, Aug., 2018.

Silva, V., Leite, J., Camata, J. J., de Oliveira, D., Coutinho, A. L., Valduriez, P., and Mattoso, M. Raw data queries during data-intensive parallel workflow execution. *Future Generation Computer Systems* vol. 75, pp. 402 – 422, 2017.

Wang, H. and Zhai, Z. J. Advances in building simulation and computational techniques: A review between 1987 and 2014. *Energy and Buildings* vol. 128, pp. 319 – 335, 2016.

Wu, K. Fastbit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series* 16 (1): 556, 2005.