SAVIME: A Database Management System for Simulation Data Analysis and Visualization

Hermano Lustosa, Fabio Porto, Patrick Valduriez

SAVIME: A Database Management System for Simulation Data Analysis and Visualization

Hermano Lustosa[1], **Fabio Porto**[1], **Patrick Valduriez**[2]

[1]National Laboratory for Scientific Computing (LNCC)
Petrópolis – RJ – Brazil

[2]Inria and LIRMM
Montpellier – France

{hermano,fporto}@lncc.br, patrick.valduriez@inria.fr

Abstract. *Limitations in current DBMSs prevent their wide adoption in scientific applications. In order to make scientific applications benefit from DBMS support, enabling declarative data analysis and visualization over scientific data, we present an in-memory array DBMS system called SAVIME. In this work we describe the system SAVIME, along with its data model. Our preliminary evaluation show how SAVIME, by using a simple storage definition language (SDL) can outperform the state-of-the-art array database system, SciDB, during the process of data ingestion. We also show that is possible to use SAVIME as a storage alternative for a numerical solver without affecting its scalability.*

1. Introduction

The increasing computational power of HPC machines allows for performing complex numerical simulations. These simulations produce huge datasets, which are analyzed and visualized to enable researchers to gain insights about the phenomena being studied. Traditionally, simulation code stores its raw data in the file system, and another application reads it from disk, performs analysis and creates the visualization files. However, due to the I/O gap in HPC environments, doing so can be very inefficient for large scale simulations [Ahrens 2015]. Two popular approaches, in-situ and in-transit analysis [Oldfield et al. 2014], have been proposed to address this problem by favoring intense memory usage instead of relying on disk storage.

DBMSs are not commonly adopted in any of these approaches. In-situ and in-transist analysis relies on libraries, and the post-processing approach consists of storing data in scientific data formats such as HDF[Group 2017] and NetCDF[Unidata 2017]. DBMSs are considered inadequate for scientific data management due to many factors. The first one is the impedance mismatch problem [Blanas et al. 2014, Gosink et al. 2006], i.e., the incompatibilities between the representation formats of the source data and the DBMS. This impedance mismatch yields costly conversions between formats, which adds prohibitive overhead during data ingestion.

Also, data usage patterns for scientific applications are very different from those common in commercial applications. The workload is mainly analytical and not necessarily all data needs to be persisted. It is common that data is analyzed and summarized, having its volume being drastically reduced by either storing only the summarized version or by simply discarding parts which are not interesting. Some other key points also do not favor

DBMS usage. Data is heterogeneous, comprising different formats and sources, making it hard to use a single database solution for all data. The analysis is also complex and, in many cases, cannot be easily done with a declarative language like SQL.

However, for the vast variety of queries that can be expressed in a declarative language, a DBMS solution could offer a convenient and efficient way to perform analysis, given that the underlying data model is flexible enough to accomodate such data, and that the process of data ingestion is seamless, not incuring in costly data conversions.

Therefore, given the current lack of a database solution that could facilitate the simulation data analysis and visualization, we propose an array based data model [Lustosa et al. 2017] named TARS, to cope with simulation data and to allow a more efficient representation, along with a prototype system that implements this model, currently named SAVIME. SAVIME supports a DDL and a SDL that enable fast data ingestion, and a DML that allows for declarative analysis and visualization of simulation data.

In this paper, we present SAVIME and the TARS data model, along with an evaluation in which we compare SAVIME with SciDB, the state-of-the-art array DBMS. This document is organized as follows. In Section 2 we discuss the TARS data model implemented in SAVIME. In Section 3 we present SAVIME, its execution model and its DDL, SDL and DML. In Section 4 we show the results of our evaluation comparing SAVIME and SciDB and embedding SAVIME with a real life application. In section 5, we discuss the related work and finally in Section 6 we conclude.

2. Typed Array Data Model

Scientific data is usually represented as multidimensional arrays, which are common as the result of scientific experiments, measurements and simulations. In short, an array is a regular structure formed by a set of dimensions. A set of indexes, one per dimension, identifies a cell that contains values for array attributes.

If carefully designed, arrays offer advantages when compared to tables. Cells in an array are ordered, unlike tuples in a relation. Thus, an array DBMS can quickly lookup cells by taking advantage of this ordering. Using arrays instead of tables can also save storage space, since array indexes do not need to be stored and can be inferred by the position of the cell. Furthermore, arrays can be split into subarrays, called tiles or chunks. These subarrays are used as processing and storage units, and help answering queries efficiently. Thus, if your data is array-ike, using an array database is the way to go.

However, current array data model implementations, e.g., SciDB and RasDaMan, have limitations, preventing a more wide acceptance in scientific applications. In SciDB [Paradigm4 2017] for instance, due to some representation constraints, it might be necessary to preload multidimensional data into an unidimensional array and then rearrange it during data loading. RasDaMan requires either the creation of a script or the generation of compatible file formats for data ingestion. The result is an inefficient loading process in both cases.

Scientists do not necessarily want to persist all data for a long period of time, instead they want to analyze huge dataset as switfly as possible. Therefore, adopting a DBMS which imposes high overhead for data ingestion does not make sense, even an array DBMS that offers a data representation adequate for scientific data, because the possible

convenience of using a declarative query language does not compensate the trouble to ingest data into the system. In this section we present a briefly overview of the TARS model as presented in [Lustosa et al. 2017].

2.1. Model Overview

Given the lack of flexibility in current array DBMSs, we propose the TARS data model to cope with the aforementioned issues. A TAR Schema (TARS) contains a set of Typed ARrays (TARs). A TAR has a set of dimensions and attributes. A TAR cell is a tuple of attributes accessed via a set of indexes. These indexes define the cell location within the TAR. A TAR has a type, formed by a set of roles. A role in a type defines a special purpose data element with specific semantics.

In TARS (Figure 1), we define mapping functions as a way to provide support for sparse arrays, non-integer dimensions and heterogeneous memory layouts. With TARS, it is possible to combine array data from different sources, different storage layouts, and even with different deegres of sparsity by associating different mapping functions to different subarrays, or as we call them, subTARs.

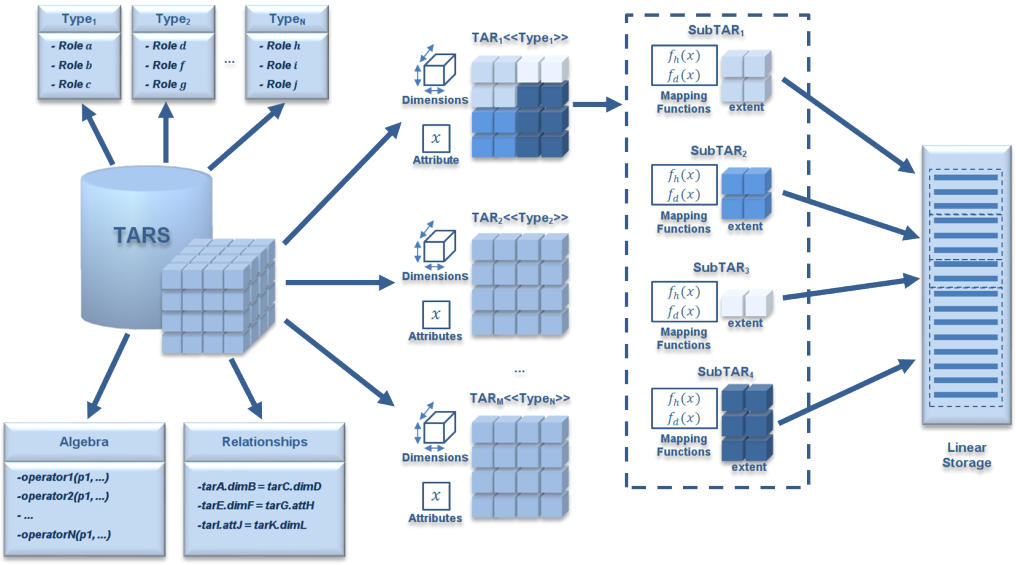

Figure 1. Typed Array Schema Elements

A subTAR covers a n-dimensional slice of a TAR. Every subTAR is defined by the TAR region it represents and two mapping functions: position mapping function and data mapping function. The position mapping function reflects the actual data layout, since it defines where every TAR cell within a given subTAR ended in linear storage. Therefore, the position mapping function should implement the multidimensional linearization technique used for the data. The data mapping functions translate a linear address into data values. In a simple scenario, this function does basically a lookup into a linear array that stores the data. In a more complex scenario, it could compute a derived value from the actual subTAR data.

2.2. Physical Specification

In this section, we describe how the TARS data model is implemented in SAVIME. TARS structures are created in SAVIME with the use of the supported SDL and DDL. Users can

define TARs, datasets, and types. Once a TAR is defined and a series of datasets are loaded into the system, it is possible to specify a subTAR by attaching datasets to it. A dataset is a collection of data values of the same type, like a column in a column-store DBMS (SAVIME uses vertical partitioning). A dataset can contain data for a TAR attribute within a TAR region specified by a subTAR.

TAR dimensions indexes form a domain of values that are represented in SAVIME in two main forms. It can be an implicitly defined range of equally spaced values, in which case, all the user must specify is the lower and upper bounds, and the spacing between two adjacent values. It is called implicit because these indexes do not need to be explicitly stored. For instance, the domain $D_i = (0.0, , 2.0 , 4.0 , 6.0 , 8.0 , 10.0)$ is defined by the lower bound 0.0, the upper bound 10.0 and all values are equally spaced in 2.0 units.

Dimensions whose indexes do not conform with these constraints have an explicit definition. In this case, the user provides a dataset specifying the dimension indexes. For instance, consider the following domain $D_e = (1.2, , 2.3 , 4.7 , 7.9 , 13.2)$. It has a series of values that are not well-behaved and equally spaced, and thus, can not be represented implicitly.

The data representation within the subTAR requires the combination between the dimension domain and the dimension specifications. All subTARs in a TAR have a list of dimension specifications, one for each dimension in the TAR. These dimension specifications define the TAR region the subTAR encompasses, but they also are a fundamental part in the implementation of the mapping functions. These functions are defined conceptually in the model, but are implemented considering six possible configurations between dimension specifications (ORDERED, PARTIAL and TOTAL) types and dimension types (IMPLICIT and EXPLICIT).

An ORDERED dimension specification indicates that the indexes for the cells in that dimension are dense and sorted in some fashion. A PARTIAL dimension implementation, indicates that there are some holes in the datasets, meaning that some cells at given indexes are not present. Finally the TOTAL representation indicates that data is fully sparse and that all indexes must be given for every cell, in other words, it means that we have a degenerated array that is basically tabular data.

3. The system SAVIME

SAVIME has a component-based architecture common to other DBMSs, containing modules such as an optimizer, a parser and an query processing engine, along with auxiliary modules to manage connections, metadata and storage. A SAVIME client communicates with the SAVIME server by using a simple protocol that allows both ends to exchange messages, queries and datasets. All modules are currently implemented as a series of C++ classes, each one of them with an abstract class interface and an underlying concrete implementation.

3.1. Languages DDL, SDL and DML

SAVIME's DDL supports operators to define TARS and Datasets, for instance, the commands:

```
CREATE_TAR("FooTAR", "*", "Implicit, I, long, 1, 1000, 1
                         | Implicit, J, long, 1, 1000 , 1",
```

```
                              "attrib, double");
CREATE_DATASET("FooBarDS1:double", "ds1_data_source");
CREATE_DATASET("FooBarDS2:double", "ds2_data_source");
LOAD_SUBTAR("FooTAR", "Ordered, I, 1, 100 | Ordered, J, 1, 100",
            "attrib, FooBarDS1");
LOAD_SUBTAR("FooTAR", "Ordered, J, 101, 200 | Ordered, I, 1, 100",
            "attrib, FooBarDS2");
```

Initially we can issue a CREATE_TAR command to create a TAR name FooTAR. It has 2 dimension (I and J) whose indexes are long integers. These are implicit dimensions, whose domains are integers equally spaced by 1 unit from 1 to 1000. This TAR also has a single attribute named attrib whose type is a double precision real number.

After that we create 2 datasets named FooBarDS1 and FooBarDS2, they are double typed collections of values in a data source (usually a file or memory based file). Finally we issue 2 LOAD_SUBTAR commands to create 2 new subtars for the TAR FooTAR, the first one ecompasses the region that contains the cell s whose indexes are in $[1, 100] \times [1, 100]$ for dimension I and J respectively, in both case we have an ordered representation indicating that data is dense and ordered first by the I index and second by J index. The second subtar, however, encompasses the cells whose indexes are in $[1, 100] \times [101, 200]$ but instead ordered first by J index and second by the I index. It is an example of how the SDL works, since users can express and consolidate data sources with different ordering layouts into a single TAR. The final part of the command indicates that datasets FooBarDS1 and FooBarDS2 are attached to the "attrib" attribute, meaning that the data for "attrib" in the cells whithin each subTAR region can be found in these datasets.

SAVIME also supports a functional DML with operators similar to the ones implemented in SciDB, for operations such as filtering data based on predicates, calculating derived values, joins and aggregations. Here is an example of a query in SAVIME.

```
AGGREGATE(
          WHERE(
                 DERIVE(FooTAR, attrib2, attrib*attrib),
                 attrib2 >= 2.0 and attrib2 <= 10.0
                ),
           sum, attrib2, sum_attrib2, I
);
```

This DML query consists of three nested operators. Initially, a new attribute called attrib2 is created by the operator DERIVE and its value is defined as the square of the attribute attrib. After that, the WHERE operator is called, it filters data according to the predicate, in this case, it returns a TAR whose cells have the value for attrib2 set between 2 and 10. Finally, we use the AGGREGATE operator to group data by dimension I indexes and sum the value for attrib2 creating the sum_attrib2, the resulting TAR will present only one dimension (I) and a single attribute sum_attrib2 whose values are the result of the sum of the attrib2 accross dimension J.

3.2. Query Processing

As presented in the previous section, a SAVIME query contains a series of nested operators. Most of them expect one or more input TARs, and originate a newly created output TAR.

Unless a special operator is called to materialize the query resulting TAR, it is generated as a stream of subTARs, sent to the client and then discarded. SAVIME operates TARs as a subTARs stream pipelined across operators. SubTARs are processed serially or in parallel with OpenMP constructs.

During query processing, when a subTAR for a TAR holding intermediated results is generated and passed on to the next operator, it is maintained in a temporary subTARs cache. These subTARs contain their own group of datasets that could require a lot of storage space or memory. Therefore, once a subTAR is no longer required by any operator, it must be removed from memory. An operator implementation is agnostic regarding its previous and posterior operations in the pipeline, and does not know when to free or not a subTAR. All the operators implementation needs to establish is when it will not require a given subTAR any longer. When this happens, the operator notifies the execution engine that it is done with a given subTAR and it is then discarded.

However, since the same subTAR can potentially be input into more than one operator during a query, freeing it upfront is not a good idea, because it might be required again. In this case, SAVIME would have to recreate it. To solve this problem, every subTAR has an associated counter initially set to the number of operators that have its TAR as their input. When an operator notifies the engine that it no longer needs that specific subTAR, the respective counter is decreased. Once the counter reaches zero, all operators possibly interested in the subTAR are done, and now it is safe to free the subTAR. This approach always frees the used memory as soon as possible and never requires a subTAR to be created twice. However, some operators might require many subTARs to be kept in memory before freeing them. In an environment with limited memory, it would not be feasible to cope with very large TARs in this case. A solution then, would be the adoption of a more economical approach, trading off space with time by freeing and regenerating the subTARs whenever memory is running low.

4. Experimental Evaluation

We ran a series of experiments in order to validate SAVIME as a feasible alternative to simulation data management. We compare SAVIME with SciDB and evaluate how SAVIME affects the performance of actual simulation code.

4.1. SAVIME vs. SciDB

In this section, we compare SAVIME, SciDB (version 16.9) and a third approach based on the usage of NetCDF files (version 4.0), used as a baseline. All scripts, applications and queries used in our evaluation are available at `github.com/hllustosa/savime-testing`.

We use two datasets, a dense and a sparse one, based on data from the HPC4e BSC seismic benchmark [Center 2016] in our experiments. The dense dataset contains 500 trials (one dataset for each) for a 3D regular mesh with dimensions 201x501x501 containing a velocity field. In total, we have over 30 billion array cells and more than a 120 GB of data. All data is held in a single 4D structure (TAR in SAVIME, array in SciDB and in a single NetCDF file) containing the X, Y, and Z dimensions, and an extra trial dimension to represent all 500 trials. Both structures have the same tiling configuration, i.e., the same number of chunks/subTARs (500 of them, one for each trial) with the same extents.

The sparse dataset is also a 4D structure with 500 simulation trials, but only a subset of the cells are present (around 24% of the dense dataset). It comprises almost 8 billion array cells and over 30 GB of data. We used a sparse 4D array/TAR in SciDB and SAVIME, and a 2D dense array in NetCDF. NetCDF lacks the ability to natively represent sparse data, thus we indexed the x, y and z values and represented them as a single dimension and stored coordinate values as variables.

The computational resource used is the fatnode from the cluster Petrus at DEXLab. This fatnode has 6 Intel(R) Xeon(R) CPU E5-2690 processors amounting to 48 cores and over 700 GB of RAM. Data is kept in a shared-memory file system to simulate an in-transit data analysis, in which data is not kept on disk (for both SAVIME and SciDB). Initially, we evaluate the loading time of 500 tiles/chunks in all three approaches, considering that data is being transferred and continually appended to a single array/TAR/file as it is being generated by a solver.

As we can see in Figure 2 on the left graph, the ingestion time taken by SciDB is almost 20 times longer than the time taken by SAVIME, due to costly rearrangements needed on data to make it conform with the underlying storage configuration. Besides, there is an extra overhead during the lightweight data compression done by SciDB, which makes the dataset roughly 50% smaller when stored but increased loading time prohibitively. In contrast, SAVIME does not alter or index the data during the process of data ingestion, therefore the loading process is computationally much cheaper.

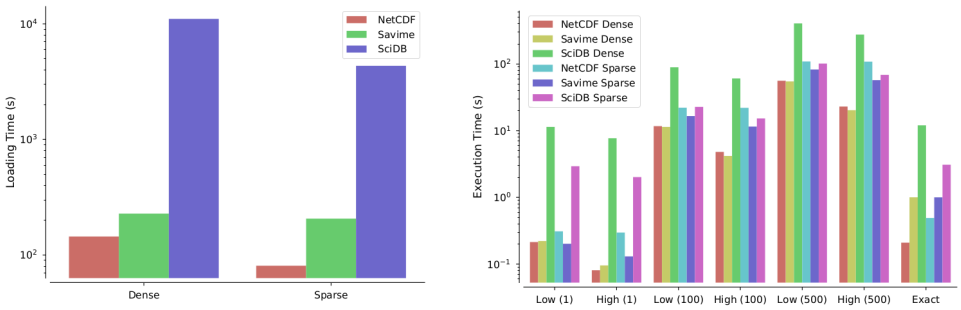

Figure 2. Ingestion and query execution time

We evaluate the performance considering ordinary and exact window queries. Ordinary Window queries consist of retrieving a subset of the array defined by a range in all its dimensions. The performance for this type of queries depends on how data is chunked and laid out. High selectivity queries, which need to retrieve only a very small portion of the data tends to be faster than full scans. Therefore, we compared low and high selectivity queries, filtering from a single to all 500 tiles. We also considered the performance of exact window queries, which is the easiest type of Window Query. They consist of retrieving data for a single tile or chunk, meaning the system has close to zero work filtering out the result.

We implement these queries as specific operators in SAVIME and SciDB, and with the help with a custom OpenMP application using the NetCDF library. The experimental results are shown in Figure 2 on the right graph. The average time of 30 runs are presented.

We considered window queries with low selectivity (over 70 % of all cells in a tile) and high selectivity (around 20 % of all cells in a tile), and intersecting with only 1, 100 or even the total 500 tiles.

It is noticeable that SAVIME either outperforms SciDB or is as efficient as it in all scenarios. The most important observation is that, even without any previous data preprocessing, SAVIME is able to simply take advantage of the existing data structure to answer the queries efficiently, which validates the model as a feasible alternative to existing implementations. The results show that, for any storage alternative in both dense and sparse formats, the subsetting of a single tile is very efficient. The differences shown for the exact window query and for low and high selectivity window queries that touch a single tile are very small. SciDB takes a few seconds in most cases, while SAVIME takes in average 1 second. NetCDF is the most efficient in this scenario, retrieving desired data in less than a second.

However, for queries touching 100 or 500 chunks, we can see the differences between querying dense and the sparse arrays. The dense dataset is queried more efficiently, since it is possible to determine the exact position of every cell and read only the data of interest. It is not possible for sparse data, since one is not able to infer cell positions within the tiles. In this case, every single cell within all tiles that intersect with the range query must be checked.

In dense arrays, we can observe a reduced time for retrieving data in high selectivity queries in comparison with low selectivity queries. The execution time of window queries should depend only on the amount of data of interest, since cells can be accessed directly and thus, no extra cells need to be checked. The execution times considering 100 or 500 tiles in SAVIME and NetCDF are in accordance with this premise. However, SciDB shows poorer performance, being up to 8 times slower. It is very likely that SciDB needs to process cells outside of the window of interest depending on the compression technique and the storage layout adopted. SciDB seems to be more sensible to tiling granularity, requiring fine-grained tiles that match the window query to have a performance similar to the NetCDF approach.

There is not much to be done for querying sparse arrays except for going through every cell in the tiles intersecting the window specified by the query. The query time for sparse data in all alternatives show very similar performance. The main difference is that for achieving this result with NetCDF, an OpenMP application needed to be written, while the same result could be obtained with a one-line query in SAVIME and SciDB.

Our conclusion is that the regular chunking scheme imposed by SciDB not only slows down the ingestion process significanly as it has not real impact in improving performance for simples operations, since SAVIME using a more flexible data model can solve similtar queries presenting a compatible performance.

4.2. Integration with Numerical Solver

In this section, we evaluate the amount of overhead imposed to the simulation code when integrating with SAVIME. We use the simulation tools based on the MHM numerical method [Gomes et al. 2017] as a representative numerical simulation application. We compare three approaches. In the first approach, SAVIME is used IN-TRANSIT, in a single node (fatnode) while the simulation code runs in a different set of nodes, and thus

data needs to be transferred. In the second approach, SAVIME is used IN-SITU, with individual SAVIME instances running on each node, the same used by the simulation code. In this scenario, the data does not need to be transferred, since it is maintained in a local SAVIME instance that shares the same computational resources used by the simulation code. In the third approach SAVIME is not used, but instead, the data is stored in ENSIGHT files (the standard file format used by MHM), and analysis are performed by an ad-hoc Message Passing Interface application in Python. This last scenario serves as a baseline implementation, thus we call it the baseline approach. The computational resource used is the Petrus cluster at DEXLab, with 8 nodes, each with 96 GB of RAM and 2 Intel(R) Xeon(R) CPU E5-2690 processors.

Preliminarily, we start the evaluation by measuring the overhead of loading data in a remote node running SAVIME. In Figure 3 we can see the time of running the simulation and discarding the data, which is the time to solely carry out the computations without any I/O whatsoever and the time to transfer and load data into SAVIME. We vary the size of the MHM meshes, which impact on the level of detail of the simulation. The larger the mesh size is, the more realistic and complex the simulation is, and also, more resources (time and memory) are consumed.

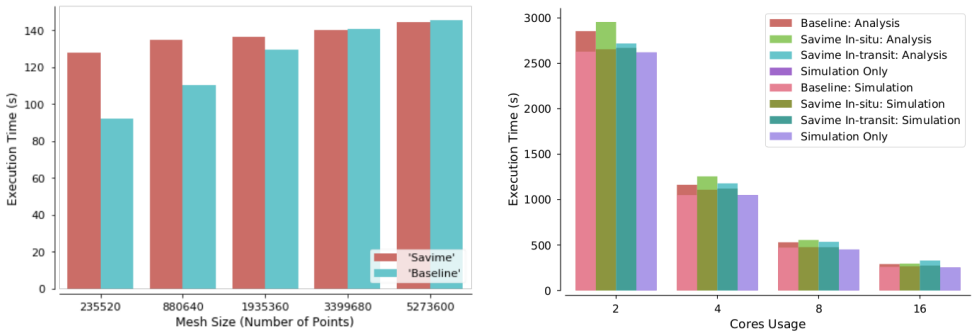

Figure 3. Mesh size x Simulation Execution Time (left) and Simulation and Analysis Scalability (right)

Results show that for very small meshes, which are computationally cheap, the time to load data is more significant, and thus there is some overhead (around 40%) in storing data in SAVIME. However, as meshes get larger, the time taken to compute them increases in a manner that the transfer and loading time is negligible in comparison to the time taken to compute the solution. The transfer and loading time is masked by the dominant process of solving the systems of equation. Therefore, this shows that, for large enough problems, it is possible to load data into SAVIME without compromising the simulation code performance. However, under an impedance mismatch scenario, as observed with SciDB, the loading time would be significant, impacting on the simulation cost.

To evaluate the integration between SAVIME and a simulation tool based on the MHM solver, we use a 2D transport problem over a mesh with 1.9 million points. We run the simulation up to the 100th time step, and store either in SAVIME (approaches 1 and 2) or in an ENSIGHT file (approach 3), data from 50% of all the computed time steps. In both cases, data is always kept in a memory based file system, and never stored on disk.

Once data has been generated, it is analyzed with a PARAVIEW pipeline that carries out the computation of the gradient of the displacement field of the solution. This part is either done by a special operator in SAVIME, or by an AD-HOC MPI Python application using the Catalyst library (baseline), depending on the approach being run. Additionally, we measure the cost of running the simulation without any further analysis, to highlight the *simulation only* cost.

Figure 3 shows the results when running the three approaches, varying the amount of MPI processes spawned or the number of cores used by the MHM simulation code. In this experiment, the simulation code runs and produces its results and then, the simulation output data is read and processed in the analysis step. The plot shows, for each evaluated number of MPI processes, the simulation time and the analysis time as stacked bars. The graph shows that the cost of the analysis process is significantly smaller than the cost for computing the simulation. Moreover, as the *Simulation Only* run shows, the overhead introduced by storing the data in SAVIME or as an ENSIGHT file is negligible, which confirms the claim that SAVIME can be introduced into the simulation process without incurring in extra overhead. From the point of view of the effect of SAVIME on simulation scalability, the storage of data in SAVIME does not impair the capability of the simulation code to scale up to 16 cores.

The IN-TRANSIT approach differs from the other two approaches since it uses a separate computational resource to execute the analysis step. As we see in Figure 3, even when we increase the number of cores the simulation code uses, the analysis time does not change, because the analysis step is done in the fatnode, and always uses the same number of cores (16) independently from the actual number of cores used by the simulation code. The IN-TRANSIT approach illustrates a scenario in which all data is sent to a single computational node and kept in a single SAVIME instance. This approach offers some extra overhead and contention, since all data is sent to a single SAVIME instance, but this enables posterior analysis that transverse the entire dataset without requiring further data transfers.

The SAVIME IN-SITU approach uses the same computational resources used by the simulation code. When we increase the number of cores used by the simulation code, we also increase the numbers of cores used by SAVIME for analysis. The same is true for the baseline approach, meaning that the AD-HOC application also uses the same number of cores the simulation code uses. Even though the SAVIME IN-SITU approach is slightly slower than the baseline approach, we see that both are able to scale similarly. The difference observed in performance between using SAVIME and coding a specialized application becomes less significant as we increase the number of cores being used during the analysis phase. Nevertheless, the small performance loss in this case might be justified by the convenience of using a query language to express analysis instead of the extensive and error prone process of coding other applications to execute analysis.

5. Related Work

The definition of the first array data models and query languages dates back to the works of Baumann [Baumann 1994] and Marathe [Marathe and Salem 1997] [Marathe and Salem 1999]. Since that time, a myriad of systems emerged in order to allow for the storage and analysis of data over multidimensional arrays. Many array DBMSs

have been proposed, the most prominent ones are RasDaMan [Baumann et al. 1997] and, more rencently, SciDB [Cudre-Mauroux et al. 2009].

Due to the fact that ingesting data into these systems is not an easy task, other arrays systems, such as ArrayBridge [Xing et al. 2017] and ChronoDB [Zalipynis 2018] have been developed. ArrayBridge works over SciDB, and gives it the ability to work directly with HDF5 files. ChronoDB works over many file formats in the context of raster geospatial datasets. SAVIME has the similar goal to ease data ingestion, however it does so by enabling seamless data ingestion considering many different array data source by supporting a SDL and a flexible data model, which makes it different from ArrayBridge. SAVIME also offers its own DML, while ChronoDB makes use of existing applications to process data. SAVIME is also different from TileDB [Papadopoulos et al. 2016], which is not exactly a DBMS with a declarative query language, but a library that deals with array data. In addition, SAVIME is a specialized in-memory solution, while the rest of these systems are usually more disk oriented solutions.

6. Conclusion

The adoption of scientific file formats and I/O libraries rather than DBMSs for scientific data analysis is due to a series of problem concerning data representation and data ingestion in current solutions. To mitigate these problems, and to also offer the benefits of declarative array processing in memory, we propose a system called SAVIME. We showed how SAVIME, by implementing the TARS data model, does not impose the huge overhead present in current database solutions for data ingestion, while also being able to take advantage of preexisting data layouts to answer queries efficiently.

We compared SAVIME with SciDB and a baseline approach using the NetCDF platform. The experimental results show that SciDB suffers from the aforementioned problems, not being an ideal alternative and that SAVIME enables faster data ingestion, while maintaining similar performance during window queries execution. We showed that SAVIME can also match the performance of NetCDF for loading and querying dense arrays while providing the benefits of a query language processing layer.

We also assess SAVIME's performance when integrating with simulation code. In this evaluation, we showed that storing data in SAVIME does not impair the scalability of the solver. In addition, results also show that it is possible to retrieve SAVIME data and generate viz files efficiently by using the special purpose visualization operator.

SAVIME is available at github.com/hllustosa/Savime. Future work might focus on the improvement and optimization of current operators, the development of new special purpose TAR operators.

References

Ahrens, J. (2015). Increasing scientific data insights about exascale class simulations under power and storage constraints. *IEEE Computer Graphics and Applications*, 35(2):8–11.

Baumann, P. (1994). Management of multidimensional discrete data. *The VLDB Journal*, 3(4):401–444.

Baumann, P., Furtado, P., Ritsch, R., and Widmann, N. (1997). The rasdaman approach to multidimensional database management. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, SAC '97, pages 166–173, New York, NY, USA. ACM.

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

Center, B. S. (2016). New hpc4e seismic test suite to increase the pace of development of new modelling and imaging technologies. [Online; accessed 01-feb-2018].

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. (2009). A demonstration of scidb: A science-oriented dbms. *Proc. VLDB Endow.*, 2(2):1534–1537.

Gomes, A. T. A., Pereira, W. S., Valentin, F., and Paredes, D. (2017). On the implementation of a scalable simulator for multiscale hybrid-mixed methods. *CoRR*, abs/1703.10435.

Gosink, L., Shalf, J., Stockinger, K., Wu, K., and Bethel, W. (2006). Hdf5-fastquery: Accelerating complex queries on hdf datasets using fast bitmap indices. SSDBM '06, pages 149–158, Washington, DC, USA. IEEE Computer Society.

Group, T. H. (2017). Hdf5 - the hdf group. [Online; accessed 01-feb-2018].

Lustosa, H., Lemus, N., Porto, F., and Valduriez, P. (2017). TARS: An Array Model with Rich Semantics for Multidimensional Data. In *ER FORUM 2017: Conceptual Modeling : Research In Progress*, Valencia, Spain.

Marathe, A. P. and Salem, K. (1997). A language for manipulating arrays. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 46–55, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Marathe, A. P. and Salem, K. (1999). Query processing techniques for arrays. In *ACM SIGMOD Record*, volume 28, pages 323–334. ACM.

Oldfield, R. A., Moreland, K., Fabian, N., and Rogers, D. (2014). Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 83–92, New York, NY, USA. ACM.

Papadopoulos, S., Datta, K., Madden, S., and Mattson, T. (2016). The tiledb array data storage manager. *Proc. VLDB Endow.*, 10(4):349–360.

Paradigm4 (2017). Scidb. [Online; accessed 01-feb-2018].

Unidata (2017). netcdf. [Online; accessed 01-feb-2018].

Xing, H., Floratos, S., Blanas, S., Byna, S., Prabhat, Wu, K., and Brown, P. (2017). ArrayBridge: Interweaving declarative array processing with high-performance computing. *arXiv e-prints*, page arXiv:1702.08327.

Zalipynis, R. A. R. (2018). Chronosdb: Distributed, file based, geospatial array dbms. *Proc. VLDB Endow.*, 11(10):1247–1261.