Database System Support of Simulation Data

Hermano Lustosa
National Laboratory for Scientific Computing
DEXL Lab
Petropolis, RJ, Brazil
hermano@lncc.br

Fabio Porto
National Laboratory for Scientific Computing
DEXL Lab
Petropolis, RJ, Brazil
fporto@lncc.br

Pablo Blanco
National Laboratory for Scientific Computing
HeMoLab
Petropolis, RJ, Brazil
pjblanco@lncc.br

Patrick Valduriez
Inria and LIRMM
Montpellier, France
patrick.valduriez@inria.fr

ABSTRACT

Supported by increasingly efficient HPC infra-structure, numerical simulations are rapidly expanding to fields such as oil and gas, medicine and meteorology. As simulations become more precise and cover longer periods of time, they may produce files with terabytes of data that need to be efficiently analyzed. In this paper, we investigate techniques for managing such data using an array DBMS. We take advantage of multidimensional arrays that nicely models the dimensions and variables used in numerical simulations. However, a naive approach to map simulation data files may lead to sparse arrays, impacting query response time, in particular, when the simulation uses irregular meshes to model its physical domain. We propose efficient techniques to map coordinate values in numerical simulations to evenly distributed cells in array chunks with the use of equi-depth histograms and space-filling curves. We implemented our techniques in SciDB and, through experiments over real-world data, compared them with two other approaches: row-store and column-store DBMS. The results indicate that multidimensional arrays and column-stores are much faster than a traditional row-store system for queries over a larger amount of simulation data. They also help identifying the scenarios where array DBMSs are most efficient, and those where they are outperformed by column-stores.

1. INTRODUCTION

Scientific applications produce ever increasing amounts of data that need to be managed and analyzed. A numerical simulation (simulation for short) is a type of simulation, based on numerical methods, used to represent the evolution of a physical system quantitatively. It has been extensively adopted in many disciplines, such as: astronomy; medicine, biology, and oil and gas. It enables the investigation of phenomena where observed data is not available, such as the state of our universe after the big bang, or even when a complex and dangerous intervention, as in medicine, needs to follow some trials.

A simulation solves a set of differential equations, representing the physics involved in the studied phenomenon and its evolution in space and time. The simulation adopts a spatial representation of the phenomenon domain in the form of a geometrical mesh, composed of triangles or tetrahedrons, that guides the computation through a discretization of the space on its elements such as: vertexes, edges and faces. The solution of the equations finds the values for the predictive variables, such as velocity and pressure, and their variation along the discretized space and time. Moreover, a simulation may combine one dimension (1D) with 3 dimensional (3D) meshes. The latter requires more intensive computation and is left for areas of the simulation domain that demand more precise calculation.

Simulations need to have their initial conditions tuned in order to improve their quality. Aiming at finding the right parameter composition, modelers run parameter sweeps, that iterate the simulation execution with different parameter values. Thus, the combination of precise 3D space model and extended time-step simulation interval, with parameter sweeps leads to large simulation output files.

In this work, we are interested in managing the data produced by simulations. Such data can be viewed as a recurring point cloud that repeats itself along many time steps and simulation trials (see Figure 1). Each point on a cloud is associated to predictive variables, whose values vary on each time step and simulation trial, forming a set of point clouds.

Typically, the simulation output is written to disk as raw files. The latter may be input into a visualization tool, such as Paraview, which allows the modeler to visually assess the simulation quality. Similarly, modelers may apply quantitative analysis on the predictive variable results by writing specific programs to process the simulation files. This method becomes inefficient and hard to scale as the volume of data increases. As an example, our colleagues at HeMoLab have written programs that load in memory large simulation datasets to capture the simulation results in a zone of interest, such as looking for the behavior on a brain

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

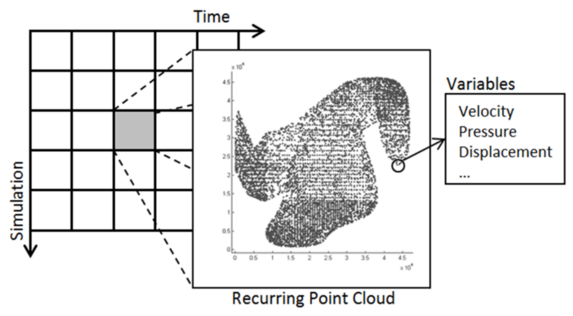
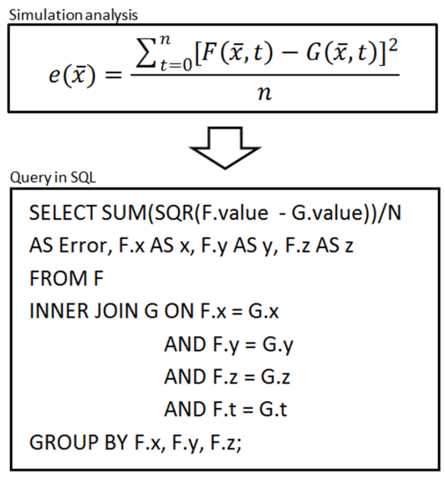


Figure 1: Recurring Point Cloud Data on Different Time Steps and Simulations

aneurysm region [3], which requires a query on a space-time slice of the dataset. Traversing thousands of simulation datasets and comparing results in a particular region and time interval can be efficiently computed in a parallel DBMS, if the parameter sweep computation output is well stored in a distributed system.

Furthermore, due to the lack of isolation between the applications and the data, any eventual changes to the data (i.e. improving the simulation) might require the software to be changed as well. In this context, providing the benefits of a declarative query language for simulation results analysis and a system that scales to support large numerical simulation data will bring state-of-the art data management techniques to this area.

Figure 2 illustrates an analysis used to evaluate the simulation output. It comprises the calculation of the mean squared error between the simulation data (denoted by F) and a reference function (denoted by G), in order to determine how much the resulting predictive values deviate from a pre-calculated reference value in every point. If the data is stored in a DBMS, this analysis can be expressed by a declarative (SQL-like) query, instead of requiring additional coding of another application.



$$e(\bar{x}) = \frac{\sum_{t=0}^{n}[F(\bar{x},t) - G(\bar{x},t)]^2}{n}$$

```
SELECT SUM(SQR(F.value  - G.value))/N
AS Error, F.x AS x, F.y AS y, F.z AS z
FROM F
INNER JOIN G ON F.x = G.x
              AND F.y = G.y
              AND F.z = G.z
              AND F.t = G.t
GROUP BY F.x, F.y, F.z;
```

Figure 2: Mean Squared Error Analysis Example

Traditional row-store DBMSs, usually tailored for OLTP workloads, are considered inadequate for the analytical workload required by scientific applications [18]. Stonebreaker has pointed [19] the inability of current relational DBMS products to satisfy the requirements of a wide range of domains, and argues for specific database engines for different kinds of data centric applications with distinct data models.

Specific data models can offer not only a more convenient representation but may improve query performance through more adequate algorithms. An array DBMS, such as SciDB [2], offers an interesting multidimensional model to represent simulation data, as it leads to a very intuitive mapping of the geometrical portion of simulation data to dimensions and cells in the array. Specifically, the space-time domain dimensions and the simulation trials are mapped to array dimensions, while predictive variable values are mapped to cells of the multidimensional array.

However, some data characteristics might pose difficulties for an array representation. Unstructured meshes, designed to represent complex physical domains, have an irregular distribution of points in space. Mapping such irregular distribution of points onto arrays can produce a sparse multidimensional cube, and incur into performance penalties during array transfer from disk into memory. As our experiments show, different mappings may produce variations in performance of more than 700%. For this reason, we investigate different techniques for managing the data produced by simulations, such as those coming from HeMoLab, by using an array DBMS.

Moreover, recent advances in in-memory column-stores for analytical workloads in scientific data [4, 10], such as MonetDB [9], may also switch the balance from a choice based on data representation to an efficient relational DBMS. Therefore, we implemented our techniques in the SciDB array DBMS and compared them, through experiments over real-world data, with two other techniques based on PostgreSQL (regular row-store RDBMS) and MonetDB (column-store RDBMS). The experiments are carried out according to a benchmark composed by range queries and real life analysis used for improving the simulation.

The experimental results indicate that whenever the complete dataset fits in memory, MonetDB outperforms the other techniques. For large datasets, SciDB equipped with our techniques shows better results.

The remainder of this paper is organized as follows. Section 2 describes the cardiovascular system simulation, developed at LNCC, that motivated this work. Section 3 presents the multidimensional array data model. Section 4 describes our techniques to represent simulation data using an array DBMS. Section 5 describes the representation of simulation data using a relational DBMS. Section 6 presents our experimental evaluation. Section 7 discusses related work. Finally, Section 8 concludes.

2. NUMERICAL SIMULATION DATA

In order to run a simulation, one must first build the mathematical model that captures the behavior of the studied phenomenon. Commonly, mathematical models are expressed by differential equations that are solved by a variety of numerical methods, such as FDM (Finite Difference Method) and FEM (Finite Element Method). The software component that solves a set of differential equations applying a particular numerical method is called a solver. The adopted numerical method might require the discretization of the physical domain in a form of a mesh.

Meshes have topological and geometrical representations. Geometrical aspects are related to shapes, sizes and absolute positions of their elements, such as vertexes or points. The topology representation captures the relationships among elements, like their neighborhood or adjacency, without considering their position in time and space.

The distinction between these two concepts, i.e. mesh geometry and topology, is important in the context of our work. We believe that, although the topological representation might require a more elaborated solution [8], both arrays and relations can be sufficient to represent the geometry of a mesh. Indeed, important queries can evaluate the simulation output by assessing the values of predictive variables associated to mesh points in time, irrespectively of topological constraints. Thus, in this paper, we constrain our discussions to geometric aspects of meshes.

Vertexes in the geometrical representation of a mesh can be viewed as forming a point cloud. A mesh guides solvers during simulation computation, such that, the predictive variables computed by the model have their values assigned at each vertex point, for each time step. Moreover, runs with different parameters of the same model may use the same geometry mesh as space discretization. The resulting output data, shown in Figure 1, can be viewed as a recurring point cloud appearing at different time steps and simulations.

2.1 Use Case

The HeMoLab group at LNCC has developed computational tools to simulate the human cardiovascular system [7]. The simulation adopts a geometrical representation of the arteries in the human body in the form of a mesh. This representation is used in 1D and 3D simulations, working together as a coupled model [1].

One dimensional (1D) models represent the human cardiovascular system in a simplified form. Main arteries of the human body are modeled as a set of lines. Three dimensional (3D) models are used in order to achieve a higher level of detail about the behavior of the blood flow inside an artery.

Figure 3 shows how the 1D and 3D models are arranged in space. Both 1D and 3D meshes actually contain points in 3D with x, y and z coordinates. However, the 1D model contains a linear representation of the arteries in the entire body, while the 3D model has a much more detailed representation of a single artery. Combined, 1D and 3D meshes have over 90 thousand points, most of them (about 99%) belong to the 3D model and the remaining 1% to the 1D model.

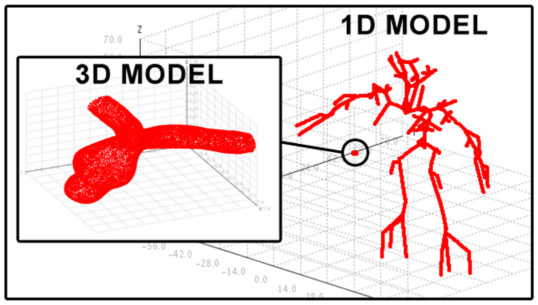


Figure 3: 3D-1D coupled model

The numerical solver, which is in charge of executing the simulation, computes for each position in the mesh, and for many time steps, physical quantities; such as: pressure, blood flow velocity and mesh displacements in the case of dealing with deformable domains. The solver reads a set of input files that describe the parameters of the model and stores the results in raw files, which requires scientists to write specific applications for any kind of analysis needed. In this context, storing simulation data in a DBMS would allow researchers to use a declarative query language to execute analyses more efficiently.

These analyses could be as simple as retrieving data from a subset of simulations or time steps, or even obtaining the result from an aggregation function for a spatial window (range query). In our benchmark we include more complex real life analyses, such as calculating the mean squared error between the data in the simulation and a reference function (see Figure 2), and obtaining the time step required for an attribute to reach its maximum value on each point.

The Human Cardiovascular simulation and the analytical queries mentioned above will form the basis of our benchmark to evaluate strategies for managing point cloud data.

3. MULTIDIMENSIONAL ARRAYS

In this section, we describe the multidimensional array data model, and its implementation in SciDB. We chose to use SciDB as the array DBMS in our evaluation because it directly implements the model instead of just adding array processing capabilities to a relational system. Also, SciDB is a distributed system, tailored to the requirements of some scientific application domains.

Multidimensional arrays are defined by a set of named dimensions. A set of indexes for all dimensions identifies a cell in the array. Cells can have many attributes in which the values are stored just like tuples in a relational DBMS. Cells can be non-existing or empty, and arrays are said to be sparse if they have many empty cells.

SciDB partitions the array into regular tiles, or chunks [2]. Chunks are physical units of distribution of arrays on computer nodes. The array is partitioned by ranges of indexes of fixed size on each dimension. The product of the number of partitions on each dimension gives the number of chunks the array will be split into.

Figure 4 shows a bi-dimensional array with dimensions i and j. The chunk size for dimension i is 4 while its total length is 16. The chunk size for dimension j is 5, and its total length is 20. This configuration results in a total of 16 chunks. SciDB allocates chunks to worker nodes that are responsible for maintaining portions of the arrays.

We believe that arrays are a convenient model for representing simulation data. Different experiments, time steps and the spatial coordinate system can be mapped to dimensions in the array, while the physical quantities can be stored as attributes. The DBMS takes advantage of the array structure in order to answer range queries for dimension values. Dimensions work like multivalued indexes in a relational DBMS, which means there is a fast access path to a set of cells when they are specified by index ranges. In practice, portions of the array for each dimension or combination of dimensions can be retrieved efficiently by the DBMS.

Regular tiling or chunk partitioning also plays an important role in improving performance. Storing subarrays on disk as chunks yields an efficient representation of data, i.e.,

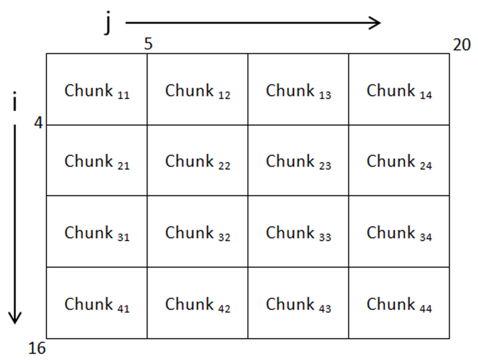


Figure 4: Chunk configuration

cells that are close to each other on the array, will likely be on the same portion of the disk, since chunks are the basic I/O unit in SciDB [2]. The advantage of this representation is that when a subarray is referenced by a query, the data will probably be sitting on a limited amount of disk pages where the chunks containing the data of interest are stored, instead of being scattered across many different pages. Obviously, a reduced amount of disk pages read from disk results in a faster query processing time.

4. ARRAY REPRESENTATION

There is an intuitive simplicity regarding the mapping of simulation data to multidimensional arrays. However, a naive mapping of coordinate values onto indexes of an array may lead to inefficient access due to the following data characteristics:

- the subjacent geometry mesh coordinate system may adopt a non-integer representation.
- the physical geometry of a phenomenon may be modeled by an irregular mesh.

Considering the mapping of dimensions involved in a simulation, the time-step and the simulation trial dimensions are both integers and do not include holes in their series, which leads to a more straightforward mapping to indexes in the array. Conversely, a mesh whose points are referenced by non-integer coordinates, when naively mapped onto integer array indexes, produces huge gaps between neighboring cells within the array. In this scenario, a space preserving mapping strategy would induce huge variable density chunks due to an uneven distribution of points.

Uneven distributed data can directly impact on the cells distribution per chunk. As the latter is used as I/O access unit in SciDB, the performance during query processing is affected by this irregular data distribution. In the following sections we describe three different techniques for mapping the coordinate values to integer indexes. We start by formalizing the terminology we will use.

- Definition 1. A geometry mesh is defined as $M = (P, E)$, such that $P = \{p_1, p_2, \cdots, p_n\}$ is a set of vertexes and $E \subseteq P \times P$. Moreover, each vertex $p_i = (id, X_n)$, in which X_n is a n dimensional coordinate index and id is an identifier.
- Definition 2. An array A is defined as $A = (D, C)$, such as $D =< D_1, D_2, \cdots, D_m >$ is a list of m dimensions and $C = \{c_1, c_2, \cdots, c_v\}$ is a set of v attributes. Each dimension D_i is indexed from 1 to an integer $maxindex$, in one increments. An index $i =< i_1, i_2, \cdots, i_k >$, with $k \leq m$, and $1 \leq i_j \leq maxindex_j$, for $1 \leq j \leq k$, defines a subarray of A.

To deal with the problems caused by the irregularity of data distribution, we classify the mapping techniques into two groups: space preserving and non-space preserving.

4.1 Precision Elimination Method

We denote by precision elimination, (P.E. for short), the naive method for mapping coordinate values with finite precision to integer values. It consists of multiplying each different coordinate value by 10^{Pr}, where Pr is the precision in which the coordinate values are specified.

Suppose a mesh with dimensions $[X_1,\ X_2,\ \ldots, X_n]$ containing the point $p(x_1,\ x_2,\ \ldots, x_n)$ specified with Pr decimal places of precision. The array used in this case also contains the dimensions $[D_1,\ D_2,\ \ldots, D_n]$, and the data related to the point $p(x_1,\ x_2,\ \ldots, x_n)$ will be stored on the cell identified by the indexes $(x'_1,\ x'_2,\ \ldots, x'_n)$ defined as $x'_i = x_i * 10^{Pr}$.

The method is efficient and enables spatial window queries to be expressed easily. Consider a region of interest R in the original mesh representation, specified by their bordering points: lower bounds $(l_{r1}, l_{r2}, \ldots, l_{rn})$ and upper bounds $(u_{r1}, u_{r2}, ..., u_{rn})$, where l_{ri}/u_{ri} is the lower/upper bounds for all i dimension, $1 \leq i \leq n$. The subarray S containing only the cells related to the points within R can be defined with the lower bounds $l_{si} = l_{ri} * 10^{Pr}$ and the upper bounds $u_{si} = u_{ri} * 10^{Pr}$, for all dimensions i, $1 \leq i \leq n$.

However, this method has some drawbacks. Multiplying the coordinate values in order to eliminate the precision may be undesirable in some applications. The index values will need to be truncated in case the coordinate values are specified with many decimal places. It might lead to information loss if the original mesh representation is not explicitly stored somewhere else.

In addition, multiplying coordinate values is equivalent to scaling up the points. The higher the precision, the higher the scale factor is, and the farther apart adjacent cells will be in the array. The highly sparse array obtained can be hard to deal with, as the cells are distant from each other, and unevenly distributed throughout array regions. Defining the partitioning in this case is challenging, and some chunks will likely be excessively filled while other chunks will contain just a few cells.

As we show in our experimental evaluation, unbalanced sparse chunks can negatively affect performance. For this reason, we propose the following two methods in order to avoid the sparse and unbalanced data distribution.

4.2 Non Space Preserving Methods

The naive mapping described above is a space preserving method, i.e., the general form of the mesh is preserved in the final array representation. However, in order to cope with the irregular data distribution one must rearrange the points altering the form of the mesh. We denote by non space preserving methods, the techniques in which the alteration of coordinate values does not keep the original spatial distribution of mesh points.

The first step for such methods is to create a more compact data representation. Dead spaces between coordinate values are removed without changing the relative positions of the points to each other. This process reduces the spacing between adjacent points and creates a less sparse array.

A sorted list L_i is created for each $X_1,\ X_2,\ \dots,X_n$ dimensions, with $1 \leq i \leq n$. Each list L_i contains the coordinate values $a_{i0},\ a_{i1},\ a_{i2},\ \dots,a_{im}$ that appear in some point on the mesh for the dimension i. Every referenced coordinate value a_{ij} has its position in the compacted representation given by a function $L_i(a_{ij})$. A point $p(x_1,\ x_2,\ \dots,\ x_n)$ on the mesh is replaced by $p(l_1(x_1),\ l_2(x_2),\ \dots,\ l_n(x_n))$. This means that the existing space among points in the mesh have been removed, thus producing a more compact array representation.

Figure 5.2 shows a resulting compact representation. Note that although the shape of the mesh is deformed, because the spacing between adjacent coordinate values has been removed, the relative position of the points is maintained. This property is important, since the compact form must allow spatial window querying on the original representation to be translated into the new representation. Given a window R enclosing a set of points in the mesh, it must be possible to define a window R' on the compact representation that encloses the exact same set of related points.

This statement can be verified as a coordinate value that appears on many points will be consistently replaced by the same integer value wherever it appears. Not only this, but the integer values are attributed in a fashion that maintains the ordering of the coordinate values. As all points are changed in the same manner, they maintain their relative position to each other, making it possible to translate spatial window queries.

Equi-depth Histogram Method

The equi-depth histogram based partitioning method is a non space preserving mapping strategy. We adopt it in oder to minimize the irregularity in data distribution and generate more balanced chunks. Equi-depth histograms are similar to equi-width histograms. The main difference is that regular histograms have bins for equally sized portions of the data domain, but containing a different number of occurrences on each one, while equi-depth histograms have bins for portions of the domain with variable sizes, but containing roughly the same amount of occurrences on each one. Thus, equi-depth histograms help splitting the mesh into regions containing a close number of points on each one. These regions will be used to define the chunk partitioning scheme of the multidimensional array.

Figure 5 depicts the steps executed to obtain the mapping in a 2D projection for the HeMoLab's 3D model mesh. In Figure 5.1, we have the original points in the mesh projected in a 2D surface. Figure 5.2 shows the resulting representation after the first step comprising the removal of dead spaces.

A set of equi-depth histograms, one for each dimension, is created in the following step (Figure 5.3). The combination of histograms for each dimension creates variable sized regions, which enclose a variable amount of points. The difference in the amount of points contained in the most populated and the least populated regions tends to decrease as we increase the granularity of the partitions (decrease the bin sizes for the histograms). Thus, we can vary the sizes of

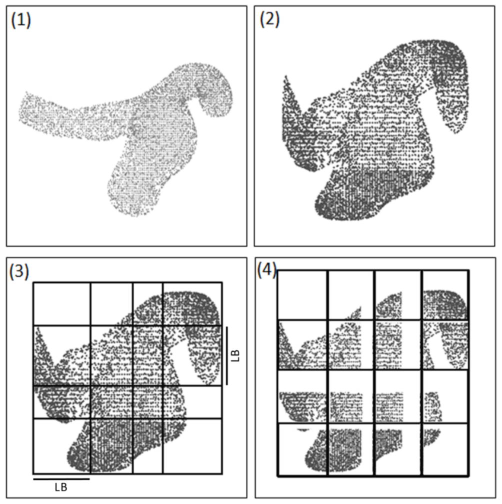


Figure 5: Equi-depth Histogram Based Partitioning Method

the bins and recreate the histograms until the configuration that minimizes this difference is found.

The regions created by the combination of the histograms will be used as the base for the definition of the chunk configuration. Ideally, chunks in SciDB should occupy a few megabytes on disk [2]. In other words, chunks should contain between 500 thousand to 1 million cells for arrays with 8-bit double precision floating point attributes. Therefore, the amount of points on most regions should be large enough so the chunks have enough cells. The higher the granularity, the smaller is the amount of points on each region. Thus, we search for the more granular partitioning that still enables to define sufficiently populated chunks.

As stated previously, the regions depicted in Figure 5.3 have variable sizes, so they cannot be directly used as the chunk configuration for an array in SciDB, due to the fact that it only allows regular tiling. To overcome this limitation, points are translated so they become distributed in equally sized regions of space.

Figure 5.4 depicts the mesh after the points have been translated. The process to define the new position for a point is as follows. Every histogram H_i has a set of bins $B_{i0},\ B_{i1},\ \dots,B_{i1}$ for every dimension i. A coordinate value x_i on a point p of the mesh is contained in the range of values associated to bin B_{ij} (the j^{th} bin for the histogram associated with the dimension i). The index j of a bin B_{ij} associated to a coordinate value x_i is denoted by $B_i(x_i)$. We define as $IB_i(x_i)$ the initial coordinate of a bin, i.e., the smallest value contained in a bin B_i that also contains the coordinate value x_i.

The size of the new equally sized regions in the dimension i will be given by the largest bins LB_i for the histograms H_i. Note that in Figure 5.4, the new regions of the space have the length in dimension i of the largest bin for the histogram H_i. Every point $p(x_0,\ x_1,\ \dots,\ x_n)$ is replaced by a new point $p'(x'_0,\ x'_1,\ \dots,\ x'_n)$. The new coordinate values for p' are given by the formula: $x'_i\ =\ LB_i * B_i(x_i) + (x_i - IB_i(x_i))$ for $1 \leq i \leq n$.

The array used to store data mapped with the use of this method has the dimensions $[X_1,\ X_2,\ \dots,X_n]$ and the coordinate values of every p' will be used as indexes to allocate the points in the array. The LB_i values are used to define the chunk sizes for each dimension. It is important to note that even after moving the points, we still are able to translate spatial window queries, as long as the mapping between the original coordinate values and the indexes used for the points is explicitly stored.

The final array representation is still sparse, but the cells will be distributed much more evenly among the chunks, which is the main advantage of this technique. This method not only maps the coordinate values into indexes, but also helps defining the chunk configuration, which is an important aspect in designing multidimensional arrays in SciDB.

However, the process for creating this mapping might be costly as we potentially need to create many histograms and evaluate the partitions given by their combinations. Additionally, every spatial window query specified based on the original mesh representation must be translated into a region in the array.

This translation can be described as follows. Suppose a region of interest R in the original mesh representation, specified as lower bounds $(l_{r1}, l_{r2}, \dots, l_{rn})$ and upper bounds $(u_{r1}, u_{r2}, ..., u_{rn})$. We define a function $M_i(xo_i) \rightarrow x'_i$ that maps the original real coordinate value to the corresponding index stored in the array. The values for this function are pre-calculated and explicitly stored, thus, evaluating the function requires a lookup in the structure used to store the mapping. The subarray S containing only the cells related to the points within R can be defined with the lower bounds $l_{si} = M_i(l_{ri})$ and the upper bounds $u_{si} = M_i(u_{ri})$ for every dimension i.

We consider the use of this technique under two main assumptions about the underlying mesh:

- It has low dimensionality (usually 2D or 3D).
- It is a relatively small portion of the entire dataset.

In our use case, the main dataset contains data from 20 simulations, each one possessing 240 time steps. The number of data records is equal to the product of the number of points in the mesh (close to 90 thousand), the number of simulations and time steps, which amounts to over 430 million. Since the number of points in the mesh is much smaller than the amount of records in the entire dataset, maintaining and querying the mapping, that is as large as the amount of points in the mesh, should be much cheaper than maintaining and querying the entire dataset.

Space-Filling Curve Method

Although a more balanced data distribution is achieved with the use of the second method, the resulting array is still sparse and contains irregularly filled chunks. Our third method to map coordinate values to array indexes relies on the use of space-filling curves to create a dense data representation.

Space-filling curves can be understood as a path that defines a order in which points are visited. They enable the mapping of multidimensional points into a single scalar value or code, that depends on the position of the point in the curve. Space-filling curves are defined in such manner that adjacent codes will be attributed to points that are relatively close to each other in space, because the next point the curve passes by is likely to be on the neighborhood of the last visited point.

This characteristic of space-filling curves makes them useful in indexing points, because it helps maintaining a coherent data representation. With this method, the original mesh dimensions $X_1, X_2, \dots, X_n$ are represented by a single dimension D in the array. A single index is attributed for points according to a space-filling curve, in such a manner that adjacent indexes are given to points that are close in space.

Spatial window queries, previously defined as a set of upper and lower bounds for different dimensions, now have to be expressed with multiples ranges of values for a single dimension. The points enclosed in a window R of the original mesh representation will be distributed in a set of continuous intervals of code values $[L_1, U_1], [L_2, U_2], ..., [L_n, U_n]$. The best, and very unlikely, scenario is when all points are contained in a single interval. The worst scenario is when the number of intervals is equal to the number of points within R, which means that the codes are scattered in non-contiguous intervals.

In a multidimensional array, fewer intervals for a range query means data will probably be sitting on a limited number of chunks. Fewer chunks, means faster retrieval time, and faster query processing time. It is common that just a portion of the data contained in the chunk is necessary to answer a query. Therefore, minimizing the amount of chunks read from disk to memory is also going to minimize the amount of unnecessary data loaded along the data of interest.

The Hilbert Space-Filling curve has very interesting clustering properties [13] that help creating an efficient data representation in a single dimension. We use compact Hilbert indexes [6] to allocate the positions of the points in the array. Figure 6 shows a mesh with vertexes colored by their respective compact Hilbert indexes. Note that large clusters of points in a same region of the mesh have similar colors, indicating their respective indexes have also close values.

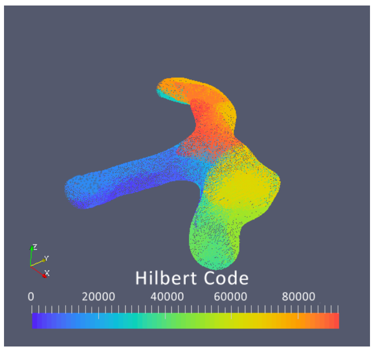


Figure 6: 3D Model Mesh Points Colored by Their Hilbert Codes

The process of attributing indexes to points goes as follows. First, we create a compact integer representation of the mesh as explained in section 4.2. Then, for every point $p(x_0, x_1, \dots, x_n)$ in the compact representation, we obtain $chi(p)$, that is the compact Hilbert index for p. All points p are ordered according to their $chi(p)$ value and are added to a list L_{chi}. Finally, the index for the point in the array is attributed according to the order of the points in L_{chi}. This is done because not necessarily the $chi(p)$ values will be contiguous. There may be intervals of indexes that are not attributed to any point p. These intervals would create an undesirable sparse distribution, so they are removed once the indexes are attributed according to the order in which the points appear in L_{chi}.

In order to enable querying and rebuilding the original space distribution, an auxiliary one-dimensional array must be created, containing a dimension D' and an attribute set $[Xa_0, Xa_1, ..., Xa_n]$ for every dimension in the original mesh representation where each attribute holds values for a given mesh dimension. This array contains the mapping between the original coordinate values representing a point and the associated indexes. Spatial window queries are expressed with joins between the main array and the auxiliary array, which is slightly less efficient than directly expressing the boundaries of the spatial window.

Even though range queries might not be as efficient in this representation, the overall performance of queries that represent more elaborated data analyses are expected to be improved for two main reasons. The resulting array is dense, and the data distribution per chunk is perfectly balanced. The array has a smaller dimensionality that might affect positively the performance for executing array operations that are the building blocks of complex queries.

5. RELATIONAL REPRESENTATION

As expected, one may argue using relational DBMSs to represent and store simulation data. This section briefly describes a relational structure used to represent simulation data in relational DBMSs. In our evaluation, we use both a row-store DBMS and a column-store DBMS, respectively PostgreSQL and MonetDB.

Schemas in both systems are roughly the same. The data is stored in a single table containing attributes to represent the physical quantities, the mesh coordinate system, the time and the simulation identifiers. A single table offers a simple representation of data and does not require costly joins in order to answer range queries.

Different from the array representation that requires a mapping between coordinate values to integer indexes, the relational representation naturally allows the original mesh coordinate values to be directly stored as double precision floating point numbers in columns of the relation. Any range query can be expressed as a logical predicate using the SQL operator BETWEEN for defining intervals and clauses to implement analytical aggregations.

The primary index for the relation is composed by the set of attributes containing the combination of the coordinate values for the points in the mesh and the time and simulation identifiers. This combination of values uniquely determines another set of attributes that represent the physical quantities, such as pressure, velocity and displacement related to the points in a instant in time.

Regarding the physical model, we considered the construction of indexes to improve query performance, when suited. Our preliminary tests showed that independent secondary indexes have better performance for the workload under consideration in our benchmark. Thus, we use secondary indexes for every attribute on the primary key, enabling the DBMS to decide which index or combination of indexes will be more efficient. The definition of secondary indexes is necessary only for PostgreSQL, since MonetDB implements the Imprints index structure [17] that is created during query processing.

6. EXPERIMENTAL EVALUATION

In this section we present our experiment results. We start describing the computational environment, followed by the benchmark composed of representative simulation quantitative analysis. Next, we discuss the performance of the proposed methods for representing simulation data in SciDB and finally we compare SciDB, using the proposed methods, with relational systems.

6.1 Environment

The benchmark was executed in a virtual machine environment (single core, 12GB of memory and 250 GB of storage), with Ubuntu Server 14.04 LTS as both the guest and the host system. The choice of a single core VM aimed at reducing uncontrolled internal system parallelism that could affect results. We used versions 14.12, 5.0 and 9.4 for SciDB, MonetDB and PostgreSQL respectively. For the comparative tests, all DBMSs were running with a single instance, while for the distributed tests we evaluate SciDB alone using 4 and 8 instances distributed respectively in 2 and 4 nodes.

6.2 The Benchmark

The benchmark is designed considering typical quantitative analysis performed while evaluating the quality of simulation or supporting the interpretation of the modeled phenomenon. It includes the data from the cardiovascular system simulation implemented into SciDB, MonetDB and PostgreSQL and four set of analytical queries.

6.2.1 Data Definition

We defined the schemas in SciDB using the DDL for the AQL (Array Query Language). AQL is one of the two languages supported in SciDB. The following CREATE ARRAY commands define the schemas for arrays that implement the P.E., Histograms Based Partitioning and Hilbert Space-filling Curve methods respectively (main and auxiliary mapping arrays).

```
CREATE ARRAY PrecisionElimination <pressure:double> [x
    =0:*,350000,0, y=0:*,350000,0, z=0:*,350000,0,
    time_step=0:*,50,0, simulation_number=1:*,4,0];

CREATE ARRAY Histogram <pressure:double> [x=0:*,16091,0, y
    =0:*,24565,0, z=0:*,25186,0, time_step=0:*,50,0,
    simulation_number=1:* 4,0];

CREATE ARRAY HistogramAux <x:double, y:double, z:double,
    x_map:double, y_map:double, z_map:double> [i
    :*,1000000,0];

CREATE ARRAY Hilbert <pressure:double> [hilbertCode
    =0:*,5000,0, time_step=0:*,50,0, simulation_number
    =1:*,4,0];

CREATE ARRAY HilbertAux <x:double, y:double, z:double> [
    hilbertCode=0:*,5000,0];
```

Attributes are listed between angle brackets and dimensions are specified between square brackets. Each dimension is defined with a name and boundaries, followed by two numbers representing the chunk size and the overlap between chunks. Chunks sizes are defined according to each strategy. Larger chunks are specified for the very sparse PrecisionElimination array, while more compact chunks are defined by the Histogram Based Partitioning strategy for the Histogram array. The Hilbert array contains dense chunks whose sizes on each dimension are defined in such manner that the total amount of cells is 1 million (5000 * 50 * 4).

Tables for the relational schema are defined with the following CREATE TABLE command. The only difference between the schemas is the addition of secondary indexes in PostgreSQL.

The dataset consists of 20 simulations comprising 240 time steps and over 90 thousand points for the 3D-1D mesh. The total amount of records (rows and cells) is bigger than 430 million.

```
CREATE TABLE [postgreSQLTable/monetDBTable](simulation
      BIGINT, time_step BIGINT, x DOUBLE, y DOUBLE, z
      DOUBLE, pressure DOUBLE, PRIMARY KEY (simulation,
      time_step, x, y, z));
```

6.2.2 *Queries*

Four sets of queries are used in our benchmark. Two sets are composed of spatial window queries (range queries) evaluating the results of an aggregation function on a region of space. The regions of space selected are the complete 1D model, and a portion of the 3D model representing an aneurysm formed on the artery.

The 1D model region of the mesh is very sparse and has a relatively small amount of points compared to the entire mesh. The aneurysm region contains much more points, that are arranged in a denser region of space. These two opposite regions of the mesh are chosen to evaluate the performance of each strategy in different scenarios.

We denote by Hemolab 1 the third set of queries in our benchmark. Hemolab 1 comprises the calculation of the mean squared error between the data in the simulation and a reference function. The values for this reference function are precalculated and stored in the database. The final result for this query is a single value for each point for every simulation under consideration. The value is the mean squared error between the values on the simulation and the reference function on each point of the mesh. This query evaluates the execution of a join operator based only on dimension values.

The hemolab 1 query in AFL and SQL is:

```
AGGREGATE(
        APPLY(
              CROSS_JOIN(
                    BETWEEN(PROJECT([ARRAY], pressure), null,
                            null, [SIM_NUM], null, null, [SIM_NUM])
                             as a, [REFERENCE_ARRAY] as b, a.
                            time_step, b.time_step, a.space, b.
                            space
              )
              ,prod, (a.pressure - b.pressure)*(a.pressure - b
                     .pressure)
        )
        ,sum(prod), simulation_number, space
)

SELECT SUM((a.pressure- b.pressure)*(a.pressure- b.
        pressure))
         FROM (SELECT pressure, simulation, x, y, z,
                time_step
                      FROM [TABLE] WHERE simulation = [SIM_NUM])
                              AS a
         INNER JOIN
                (SELECT pressure, simulation, x, y, z,
                      time_step
                      FROM [REFERENCE_TABLE]) AS b
         ON a.x = b.x AND a.y = b.y AND a.z = b.z AND a.
                time_step = b.time_step GROUP BY a.x, a.y, a.z,
                 a.simulation;"
```

The fourth set of queries is denoted by Hemolab 2. This analysis is executed in order to obtain the time required for attributes in the simulation to reach their maximum values for the first time. Maximum values of the attributes on every point are calculated with use of an aggregation function. After that, values are joined with the original dataset so they can be paired up with the time steps in which they occur. Once we have all time steps in which the attribute value is at its peak, we simply return the lowest time step, i. e., the first to present the maximum value. The result for this query is also a single value for every point, that represents the time necessary for the attribute to reach its maximum value. This query contains a join based on attribute values, and it evaluates performance for queries that need to perform this kind of operation.

The hemolab 2 query in AFL and SQL is:

```
CROSS_JOIN(
     AGGREGATE(
            FILTER(
                CROSS_JOIN(
                      APPLY(BETWEEN(PROJECT([ARRAY], pressure),
                             null, null, 1, null, null, 1), time
                            , time_step) AS a,
                      AGGREGATE(BETWEEN([ARRAY], null, null, 1,
                             null, null, 1), max(pressure) AS
                           max, simulation_number, space) AS b,
                      a.simulation_number, b.simulation_number,
                             a.space, b.space
                ),
                a.pressure = a.max
            ),
            min(a.time), simulation_number, space
     ) AS a,
     PROJECT(APPLY([ARRAY_MESH], norm, SQRT(x*x + y*y + z*z
          )), norm) AS b,
     a.space, b.space
)

SELECT A.simulation, a.x, a.y, a.z, min(a.time_step), sqrt
      (a.x*a.x + a.y*a.y + a.z*a.z)
FROM (SELECT simulation, x, y, z, time_step, pressure
           FROM [TABLE] WHERE simulation <= [SIM_NUM]) AS a
       INNER JOIN
       (SELECT simulation, x, y, z, max(pressure) AS max
           FROM [TABLE] WHERE simulation <= [SIM_NUM] GROUP
                  BY simulation, x, y, z ) AS b
       ON a.simulation = b.simulation AND a.pressure = b.
              max AND a.x = b.x AND a.y = b.y AND a.z = b.z
              GROUP BY A.simulation, a.x, a.y, a.z;""
```

6.3 Experiments with SciDB

This section presents the experimental results for different methods of representing simulation data in SciDB arrays. The charts contain the average execution time for 30 runs. Queries were executed with three configurations, containing 1, 4 and 8 SciDB instances. Tags below the horizontal axis of the charts in this section indicate the number of instances related to the execution time. All queries in this section evaluate data from all time steps and simulations. This is true even for spatial range queries.

Figure 7 shows the results for the 1D model range query. The 1D model mesh contains a relatively small portion of the points. In this case, the sparse representation generated by the P.E. method was more efficient. The irregular data distribution in this case creates a set of almost empty chunks containing data for the points on the 1D model only. These chunks do not contain the dense data for the 3D model, and thus, the DBMS does not read and process unnecessary data.

The exact opposite happens for the Histogram Base Partitioning method. In this case, the more balanced chunks contain data from both 3D and 1D models, which means SciDB needs to read chunks filled with data unnecessary to answer the query, making the total execution time many times higher.

Figure 8 shows the results for the range query evaluating the region encompassing the aneurysm in artery. This is a subset of the data from the 3D model only. The results differ from those of Figure 7. In general, there is no chunk configuration that is good for all range queries. Ideally, a dense and well distributed configuration, such as the one provided by the Hilbert space-filling curve method, would minimize the variations between distinct queries, improving the overall processing time for different queries.

The irregular precision elimination method is two times slower than other strategies for the aneurysm range query. Since the entire 3D model is distributed in a small portion of space, the related points are also sitting on a limited amount of chunks. In order to retrieve data for just a portion of the

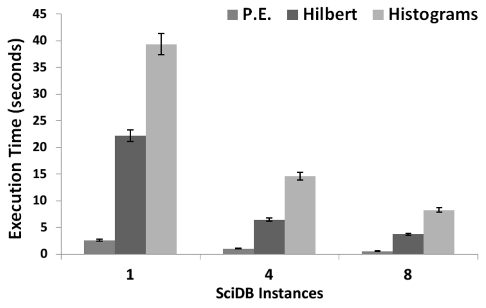


Figure 7: 1D Model Range Query Results

3D model enclosing the aneurysm, the system needs to read chunks filled with cells for the entire 3D model. The Histogram Based Partitioning method has the best performance in this case, because the granular and balanced distribution of 3D model points enabled SciDB to access chunks that encompass the data of interest more closely, thus diminishing the need to read from disk regions of the array that do not intersect with the range query.

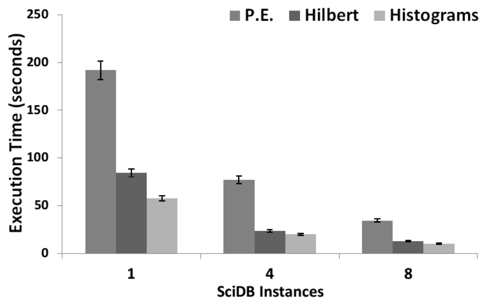


Figure 8: Aneurysm Range Query Results

Figures 9 and 10 show the results for queries Hemolab 1 and Hemolab 2 respectively. The Hilbert Space-filling curve method offers the best option among the multidimensional array representations. For instance, it is up to 60% faster for Hemolab 2 than the Precision Elimination method and the Histograms Base Partitioning method. The dense and balanced data distribution together with the lower dimensionality enable analyses to be executed more efficiently.

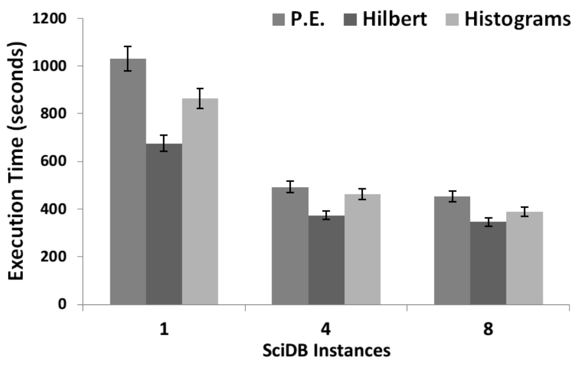


Figure 9: Hemolab1 Query Results

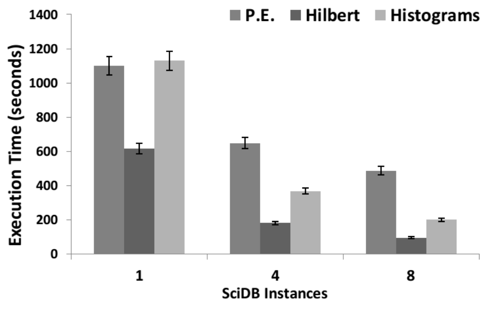


Figure 10: Hemolab2 Query Results

6.4 SciDB vs MonetDB vs PostgreSQL

In this section we compare our techniques with array representation in SciDB with relational representation in MonetDB and PostgreSQL. The charts show the execution time of different runs using as input variations on the number of time steps and number of run simulations. The horizontal axis of charts shows the results for the same analyses repeated by taking into consideration different numbers of time steps and simulations. The labels of columns indicate the number of time steps and simulations. Numbers preceded by S indicate the amount of simulations under consideration, and the number preceded by T the amount of time steps under consideration.

Figure 11 shows the time required to load data into the DBMSs in each case. A COPY command was used to load the entire dataset at once for relational systems, and specific array operators, such as LOAD and REDIMENSION, were used to load the data into SciDB.

MonetDB is the most efficient because all it does is to append data to files for each column on the relation. Both PostgreSQL and SciDB need to execute costly operations during data loading. Operations responsible for ordering data and dividing it into chunks take a longer time than just adding records to data files. Also, PostgreSQL needs to maintain the secondary indexes for each attribute on the primary key, which slows down the row insertions.

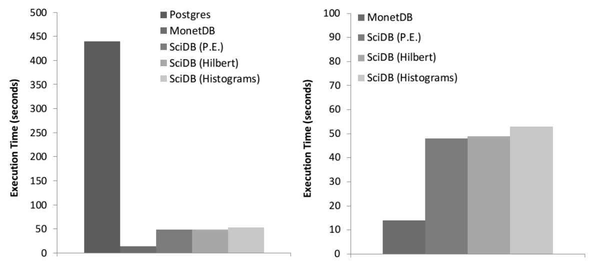


Figure 11: Loading Time

Figure 12 shows the results for the range queries evaluating data of the 1D model in a distributed settings. The results observed in the sequential scenario discussed in Section 6.3 remain valid. In this context, the PE method is again more efficient and the Histogram Base Partitioning method reads unnecessary data.

PostgreSQL shows better results than most other methods, except P.E., as the amount of data under consideration is very small, and the system is able to use a combination

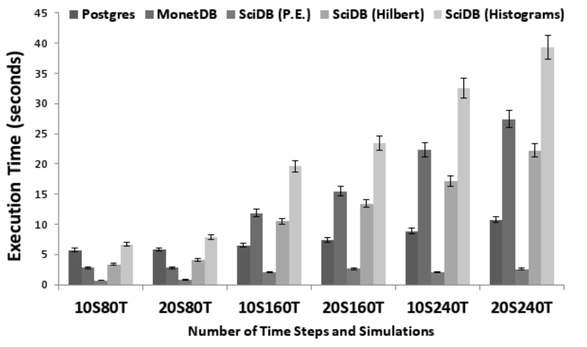


Figure 12: 1D Model Range Query Results

of indexes to retrieve a minimum amount of disk pages to answer the query. However, as the next results show, PostgreSQL is not able to maintain fast execution time when we increase the amount of data.

Figure 13 shows the result obtained for the range query evaluating a region of the 3D model enclosing an aneurysm. For this query, and all following analyses, PostgreSQL has the worst performance by a great margin. Both MonetDB and SciDB are tailored for analytical workloads, and offer a much better performance for analyses containing a larger amount of data.

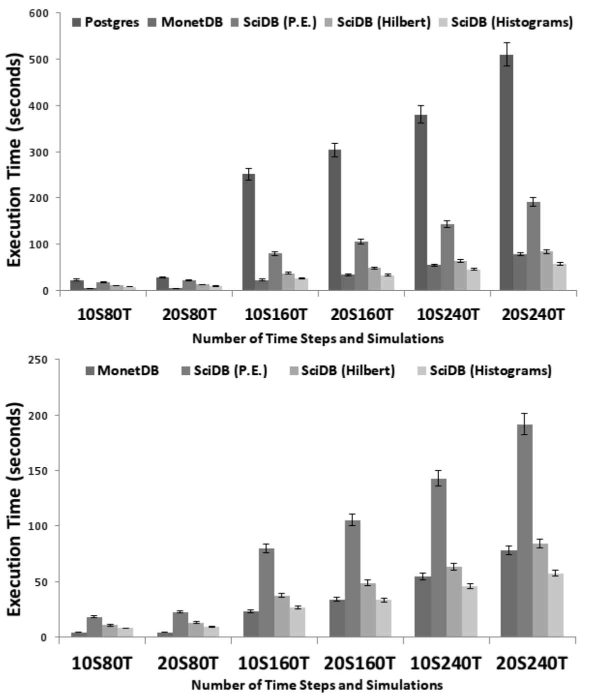


Figure 13: 3D Model Aneurysm Range Query Results

MonetDB is very efficient in the second analysis. Even though no index is previously defined, it is able to execute the analyses almost as rapidly as SciDB with the Histogram Based Partitioning Method. Another point to highlight is that the Hilbert Space-filling curve method obtained intermediate results in both range queries. The need to execute a join operation slightly increases execution time, but the dense array representation with lower dimensionality makes it still a good alternative, even for range queries.

Figures 14 and 15 show the results for the queries Hemolab 1 and Hemolab 2 respectively. Each figure contains two charts, the upper chart gives the results for all systems and the bottom chart for MonetDB and SciDB.

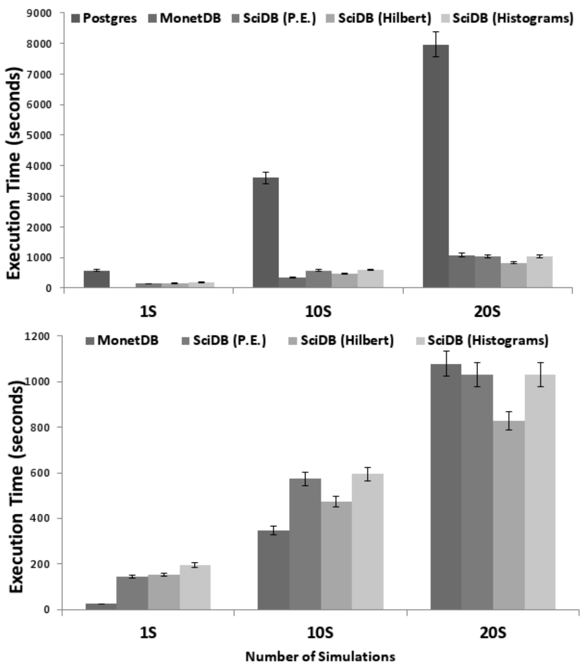


Figure 14: Hemolab 1 Query Results

The hemolab 1 results for analyses with 1 and 10 simulations are favorable to MonetDB. However, the Hilbert Space-filling curve yields a better performance when the query is scaled up to 20 simulations.

MonetDB has better performance for Hemolab 2 in all cases. Hemolab 2 in SciDB requires a join to be performed between attributes of the array instead of dimension indexes, which is inefficient. MonetDB column-store model, along with its dynamic index creation, is able to completely outperform SciDB in this scenario.

The Hilbert Space-filling curve method offers the best option among the multidimensional array representations. It is 60% faster for Hemolab 2 than P.E method and the Histograms Base Partitioning method. The dense and balanced data distribution allied with the lower dimensionality enables analyses to be executed more efficiently.

6.5 Comment on Execution Profile

MonetDB execution model heavily relies on the materialization of intermediate query results. When these intermediate results do not fit into main memory, they must be swapped to disk, resulting in performance loss. SciDB, on the other hand, has an execution model based on pipelining, and intermediate arrays do not always need to be completed materialized during query execution. Therefore, SciDB performance was less dependent on the amount of memory present on the system.

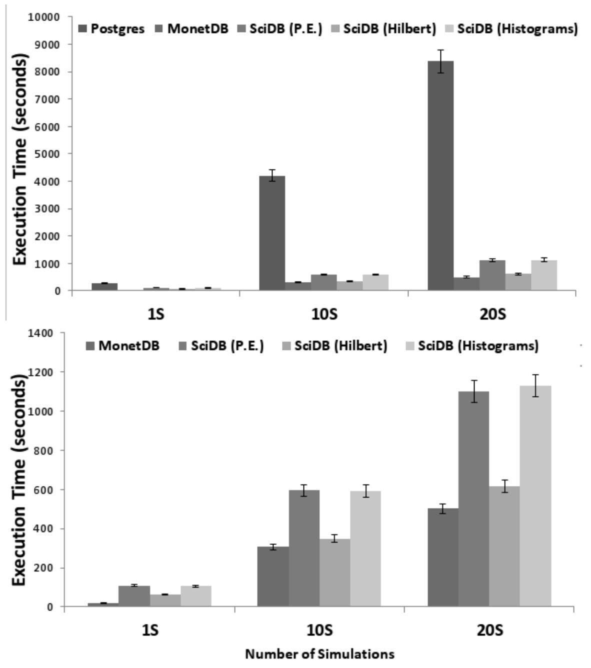


Figure 15: Hemolab 2 Query Results

The differences between SciDB and MonetDB execution models is further evidenced in Figure 16. The charts show the CPU usage and Disk I/O accomplished during execution of Hemolab 1 query. SciDB and MonetDB exhibit completely different patterns. MonetDB's CPU usage varies much more, and is interleaved with high disk I/O peaks. SciDB maintains a steady CPU usage throughout most of the execution time, and performs much less disk I/O than MonetDB.

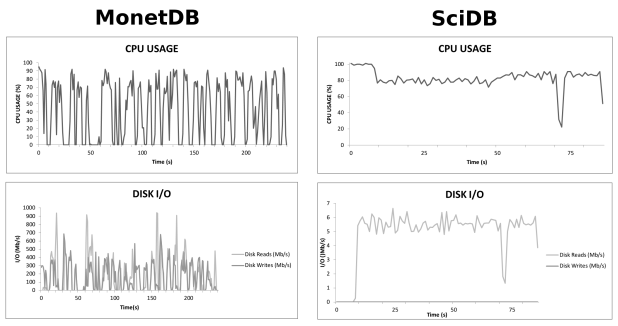


Figure 16: Hemolab 1 execution profile in MonetDB and SciDB

7. RELATED WORK

In this work, we are interested in managing scientific data generated by numerical simulations computed according to a space discretization modelled as an irregular mesh. There are few relevant works that discuss this topic.

A notorious example of work in mesh data management is Howe's [8] Gridfields data model, developed to represent computational meshes. The model is not bounded to any data structure, and separates mesh topology and mesh geometry. Another model designed to represent unstructured meshes is ImG-Complex [15]. This model represents meshes by using incidence multi-graphs. Both works deal with the topological representation of meshes and its underlying data. In our use case, we are concerned about executing analyses depending only on the geometric aspects, and thus, we do not require a specific model for dealing with topology yet.

Histograms have long been used in database management, mainly in query optimization. A technique to generate histograms for multidimensional data taking into account its sparsity is proposed in [5]. The use of histograms in query optimization and load balancing for cloud environments is discussed in [16]. These works are only examples of how histograms have been widely used in data management. The main point to highlight is that instead of using histograms in query optimization, we use equi-depth histograms as the basis for the multidimensional array partitioning strategy over distributed nodes so that each chunk holds approximately the same number of points, obeying the ordering on each dimension.

There is an extensive literature [14][11] in multidimensional indexing structures for databases using space-filling curves. In [12], there is a discussion involving the use of space-filling curves in point cloud data management with MonetDB. The main difference with our work is that instead of having a dataset of simple point cloud data, we have a recurring point cloud structure associated with other dimensions, and we evaluate the use of multidimensional arrays and a space-filling curve to represent it.

8. CONCLUSION

In this paper, we proposed the use of array DBMSs to manage the data produced by simulations. We considered multidimensional arrays as it nicely models the dimensions and variables used in numerical simulations. We presented methods for the efficient mapping of variable values in simulations to evenly distributed cells in array chunks with the use of equi-depth histograms and space-filling curves.

We also considered the use of a row-store DBMS, PostgreSQL, and MonetDB a column-store DBMSs designed to provide better performance for analytical workloads. Unlike the multidimensional array data model, the relational model does not require any modification in the original data.

Our experiments show that using our techniques in an array DBMS produces a consistent allocation of simulation data within chunks with respect to a query workload. Each technique shows better performance results in comparison to MonetDB and PostgreSQL, at least on a particular class of the benchmark queries. Furthermore, our solution becomes less sensitive to restrictions on available memory than MonetDB. Nevertheless, both SciDB and MonetDB showed significant better performance than PostgreSQL, indicating that row-store DBMSs are not a good solution for analytical queries. Conversely, MonetDB offers a much simpler data representation with generally good performance.

We observe that the choice of DBMSs, as expected, impact on our results. For instance, column-stores adopting an in-memeory pipeline execution model may not incur in the problems observed with MonetDB for large datasets. Similarly, the mapping stucture needed by the Histogram-based method could have been implemented in a relational system with spatial indexing support, improving the query resolution performance.

Acknowledgment

This work has been partially funded by CNPq, CAPES, FAPERJ and Inria (MUSIC projects) and the European Commission (HPC4E H2020 project) and performed (for P. Valduriez) in the context of the Computational Biology Institute (www.ibc-montpellier.fr) and for (F. Porto, H. Lustosa and P. Blanco) in the context of the INCT-MACC project (http://macc.lncc.br).

9. REFERENCES

[1] P. J. Blanco, M. R. Pivello, S. A. Urquiza, and R. A. Feijóo. On the potentialities of 3d-1d coupled models in hemodynamics simulations. *Journal of Biomechanics*, 42(7):919–930, 2009.

[2] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, 2010.

[3] J. R. Cebral, M. A. Castro, C. M. Putman, D. Millan, and R. F. Frangi. A.: Efficient pipeline for image-based patient-specific analysis of cerebral aneurysm hemodynamics: technique and sensitivity. *IEEE transactions on medical imaging*, pages 457–467, 2005.

[4] R. Cijvat, S. Manegold, M. L. Kersten, G. W. Klau, A. Schönhuth, T. Marschall, and Y. Zhang. Genome sequence analysis with monetdb: a case study on ebola virus diversity. In *Datenbanksysteme für Business, Technologie und Web (BTW, 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*, pages 143–150, 2015.

[5] F. Furfaro, G. M. Mazzeo, D. Saccà, and C. Sirangelo. Compressed hierarchical binary histograms for summarizing multi-dimensional data. *International Journal on Knowledge and Information Systems*, 15(3):335–380, June 2008.

[6] C. H. Hamilton and A. Rau-Chaplin. Compact hilbert indices for multi-dimensional data. In *CISIS*, pages 139–146. IEEE Computer Society, 2007.

[7] HeMoLab. Hemodynamics modeling laboratory, December 2014. http://hemolab.lncc.br.

[8] B. Howe. *Gridfields: Model-driven Data Transformation in the Physical Sciences*. PhD thesis, Portland, OR, USA, 2007. AAI3255425.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, page 2012.

[10] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. Monetdb/sql meets skyserver: the challenges of a scientific database. Technical report.

[11] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Rec.*, 30(1):19–24, 2001.

[12] O. Martinez-Rubi, P. van Oosterom, R. Gonçalves, T. Tijssen, M. Ivanova, M. L. Kersten, and F. Alvanaki. Benchmarking and improving point cloud data management in monetdb. *SIGSPATIAL Special*, 6(2):11–18, 2015.

[13] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13:2001, 2001.

[14] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 263–272, 2000.

[15] A. Rezaei Mahdiraji, P. Baumann, and G. Berti. Img-complex: graph data model for topology of unstructured meshes. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1619–1624, 2013.

[16] Y. Shi, X. Meng, F. Wang, and Y. Gan. Hedc: A histogram estimator for data in the cloud. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 51–58, 2012.

[17] L. Sidirourgos and M. Kersten. Column imprints: A secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 893–904, 2013.

[18] M. Stonebraker, J. Becla, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for science data bases and scidb. In *Conference on Innovative Data Systems Research (CIDR)*, January 2009.

[19] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11. IEEE Computer Society, 2005.