Managing Sparse Spatio-Temporal Data in SAVIME: an Evaluation of the Ph-tree Index

Stiw Herrera[1], **Larissa Miguez da Silva**[1], **Paulo Ricardo Reis**[1], **Anderson Silva**[1,2], **Fabio Porto** [1,2]

[1] National Laboratory for Scientific Computing, Rio de Janeiro, Brazil, [2]DEXL Lab

{stiw, lamiguez, paulorbr, anderson, fporto}@lncc.br

Abstract. *Scientific data is mainly multidimensional in its nature, presenting interesting opportunities for optimizations when managed by array databases. However, in scenarios where data is sparse, an efficient implementation is still required. In this paper, we investigate the adoption of the Ph-tree as an in-memory indexing structure for sparse data. We compare the performance in data ingestion and in both range and punctual queries, using SAVIME as the multidimensional array DBMS. Our experiments, using a real weather dataset, highlights the challenges involving providing a fast data ingestion, as proposed by SAVIME, and at the same time efficiently answering multidimensional queries on sparse data.*

1. Introduction

A few Array data processing systems have appeared recently offering a data model and query algebra adequate for representing and processing non-Relational data [Baumann et al. 1998, Stonebraker et al. 2011, Lustosa et al. 2016, Zalipynis 2018]. In particular, SciDB was conceived having large scale scientific data management as target applications [Brown 2010], such as data produced by the Sloan Digital Sky Survey.

Array data considers cells indexed in a multi-dimensional coordinate system. For instance, in an 2D image, pixels can be implicitly indexed by their corresponding width and height values. In astronomy, a typical sky representation involves a 2 dimensional coordinate system based on right-Ascension (RA) and declination (DEC), similar to latitude and longitude on earth coordinate system. Thus, to model these type of data, array data processing systems consider an array data model, where cells are implicitly referred by the values of indexes in a multi-dimensional coordinate system. Such implicit indexing structure facilitates the direct access to the desired cells and simplifies the database schema (i.e. no need to plan for extra indexing structures).

The above described scenarios cover a wide spectrum of array based applications. However, there are a few applications whose coordinate system takes a particular and relevant variation. This is the case of data with incomplete coverage of the indexing domain. As an example, take a set of meteorology stations distributed on the ground in the city of Rio de Janeiro. There is a total of 31 of such weather stations spread through the state, so that they can capture and distribute the observations on temperature, rainfall and wind speed, to name a few. If we consider a geo-reference coordinate system, latitude-longitude, to index each of these weather stations, the resulting array would have many pairs of (lat-long) to which no value would be assigned. This is referred to as a sparse array in the literature.

Managing sparse arrays is a challenge for array data processing systems [Lustosa et al. 2016]. This is due to two main factors: (i) if a dense data structure is used, there will be many empty cells which will hamper the query access time while allocating huge non-used memory and disk space; (ii) conversely, when a sparse allocation is adopted, the coordinate system indexes are allocated as cell attributes, which jeopardizes the performance of index-based queries.

In this paper, we investigate this problem using the SAVIME array in-memory system as a test-bed for managing and answering queries on sparse data. We compare SAVIME's TOTAL strategy with the use of a Ph-tree multidimensional indexing in answering point and range queries. Our objective is to evaluate whether Ph-tree can be an efficient solution to store and query sparse data in array data processing systems. Ph-tree is not currently implemented in SAVIME but experiments show that it could be a candidate indexing structure supporting range and punctual queries in sparse data.

The remaining of this paper is structured as follows. Section 2 describes the Related Work. In Section 3, we describe the main methods involved in our research. Section 4 depicts our Experiments and finally, Section 5 concludes and proposes future work.

2. Related Work

Much of the data produced in many different fields from both science and engineering can be more naturally represented as multidimensional arrays. These arrays can be dense or sparse, and array DBMSs must feature data structures suitable to both situations.

The system SciDB [Stonebraker et al. 2011] is a popular implementation of a full-stack array database. It provides support for sparse arrays, however, by partitioning them on ranges of fixed size indexes on each dimension. Since SciDB splits the array into chunks, which are equally sized subarrays, it becomes necessary to define a balanced partitioning scheme when dealing with irregularly distributed data. For sparse arrays, this is seldom a trivial task, since a poor partitioning scheme may impact query response time, greatly affecting database performance [Lustosa et al. 2016]. Rasdaman [Baumann et al. 1998] was a pioneer array DBMS that estabilished arrays as first-class database structures. In Rasdaman, array structures are stored as objects on top of a relational DBMS. Despite being more flexible than SciDB, allowing the decomposition of sparse arrays into tiles of variable sizes, it still requires the user to manually partition the array into chunks of a approximately the same number of elements [Papadopoulos et al. 2016]. ChronosDB [Zalipynis 2018] multidimensional dataset representation also accounts for sparse data, however, it is still under development.

SAVIME [Lustosa et al. 2020] is an in-memory array DBMS developed for the management of multidimensional data that implements the TARS array data model. SAVIME was designed to provide a flexible storage format, while having the ability of processing the data the way it is generated, without carrying out costly conversions during ingestion. It also provides support for sparse arrays, heterogeneous memory layouts and functional partial dependencies with respect to dimensions, using special purpose memory mapping functions. It combines three kinds of dimension specifications for representing different layouts of data organization: Ordered, Total and Partial for respectively dense, sparse or partially sparse arrays.

Besides array DBMSs, there are other kinds of systems developed with the special purpose of performing array data access or management. TileDB [Papadopoulos et al. 2016] presents itself as an array storage management system to support scientific applications. It proposes dealing with dense and sparse arrays as organized collections of data updates called fragments in order to improve writing performance. Each fragment is associated to a directory on the file system that contains one file per array attribute. In the case of sparse arrays, an additional file with the exact coordinates indicate the non empty cells. HDF5 [The HDF Group 2021] is a popular library used for representing array data. However, it has no specific support for sparse arrays, requiring users to represent them as an additional array layer indicating the dense regions (i.e., arrays of arrays), which also has an impact on query performance.

3. Methodology

SAVIME implementation is based on the TARS data model. A Typed ARray (TAR) has a set of dimensions and attributes. A set of Typed ARrays compose a TAR Schema (TARS). The attributes tuple that compose a TAR cell are accessed via a set of indexes, that indicate the cell location within the TAR. To provide support for sparse arrays and non-integer dimensions, the TARS model define mapping functions, this way it is possible to have array data coming from different sources and storage layouts simply by having different mapping functions for different subarrays, the so called subTARs.

A subTAR is defined by the TAR region it represents; the position mapping function, that reflects the data layout; and the data mapping function, that translates the linear address into data values. In summary, SubTARs cover n-dimensional slices of a TAR.

The indexing of a multidimensional array system, such as SAVIME, considers a constant mapping between index keys and the corresponding cell values. In a most simplistic mapping, each dimensional value in a k-multidimensional indexing space can point directly to its corresponding slice of the array. If k values are passed as an index value to a k-dimensional array then a single cell can be retrieved.

However, in the case of sparse data, a large number of k-index values do not have an associated cell value. Thus, an efficient allocation would require a special mapping function for each k-index value. This is why tree indexing structures use pointers to map index key values to cell values.

The current solution in SAVIME considers the TOTAL representation, in which each cell value is qualified with its corresponding k-index values. Finding a particular key engenders scanning every cell within all SUBTARS that intersect with the range query. This has a drastic effect in the query performance. The TOTAL representation turns the data into a degenerated array, basically tabular data.

This paper explores an alternative for allocating and indexing sparse data in SAVIME, considering the Ph-tree multidimensional indexing structure. A Ph-tree is a type of tree suitable to represent hierarchies. According to [Zäschke et al. 2014] Ph-Tree is a multidimensional spatial index that has been designed to work with large datasets. The experiments reported in [Zäschke et al. 2014] demonstrate that the data structure is effective for memory *(space efficiency)*, which is achieved by combining or compartmentalizing binary prefixes. Another advantage is the fact that it uses *hypercubes* to navigate between nodes and sub-nodes of the tree, allowing data to be efficiently found *(Efficient* *data access).* The important properties of this structure are: it doesn't need rebalancing; it does not require moving data after insertion of a given key-value entry; it allows punctual queries and range queries, and according to the [Vancea 2015] it allows greater scalability using distributed systems architectures.

Concerning the storage and indexing of sparse multidimensional data, Ph-tree offers an interesting alternative. First of all, it is a native multidimensional data structure. Secondly, each dimension key is mapped to its binary representation, and all dimension keys share the same size. Thus, in a k-dimensional space, each cell is indexed by k binary strings of size d. As a matter of fact, there may be some savings of bits space with reuse of bits prefixes, see [Zäschke et al. 2014]. From a storage allocation viewpoint, as a tree data structure, Ph-tree separates cell allocation in memory from indexing. The mapping of an index leaf node to a cell node is provided through a pointer data structure.

We expect that Ph-tree will provide fast direct access to indexed cell values, even on sparse multi-dimensional indexed arrays. Our experiments evaluate this hypothesis.

4. Experiments

In our experiments, we use a sparse dataset based on data from the Consortium for Small-scale Modeling (COSMO). In total, the dataset has over 47 million array cells, 3.3GB of data and information such as date, latitude, longitude, precipitation, temperature among others. In our experiments, we used a subset of these attributes. The data was represented as in a single 3-dimensional structure (a TAR in SAVIME and a binary PATRICIA-tries combined with hyper-cubes in Ph-three) containing the X, Y, and Z dimensions, representing date, latitude and longitude respectively. We created and loaded the corresponding TAR in SAVIME using the following commands.

```
# CREATE_TAR("COSMO", "*", "explicit, data, data | explicit, latitude,
latitude | explicit, longitude, longitude", "precip,double");
# LOAD_SUBTAR("COSMO", "total, data, 0, 47497001, precip_dim_spec |
total, latitude, 0, 47497001, precip_dim_espec | total, longitude, 0,
47497001, precip_dim_espec ", "precip, precip" );
```

All experiments were performed on the same computer architecture provided by the research group of the National Laboratory of Scientific Computing (LNCC), the Data Extreme Lab (DEXL Lab). The computer's specifications are: 768 GB of memory and 2 x Intel Xeon CPU E5-2690 processors. Additionally, reported measurements in all experiments represent the average results of 5 runs.

Experiment 1: In the first experiment, we measured the time taken to insert the data both into SAVIME and in the Ph-Tree. An insertion time of 3899 ms with standard deviation of 133 ms was measured in SAVIME, while the Ph-tree took 68756 ms with standard deviation of 1158 ms, almost 18 times higher. That is expected, since the Ph-Tree has an intrinsic cost associated with keeping the structure updated through each value received, while SAVIME's data ingestion process is straightforward, without requiring a costly data conversion. It is important to highlight that fast data ingestion time was one of the guidelines for the development of the SAVIME system. Thus, it incurs in negligible bookkeeping processes during data ingestion.

Experiment 2: Although the insertion time is shorter for SAVIME, Figure 1 shows that the elapsed-time for running range queries by SAVIME is orders of magnitude higher than those for running the same queries using the Ph-tree indexing. In this experiment, we considered an increasing range query size returning 1.703.975, 7.779.863 and 12.286.703 cells, respectively. Observe that the fast data ingestion time, as observed in Experiment 1, is penalized once sparse data is considered as target for range queries. The three queries, represented by scenarios 1, 2 and 3, are described below.

```
# Scenario 1: scan(subset(COSMO, data, 2001010000, 2003200000));
# Scenario 2: scan(subset(COSMO, data, 2001010000, 2008210000));
# Scenario 3: scan(subset(COSMO, data, 2001010000, 2012120000));
```

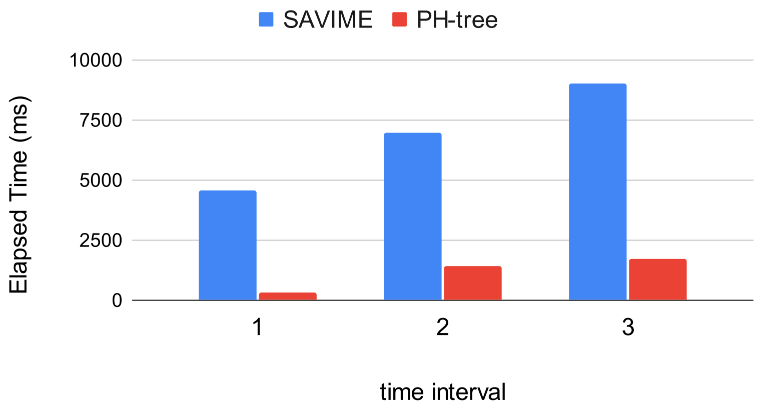

Figure 1. Range query in a date interval: SAVIME vs Ph-Tree

The standard deviation calculated for Scenarios 1 to 3 is, respectively, 89 ms, 220 ms and 244 ms for SAVIME, and 21 ms, 112 ms and 100 ms for the PH-tree.

Experiment 3: In this experiment, we want to evaluate the difference in elapsed-time between the two implementations when considering point queries. We designed an experiment with three randomly selected points within the data space. The three queries, represented by scenarios *a*, *b* and *c*, are shown below.

```
# Scenario a: subset(COSMO, data, 2001010000, 2001010000, latitude,
-32.905, -32.905, longitude, -60.782, -60.782);
# Scenario b: subset(COSMO, data, 2001010000, 2001010000, latitude,
-22.373, -22.373, longitude, -50.975, -50.975);
# Scenario c: subset(COSMO, data, 2012121500, 2012121500, latitude,
-14.700, -14.700, longitude, -52.350, -52.350);
```

Results in Table 1 are even more flagrant. As queries look for a single cell, Ph-tree can very efficiently use its index structure to directly retrieve the requested cell. In contrast, SAVIME does a SCAN in the SUBTAR containing the keys, looking sequentially for the requested cell.

5. Conclusion and future work

In this work, we discussed some of the implementation challenges regarding the representation of sparse arrays on database management systems, and presented the results of comparative experiments using SAVIME's non-indexed operations and using a Ph-Tree as an index data structure.

	SAVIME Time		Ph-tree Time	
Scenario	Mean	Standard Deviation	Mean	Standard Deviation
a	8126 ms	179 ms	0.1847048 ms	0.150914 ms
b	6461 ms	118 ms	0.0326342 ms	0.012131 ms
c	6248 ms	98 ms	0.0250956 ms	0.009485 ms

Table 1. Elapsed-time for point queries.

Our results indicate the potential of using indexes as powerful mechanisms for dealing with sparse arrays, despite the cost of keeping them updated through each data loading and updates. As future work, we plan to implement this index structure as part of SAVIME's query mechanism while improving Ph-tree insertion time, as well as studying the effects of using different data structures as indexes and to test on array benchmarks.

References

Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., and Widmann, N. (1998). The multidimensional database system rasdaman. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 575–577.

Brown, P. G. (2010). 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, page 963–968, New York, NY, USA. Association for Computing Machinery.

Lustosa, H., Porto, F., Blanco, P., and Valduriez, P. (2016). Database system support of simulation data. *Proceedings of the VLDB Endowment (PVLDB)*, 9(13):1329–1340.

Lustosa, H. L. S., Silva, A. C., da Silva, D. N. R., Porto, F. A. M., and Valduriez, P. (2020). Savime: An array dbms for simulation analysis and ml models prediction. *Journal of Information and Data Management*, 11(3).

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

Stonebraker, M., Brown, P., Poliakov, A., and Raman, S. (2011). The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer.

The HDF Group (1997-2021). Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/.

Vancea, B. A. (2015). Cluster-computing and parallelization for the multi-dimensional ph-index. Master's thesis, ETH Zurich.

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

Zäschke, T., Zimmerli, C., and Norrie, M. C. (2014). The ph-tree: a space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 397–408.