Hybrid.Poly: Performance Evaluation of Linear Algebra Analytical Extensions

Maksim Podkorytov, Michael Gubanov
Department of Computer Science
Florida State University

***Abstract*—Anecdotal evidence suggests that *Variety* is one of the most challenging problems in Big data research [1]. Different data providers use different data models and formats to represent their data, which causes significant impediment to data scientists, whose goal is to make sense of all relevant data regardless of the source. Hybrid.Poly [2], [3] is the analytical *polystore* data management system designed to make all data accessible to the analyst, oblivious of the source differences.**

In this paper, we focus on the in-depth analysis and performance evaluation of the Linear Algebra extensions added to the Hybrid.Poly language.

***Keywords*-Linear Algebra; Large-scale Data Management.**

I. Introduction

Proliferation of different *data models* and large-scale data management systems [4], [5], [6], [7], [8], [9], [1] complicate access to Big data coming from many sources in many shapes and flavors. Because of that it is sometimes called *Dark Data* [10], [11], [12], [2], [13], [14], [15], [16], [17], [3], [18], [19], [20], [21], [22] to reflect difficulties of getting insight into it. Hence, not only accessing, but also supporting any simple or complex analytical algorithms on top of *Dark Data* is also problematic.

Here we focus on performance evaluation of Linear Algebra operators of Hybrid.poly, an analytical polystore system. First, we briefly describe selected SQL language extensions, then switch to analysis of internals of in-memory algorithms, implementing these operators, finally describe experimental evaluation and related work.

II. Architecture

A. Query language

In Hybrid.Poly language [2], [3], it is possible to store and query *vectors* and *matrices* as first-class objects. Without these new *data types*, we would have had to resort to inefficient workarounds and convert these abstractions to relational equivalent and back, as demonstrated below.

First, we store *vectors* as attributes of a new type *vector* of a regular relational table. Second, we embed several linear algebra operations into our hybrid linear-relational query language to simplify implementation of both in-database relational and linear-algebra workloads. For instance, let us consider computing a *dot product* of 2 vectors. In our hybrid SQL, the query looks very natural and concise:

```
SELECT id, vector1.dot(vector2) FROM
    VectorStorage
```

In standard SQL, it is also possible, but using a much more inefficient and more complex query with a GROUP BY and JOIN. Before that even is possible, the vectors should be converted to their relational equivalent (i.e. value1, value2 that are their euclidean coordinates in the query below).

```
SELECT id, SUM(value1 * value2) FROM
    RelVecStore
WHERE index1 = index2 GROUP BY id
```

B. Matrix multiplication internals

The *matrix multiply* operator multiplies 2 matrices. Let A be a x-by-y matrix, B be an y-by-z matrix, then their product $A \cdot B = C$ is a x-by-z matrix with elements defined as follows: $C(i,j) = sum(A(i,k) \cdot B(k,j))$, $1 \leq k \leq y$. Numerically, this operator is implemented using a so-called IKJ scheme that is often used to multiply dense matrices. The implementation is a 3-level *for*-loop:

```
1 initialize C with zeros
2 for i: a matrix A row number (and the same
     row number in the result matrix C)
3   for k: a matrix B row number (and a
       column number in matrix A)
4     for j: a matrix B column number (and
         the same column number in matrix C)
5       C(i,j) += A(i,k) * B(k,j)
6 return C
```

The theoretical time complexity of calculating this product is $O(xyz)$; if $x = y = z$, that is, the input matrices are square ones, the theoretical time complexity is $O(x^3)$. The experimental complexity is lower due to caching, refer to the evaluation section for explanation.

Each matrix is stored in a flat array of doubles in a row-major order, i.e. the elements of the first matrix row are stored in the beginning of the array; then the elements of the second row are stored, and so on. Thus, the element $A(i,j)$ of x-by-y matrix A is stored in the position $i*y+j$. In order to efficiently use the modern processor architecture, in particular, multiple *cache* levels, it is necessary to read and write the data exhibiting both *spatial* and *temporal* locality, that is, the adjacent elements of the data array must be read or written within a relatively small time duration. Hence the structure of the aforementioned IKJ scheme. In the innermost for-loop, $A(i,k)$ is effectively constant, and the k-th row of matrix B is multiplied by the constant coefficient and added to i-th row of the result matrix C.

Moreover, the i-th row of matrix C stays in cache for a long time due to i being the index of the outer for-loop.

The aforementioned storage scheme may be improved further by splitting matrices into logical tiles. A tiled matrix is a *matrix of matrices*, that is, a two-dimensional array where each element of the array is a matrix. Any arithmetic operation on matrices, such as addition, subtraction, multiplication and Moore-Penrose pseudoinverse, needs to be implemented in terms of operations on the constituent blocks. As a byproduct of such matrix partitioning, it is possible to partition a large matrix across several computational nodes.

C. Matrix-vector multiplication internals

The *multVectorL* operator multiplies a matrix by a vector. Let M be an x-by-y matrix, v be a y-dimensional vector, then their product $w = M \cdot v$ is an x-dimensional vector with elements defined as follows: $w(i) = sum(M(i,j) \cdot v(j))$, $1 \leq j \leq y$. In pseudocode, this operator is implemented as follows:

```
1 initialize w with zeros
2 for i: a matrix M row number
3   for j: a matrix M column number
4     w(i) += M(i,j) * v(j)
5 return w
```

This implementation is cache-efficient, when the matrix M is stored in row-major order, and the same analysis, as we did in matrix multiplication, can also be applied here. The theoretical time complexity is $O(x \times y)$, where x is the number of rows in the matrix M and y is the number of columns in the matrix M. If the matrix M is square, that is, the number of rows is equal to the number of columns, the complexity is $O(x^2)$, the number of *elements* in the matrix. This runtime can be improved, if the matrix has special structure, so we do not need to use a double-nested for-loop to compute the matrix-vector product; for example, if the matrix or the vector have many elements that are equal to 0 (so they are *sparse*), only non-zero elements contribute to the result, and by using special data structures for storing sparse matrices and vectors, the runtime may be significantly improved.

D. Dot product

The *dot* operator calculates a dot product of 2 vectors. Let v and w be x-dimensional vectors, then their dot product is a scalar $c = sum(v(i) \cdot w(i))$, $1 \leq i \leq x$. In pseudocode, this operator is implemented as follows:

```
1 c = 0
2 for i: a vector v element number
3   c += v(i) * w(i)
4 return c
```

The time complexity is $O(n)$ where n is the vector length. This time complexity can also be made sublinear when the input vectors are sparse.

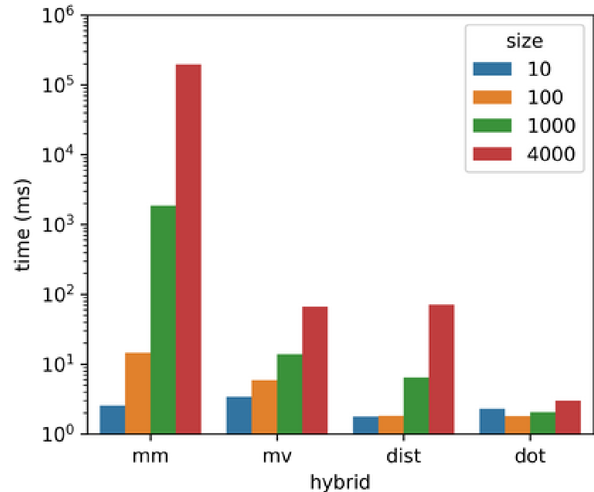

Figure 1. Hybrid.Poly Linear Algebra Operators Performance Evaluation. Vector space dimensionality ranges from 10 to 4000.

III. Evaluation

In this section, we run a series of Linear Algebra operators on Hybrid.Poly. The hardware used for the tests is a server having Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz with 80 physical cores and 192 GB RAM.

The workloads names in the Figure III are *mm* for square matrix multiplication by itself, *dist* for distance calculation between 2 vectors using a square *distance* matrix of the same size, *dot* for dot product between 2 vectors, The input sizes are dimensions of the matrices and vectors; e.g. vector of size 100 contains 100 floating-point elements and matrix of size 100 contains $100 \times 100 = 10000$ elements.

When input size increases one order of magnitude from 10 to 100, the matrix multiplication running time increases about 1 order of magnitude, and when input size increases one order of magnitude from 100 to 1000, the matrix multiplication running time increases about 2 orders of magnitude. This is different from what we predicted by theoretical computational complexity estimation, and we attribute that to better cache access patterns on small input sizes, for example, small matrices with $10^2 = 100$ elements fit entirely into L1 cache, while larger matrices with $100^2 = 10000$ elements do not fit entirely into L1 cache but fit into L2 cache.

IV. Related Work and Conclusion

Gubanov in [2] describes the design and the architecture of a multi-model polystore data fusion system Hybrid.Poly. In [3] we continue and describe a few use cases for the polystore. In this paper, we focus on evaluation of several analytical operators.

There are several recent attempts to add analytical extensions (e.g. linear algebra operators) to Map/Reduce-

based engines supporting SQL as an add-on (e.g. SimSQL [23]). Our system is a native parallel relational in-memory store, which puts it in a different category compared to any Map/Reduce-based derivative systems. In contrast to SQL or linear algebra extensions implemented on top of Map/Reduce, our engine, being a native parallel *relational* store supports *interactive* workloads by nature.

Another venue of related work is associated with declarative machine learning, namely, providing the user of a computing engine with a means to specify machine learning workloads in terms of high-level operators and freeing them from the need to specify the implementation details. The authors of SystemML implemented this approach for several back-ends – classical Hadoop, Map/Reduce on Spark [24], and explored the possibilities for the optimization [25]. Our approach differs by aiming at support of wider range of analytics than only machine learning and using an interactive parallel in-memory engine as a back-end, instead of Hadoop or Spark, that enables interactivity. Our hybrid query language is declarative, which is similar to these systems, as well as our SQL extensions also support linear algebra.

Acknowledgments: We would like to thank anonymous reviewers for their feedback on earlier drafts of this paper. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1701081.

References

[1] M. Stonebraker, "Big data means at least three different things..." in *NIST Big Data Workshop*, 2012.

[2] M. Gubanov, "Polyfuse: A large-scale hybrid data fusion system," in *ICDE*, 2017.

[3] M. Podkorytov, D. Soderman, and M. Gubanov, "Hybrid.poly: An interactive large-scale in-memory analytical polystore," in *ICDMW*, 2017.

[4] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki, "QUASII: query-aware spatial incremental index," in *EDBT*, 2018.

[5] H. Doraiswamy, E. T. Zacharatou, F. Miranda, M. Lage, A. Ailamaki, C. T. Silva, and J. Freire, "Interactive visual exploration of spatio-temporal urban data sets using urbane," in *SIGMOD*, 2018.

[6] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *SIGMOD*, 2018.

[7] X. Wang, A. Feng, B. Golshan, A. Y. Halevy, G. A. Mihaila, H. Oiwa, and W. Tan, "Scalable semantic querying of text," *PVLDB*, vol. 11, no. 9, pp. 961–974, 2018.

[8] I. Cano, M. Weimer, D. Mahajan, C. Curino, G. M. Fumarola, and A. Krishnamurthy, "Towards geo-distributed machine learning," *IEEE Data Eng. Bull.*, vol. 40, no. 4, pp. 41–59, 2017.

[9] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, "Computation reuse in analytics job service at microsoft," in *SIGMOD*, 2018.

[10] M. Priya, M. Podkorytov, and M. Gubanov, "ilight: A flashlight for large-scale dark structured data," in *MIT Annual DB Conference*, 2017.

[11] M. Gubanov, M. Priya, and M. Podkorytov, "Cognitivedb: An intelligent navigator for large-scale dark structured data," in *WWW*, 2017.

[12] M. N. Gubanov, P. A. Bernstein, and A. Moshchuk, "Model management engine for data integration with reverse-engineering support," in *ICDE*, 2008.

[13] M. Gubanov and L. Shapiro, "Using unified famous objects (ufo) to automate alzheimer's disease diagnostics," in *BIBM*, 2012.

[14] M. Gubanov, L. Shapiro, and A. Pyayt, "Readfast: Structural information retrieval from biomedical big text by natural language processing," in *Information Reuse and Integration in Academia and Industry*. Springer, 2013.

[15] M. Gubanov and A. Pyayt, "Medreadfast: Structural information retrieval engine for big clinical text," in *IRI*, 2012.

[16] S. Ortiz, C. Enbatan, M. Podkorytov, D. Soderman, and M. Gubanov, "Hybrid.json: High-velocity parallel in-memory polystore JSON ingest," in *IEEE BigData*, 2017.

[17] M. Simmons, D. Armstrong, D. Soderman, and M. N. Gubanov, "Hybrid.media: High velocity video ingestion in an in-memory scalable analytical polystore," in *IEEE BigData*, 2017.

[18] M. N. Gubanov, L. Popa, H. Ho, H. Pirahesh, J.-Y. Chang, and S.-C. Chen, "Ibm ufo repository: Object-oriented data integration," *VLDB*, 2009.

[19] S. Soderman, A. Kola, M. Podkorytov, M. Geyer, and M. Gubanov, "Hybrid.ai: A learning search engine for large-scale structured data," in *WWW*, 2018.

[20] M. Gubanov and M. Stonebraker, "Large-scale semantic profile extraction," in *EDBT*, 2014.

[21] ——, "Text and structured data fusion in data tamer at scale," in *ICDE*, 2014.

[22] M. Gubanov and A. Pyayt, "Type-aware web search," in *EDBT*, 2014.

[23] L. Shangyu, G. Zekai, G. Michael, P. Luis, and J. Christopher, "Scalable linear algebra on a relational database system," in *ICDE*, 2017.

[24] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda, "Systemml: Declarative machine learning on spark," in *VLDB*, 2016.

[25] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen, "Spoof: Sum-product optimization and operator fusion for large-scale machine learning," in *CIDR*, 2017.