David Kernert, Frank Köhler, Wolfgang Lehner

Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database

Erstveröffentlichung in / First published in:

In Memory Data Management and Analysis: First and Second International Workshops. Hongzhou, 01.09.2014. Springer, S. 44–55. ISBN 978-3-319-13960-9.

DOI: http://dx.doi.org/10.1007/978-3-319-13960-9_4

Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database

David Kernert[1,2](✉), Frank Köhler[2], and Wolfgang Lehner[1]

[1] Database Technology Group, Technische Universität Dresden, Dresden, Germany
wolfgang.lehner@tu-dresden.de

[2] SAP AG, Dietmar-Hopp-Alle 16, Walldorf, Germany
{david.kernert,frank.koehler}@sap.com

Abstract. Large numeric matrices and multidimensional data arrays appear in many science domains, as well as in applications of financial and business warehousing. Common applications include eigenvalue determination of large matrices, which decompose into a set of linear algebra operations. With the rise of in-memory databases it is now feasible to execute these complex analytical queries directly in a relational database system without the need of transfering data out of the system and being restricted by hard disc latencies for random accesses. In this paper, we present a way to integrate linear algebra operations and large matrices as first class citizens into an in-memory database following a two-layered architectural model. The architecture consists of a logical component receiving manipulation statements and linear algebra expressions, and of a physical layer, which autonomously administrates multiple matrix storage representations. A cost-based hybrid storage representation is presented and an experimental implementation is evaluated for matrix-vector multiplications.

1 Introduction

Within the recent decades, data scientists of all domains are increasingly faced with a growing data volume produced by historical events, experiments, and simulations. The era of data deluge and big data has shown the limitation of existing, often non-scalable and domain-specific persistence and computation solutions, which brought scalable database systems back into the discussion. Large numeric data, arranged in vectors and matrices, appear in many science domains, as well as in business warehouse environments. Examples can be found in theoretical nuclear science, genetics, engineering and economical correlation analysis. Analytical algorithms in those fields are often composed of linear algebra operations, including matrix-matrix, matrix-vector and elementwise multiplications. Moreover, linear algebra operations form the building blocks of machine learning algorithms [1] used in data warehousing environments, which is a common domain for commercial databases.

As conventional database management systems (DBMS) neither provide appropriate data objects nor an interface for linear algebra primitives, data scientists rely on custom, highly specialized and hand-written solutions instead.

However, rather than being responsible for reliable and hardware-dependent solutions, many scientists would prefer to get rid of implementational details. A DBMS with integrated scalable linear algebra implementations could serve as framework that provides basic primitives for their analytical queries, and avoids redundant data copying into any external algebra system. The drop in RAM prices over the last years laid the foundation for shifting databases from hard disc into memory, and analytical queries gained a considerable performance boost on large data sets [2]. By accessing data directly in memory, this development permits to bridge the gap between databases and complex analytical algorithms. Hence, database-integrated linear algebra primitives can now be provided without significant loss of performance, and use cases from the science and business world benefit from such an architecture in many ways:

- **Single source of truth.** The data is persisted and kept consistently in the database, so there is no redundant copying from other data sources to external libraries needed. Furthermore, the corresponding metadata of data sets can be updated synchronously and consistently with the numerical data.
- **Efficient implementation.** Algorithms for linear algebra operations have been researched thoroughly for decades, so there is no need to re-invent the wheel. But tuned linear algebra libraries can be exploited as kernels in the database engine to offer a computational performance that is competitive with existing numeric libraries.
- **Transparency.** A DBMS with our architecture handles different physical storage representations autonomously and provides internally well-partitioned matrices and vectors as self-contained data objects transparent to the user.
- **Manipulation of data.** In common analytic workflows, large matrices are no static objects. As they are manipulated in an iterative process, the data manipulation capabilities of a database will meet the analytical demands better than the tedious maintaining of multiple data files.

This work presents an architectural model for integrating large linear algebra objects and basic operations into a column-oriented in-memory database system. Section 2 provides an overview of recent research about the integration of array structures into databases and efficient linear algebra algorithms in general. The two-layered architectural model, a list of conceptual requirements for the logical data model and its physical mapping to the column store are presented in Sect. 3. Section 4 proposes a hybrid storage representation for large matrices and a strategy to cluster a large matrix into dense and sparse subparts. Our experimental setup and an evaluation of a sparse matrix vector multiplication are shown in Sect. 5. Finally, Sect. 6 summarizes our findings.

2 Related Work

2.1 Linear Algebra in Databases

The gap between the requirements of scientific computing and what is provided by relational databases is a well-known topic in the database community.

Ways to integrate multidimensional array data into the database context have recently been presented by the SciDB [3] team with ArrayQL[1], following the SQL extension SciQL [4]. The latter provides operators for spatial filters used for image processing but it lacks support for linear algebra objects as first class citizens.

A lot of research has been done in the context of data analytics and business intelligence, where linear algebra operations are the building blocks of data mining algorithms. Prior work [5] has shown how vanilla SQL can be used to calculate linear algebra expressions, although they add some user defined functions and infix operators to make the query look more natural. However, they admit that SQL terms rather pair up scalar values than treating vectors as "whole-objects" and does thus not fit the natural way of thinking of a data scientist with a mathematical background. They also state that expressions based on SQL require the knowledge of a certain storage representation, for instance the triple representation for matrices, which is not optimal for many use cases. From a performance perspective, Stonebraker et al. [6] propose the reuse of carefully optimized external C++ libraries as user defined functions for linear algebra calculations, but they leave the problem with resource management and suitable data structures in this "hybrid" world yet unsolved. Another approach based on Hadoop is SystemML [1], where basic linear algebra primitives are addressable via a subset of the R language with a MapReduce backend. Few commercial data warehouse vendors already offer minor support for linear algebra operations integrated in the database engine, but to the best of our knowledge there is no solution which integrates transparent optimization based on topological features of the matrix (e.g., sparsity).

2.2 BLAS and Matrix Multiplications

As we want to provide a solution that is able to compete with hand-tuned implementations, we have to glimpse outside the database world, where efficient linear algebra computation has been thoroughly researched for several decades. It is commonly agreed that a tuned BLAS[2] implementation is the best choice for computing small, dense matrices. Its interface is implemented by specially tuned libraries utilizing single-instruction multiple-data (SIMD) instructions. Libraries are provided by the open-source world or directly by hardware vendors, like ATLAS[3] or Intel MKL[4]. Although the current theoretical lower complexity bound for dense matrix multiplication is $O(n^{2.3727})$, initially presented by Coppersmith and Winograd [7,8], BLAS implementations still rely on the naive $O(n^3)$ algorithm, since the constant of the Coppersmith-Winograd algorithm is simply too high for being practicable. Nevertheless, for very large matrices a recent paper [9] shows that Strassen's Algorithm with the complexity of

[1] Array Query Language, http://www.xldb.org/arrayql/.
[2] Basic Linear Algebra Subprograms, http://www.netlib.org/blas/.
[3] Automatically Tuned Linear Algebra Software, http://math-atlas.sourceforge.net/.
[4] Intel Math Kernel Library 11.0, http://software.intel.com/en-us/intel-mkl.

$O(n^{2.8074})$ combined with a NUMA-aware hierarchical storage format outperforms the ATLAS library.

Research on sparse matrices has been less established as for dense, so there were some efforts within the last years to reduce the complexity for fast sparse matrix multiplication from a theoretical perspective [10,11]. Their general idea is to separate the matrix column/row-wise into a dense and a sparse part where the split point is determined by minimizing the number of total algebraic operations, while they admit that their work is only of theoretical value because they rely on Coppersmith-Winograd complexity for rectangular matrix multiplication. This at least confirms our conceptual model to cluster the matrix parts according to their density and treat sparse and dense parts differently. From an algorithmical perspective, there has been recent work on parallel sparse matrix-matrix multiplication [12] and cache-oblivious sparse matrix-vector multiplications [13] using a hypergraph partitioning method.

2.3 Storage Representation of Sparse Matrices

It is widely known that there are various ways to store a sparse matrix, and each of them might be best for a certain situation. The efficiency of a certain storage representation strongly depends on the specific topology of the matrix, since there are typically recurring shapes, such as diagonal, block diagonal or blocked matrices. A comprehensive overview over the different types of sparse storage representation is given in the work of Saad et al. [14]. Storing matrices in hybrid sparse-dense representations, in the way we will present in the remainder of this paper, has – to the best of our knowledge – not been presented in literature so far.

3 Architecture and Requirements

Our architectural model of the linear algebra database engine, sketched in Fig. 1, can be logically separated into two main components: First, the logical layer contains the data model and provides methods to parse linear algebra expressions and choose an appropriate algorithm for the operations to execute. Second, in the physical layer, the storage agent maps the logical linear algebra objects (i.e. matrices and vectors) into the column-oriented storage model by utilizing different internal representations depending on sparsity and shape.
The requirements for the *logical* component include:

- **Linear Algebra Query Language.** The common query language of relational databases is SQL, which was originally designed and established for expressions of the relational algebra. As a matter of fact, SQL does not comprise operations or data types of the linear algebra. In order to provide a natural interface for a database user with mathematical background it is crucial to provide matrices, vectors and multidimensional arrays in general as first class citizens. Moreover, for being able to optimize on the logical level, it is also necessary to pass a complete expression string containing basic linear algebra operators to the DBMS.

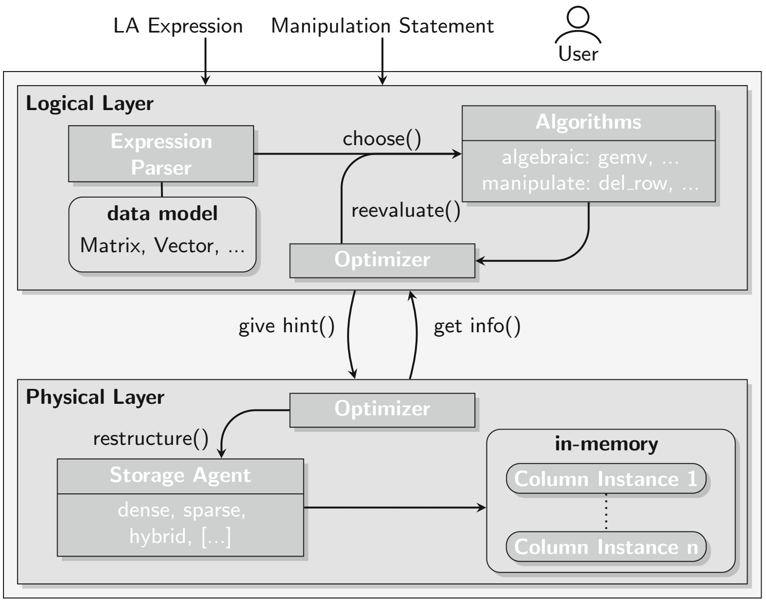

Fig. 1. Linear algebra engine architecture

- **Manipulation Language.** In contrast to a broad perception, matrices in analytical workflows are often *dynamic* objects that underly steady manipulations (e.g., in [15] several base states, which correspond to rows in the Hamiltonean matrix, are truncated before the eigenvalue calculation is repeated). Typical operations on matrices involve insertions, removals, and updates of single elements or whole rows or columns. Such in-place modifications are common in typical database applications, but infeasible with existing linear algebra libraries. The language should therefore offer a way to manipulate linear algebra objects element-, row-, column- and blockwise.
- **Linear Algebra Expression Optimization.** A linear algebra expression consists of operations on an arbitrary number of matrices or vectors. Optimizing the execution order on this layer can help to reduce the number of floating point operations significantly. As an example, consider a multiplication of three matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times l}$, $A \in \mathbb{R}^{l \times n}$

$$\text{expression} = \text{“}A \cdot B \cdot C\text{”} \tag{1}$$

Following associations law expression (1) can be evaluated in two ways, either multiply $(A \cdot B)$ first and then C from the right side or multiply $(B \cdot C)$ first and A from the left side. Assuming dense algebra, it turns out that with $k \gg \{l, m, n\}$, the second execution order requires $\frac{2}{1+\epsilon}$ times the number of floating point operations than the first order.

It is noteworthy that this holds only for dense operations, i.e., every matrix element is taken into account, regardless whether it is zero or not. Since multiplications with zero are as expensive as non-zero multiplications, the optimizer should be aware of the sparsity, which might change the optimal execution order. The number of operations N_{op} then can be obtained by considering matrix elements as triple *relations* $\{\text{row}, \text{col}, (A)_{ij}\}$. It is proportional to the join product of two matrix relations A and B with the condition $A.\text{col} = B.\text{row}$. The multiplication then rather turns into a relational join followed by a projection [10] where techniques of join size estimation (e.g., based on hashing [16]) can be applied to estimate the cost of the sparse algorithm.

Because of the importance of sparsity for optimizing the expression execution, it is desirable that the logical layer receives information about the physical data structure. This should be managed by a globally acting optimizer, which forms the interface between both layers. It combines the physical structure information with statistical information about prevalent algorithmic patterns performed on certain objects, as an efficient execution strongly depends on the conformance between the algorithm and the data representation. This information can in return be passed as a hint to the physical component, which should be able to reorganize the storage representation. The requirements of the *physical* layer are:

- **Multiple In-Memory Storage Representations.** In order to minimize the storage consumption of a large matrix, dense and sparse subparts are stored in separate representations. Each of the storage classes internally uses the native column-oriented storage of the database. As matrices are two-dimensional objects, they cannot be stored naturally in the sequentially addressable columns. Thus, matrices have to be linearized, which is effectively a mapping of matrix elements from the two-dimensional into the one-dimensional space. This is accomplished by ordering the elements according to a certain order (i.e., a space filling curve). 2D-arrays in common programming languages are arranged according to *row-major* (C++, Python) or *column-major* (Fortran, MATLAB) order. Examples for isotropic curves are the *z-curve* (or Morten-order) [17] and the *Hilbert-curve*. The adequacy of the order may depend on specific algorithmic patterns on the object, and as the columns are completely held in memory, the jumps caused by an inappropriate order will at most result in cache misses. However, most numeric libraries require a certain order, and to use them as kernels, our architecture provides a flexible transformation mechanism.
- **Leveraging Parallelization and SIMD Instructions.** In the context of distributed memory there has been recent work about parallel (sparse) matrix-vector and matrix-matrix multiplications [12]. The fundamental trade-off is communication costs versus computation costs, depending on the level of parallelization.

  Low-level parallelization and multithreading is already provided by many numeric libraries, such as ATLAS or Intel MKL. Moreover, most linear algebra calculations degenerate to numerical operations on vectors, thus they fully benefit from SIMD instruction sets. Wherever possible, we want to make use of vendor-provided C++ BLAS kernels that have already been well tuned for the specific hardware characteristics.

- **Data Load.** The common storage format of large scientific data sets that are produced by simulations and experiments are files. Our model foresees an *initial* loading of data from files by using any CSV-parser that connects via a client driver to the database.

4 Topology-Aware Restructuring Using Clustering Strategies

Matrices that are initially loaded into the database are first staged in a temporary sparse structure, for instance in the triple representation. As a consequence, algorithms on staged matrices will in general perform miserably, especially if the matrix has a rather dense topology. The database user does generally not know the topology of the matrix, at least it should not be required to specify the structure in advance. In our model the linear algebra engine restructures the staged matrices by clustering subparts into dense and sparse regions. A reasonable approach is to cluster regions density-based [18], hence classify clusters where the density distribution exceeds a certain threshold as dense and the remaining parts as sparse. The resulting clusters should have rectangular shapes with a minimal extent that should be defined according to the hardware specifications, for example a block should just be large enough to fit into the CPU cache. Figure 2 shows a 800×800 sparse Hamiltonean matrix[5] as an example from nuclear physics research (see Sect. 5.3). For the illustration we used square blocks of dimensions 100×100 and a density-based clustering with the kernel:

$$\mathcal{K}(i, j, i_0, j_0) = \begin{cases} \frac{1}{C} & \text{for } (i,j) \in \text{Block}(i_0, j_0) \\ 0 & \text{else} \end{cases} \tag{2}$$

where i is the row coordinate of a matrix, j the column coordinate, C a normation factor and (i_0, j_0) are the coordinates of a matrix element inside a fixed Block(i_0,j_0). After applying $\mathcal{K}$ to the data of Fig. 2 we effectively get a 2D histogram with 2D block bars of different heights. Figure 3 shows the density distribution relative to two different block density thresholds ρ_c. It can be imagined as a rectangular mountain range in the ocean with a variable water surface level. The higher ρ_c is, the fewer are the remaining dense parts which "protrude" from the surface. The actual question is where to place the cut level ρ_c, which is in general a nontrivial, multidimensional optimization problem.

A $m \times n$ matrix can be clustered into N^C rectangular $m_d^{(j)} \times n_d^{(j)}$ dense regions and $m_{sp}^{(i)} \times n_{sp}^{(i)}$ sparse regions with density ρ_i. Assuming costs τ_{sp} and τ_d for a single element operation in the sparse and dense storage representation,

[5] For illustration purposes we regard a relatively small matrix. Depending on the scenario, the matrices can reach dimensions of up to $10^{10} \times 10^{10}$.

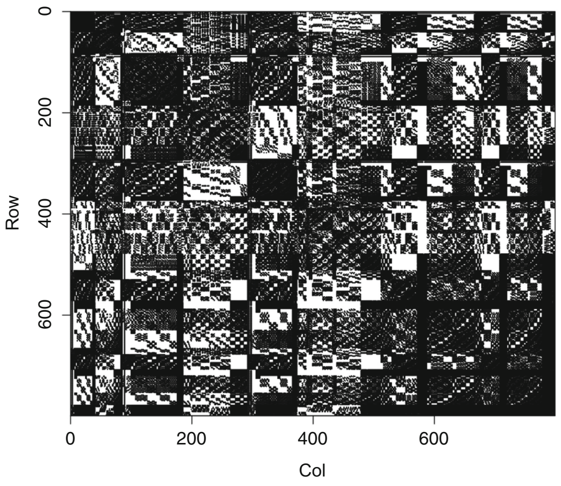

Fig. 2. A 800 × 800 Hamiltonean matrix resembling the quantum mechanical state of an atomic nucleus in the NCSM model. (Example from theoretical nuclear physics.)

the total cost T of a complete matrix operation[6] on a hybrid representation can be estimated as

$$T = \left(\sum_{j \in \{d\}} A_d \left(m_d^{(j)} n_d^{(j)} \right) \right) \tau_d + \left(\sum_{i \in \{sp\}} A_{sp} \left(N_{nnz}^{(i)} \right) \right) \tau_{sp} + \gamma A_C \left(N^C \right) \quad (3)$$

where $N_{nnz,i} = \rho_i m_{sp}^{(i)} n_{sp}^{(i)}$ is the absolute number of nonzero elements in the i^{th} sparse part. The A's denote the algorithmic complexity of the corresponding algorithm, for instance $A_d(N) = N^{3/2}$ for the naive matrix-matrix multiplication. The last term in (3) refers to the algorithmic overhead, which is connected with the number of subparts N^C. The clustering is ideal if T is minimal. Finding the absolute minimum is generally a nontrivial variational problem in a high-dimensional space. However, depending on the operation, T can degenerate into much simpler expressions, as for the general matrix-vector multiplication (GEMV). The algorithmic access pattern of the GEMV algorithm on a matrix is strictly row-major, thus a row-wise clustering keeps the conformance between algorithm and representation. This effectively means that the m rows of the matrix are clustered into m_{sp} sparse and m_d dense rows. With $A_{d,sp}^{\text{GEMV}}(N) = N$, $n_{d,sp} = n$ and $m_d = m - m_{sp}$ Eq. (3) can be transformed into

$$T^{\text{GEMV}} = nm\tau_d + n \sum_{i}^{m_{sp}} (\rho_i \tau_{sp} - m_{sp} \tau_d) \quad (4)$$

[6] T is proportional to the number of single element operations, according to the RAM model.

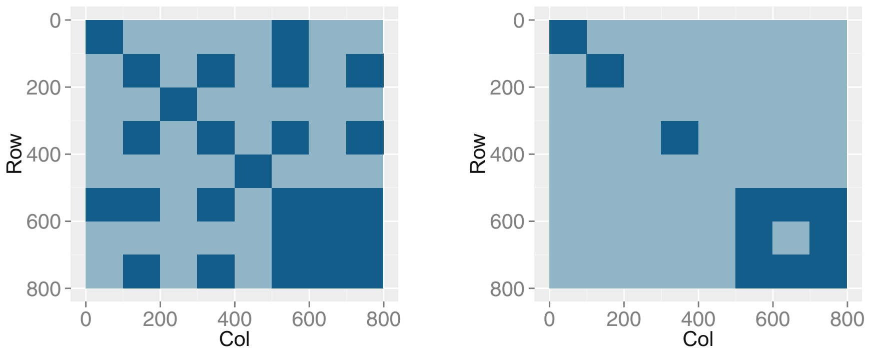

Fig. 3. Matrix density distributions relative to threshold $\rho_c = 0.5$ (left) and $\rho_c = 0.6$ (right). Dark blue denote regions with $\rho_i \geq \rho_c$, light blue means $\rho_i < \rho_c$ (Color figure online).

The right hand side of (4) is minimal if for the *row-density* in the equation $\rho_i < \tau_d / \tau_{sp}$ holds.

5 Experiments and Evaluation

5.1 Experimental Environment

In the context of the column-oriented SAP HANA database, we implemented parts of the physical layer in an in-memory column store prototype. Figure 4 shows the internal mapping of matrices of dense and sparse parts, where $K : \mathbb{N}^n \rightarrow \mathbb{N}$ can generally represent an arbitrary space-filling order (here shown with *row-major* order). Our sparse matrix-vector multiplication algorithm works with pure dense, pure sparse or hybrid representations.

5.2 Evaluation

The platform for our prototype implementation is an Intel Xeon X5650 system with 12 cores and 48 GB RAM. The performance for the GEMV operation was evaluated for sparse matrices in a pure dense, pure sparse and in a hybrid representation, containing subparts according to the density row-based clustering of (4). Without loss of generality, the relative row density was varied using generated matrices following a triangle random distribution to enable a row-based clustering into dense and sparse parts. Moreover we varied the overall density $0.24 < \rho < 1.00$ to illustrate the duality of dense and sparse representations. Figure 5 shows the graph of the measurements for the multiplication of a 12800×12800 matrix with a vector. As expected, the hybrid storage representation is always better than either pure sparse or pure dense. It converges against the performance for sparse matrices for small values of ρ and against the performance for dense matrices for high values of ρ.

Final edited form was published in "In Memory Data Management and Analysis: First and Second International Workshops. Hongzhou 2014", S. 44–55, ISBN 978-3-319-13960-9
10.1007/978-3-319-13960-9_4

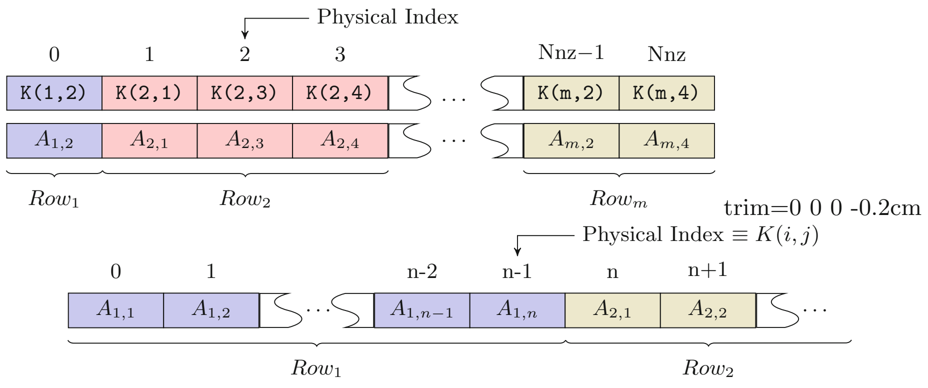

Fig. 4. The upper half shows the sparse representation in two columns: The first column contains the K-coordinate, the second the value of a nonzero element. Below, the dense representation: A single column contains every matrix element, including zero elements.

5.3 Lanczos Algorithm

The Lanczos algorithm is an iterative converging method, similar to the power method, to determine the eigenvalues of a real symmetric matrix. It used to find the energy states of an atomic nucleus, which correspond to the eigenvalues of the quantum mechanical Hamiltonean matrix [15,19]. Technically, the algorithm is composed of iterative sparse matrix-vector multiplications. According to the precision of the model, the Hamiltonean matrix can have arbitrarily many dimensions and can easily consume up to terabytes of storage. In our evaluation we used three matrices of different dimensions. Table 1 shows the speedup

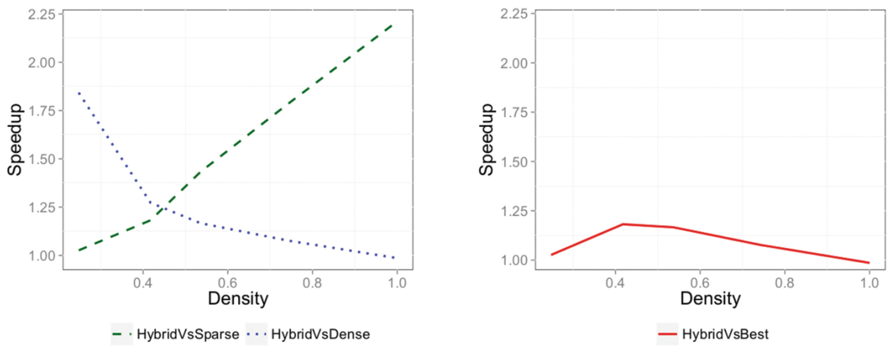

Fig. 5. Speedup in the multiplication of sparse matrices with vectors by using the hybrid representation. On the left side the speedup is shown relative to a pure sparse (dashed line) and pure dense (dotted line), and on the right it is compared to the respective best pure representation.

Table 1. Performance speedup of the multiplication of the sparse Hamiltonean $n \times n$ matrix with a random vector. The evaluation was performed on three matrices of different dimension (i.e. C1A, C2A, C2B) in either pure sparse, pure dense or in the hybrid representation. N^C is the number of subparts. The speedup is shown relative to the pure sparse and to the pure dense representation.

Matrix	n	Nnz	Density	N^C	HybridVsSparse	HybridVsDense
C1A	800	309816	0.484	27	52.0 %	0.1 %
C2A	3440	2930834	0.248	15	3.3 %	39.2 %
C1B	17040	42962108	0.148	60	0.7 %	149.7 %

of the hybrid representation compared to pure dense or pure sparse. There is again a positive speedup for each case, which however becomes less significant for the 17040×17040 matrix. This is substantiated with the complex topology of the matrixes as in Fig. 2, revealing that the rather simple row-based density clustering leaves room for optimization.

6 Summary and Conclusions

The problem of combining linear algebra operations with an efficient and scalable database environment is well-known in the database community as there are various use cases from science and business domains. We showed that it is feasible to integrate complex calculations in in-memory DBMS engines. Our architectural model aimes at applying database principles to linear algebra. It enables dynamic manipulation of matrix data and abstracts the problem of choosing an appropriate algorithm and storage representation from the user by letting the database optimize logical and physical execution. We identified sparsity as the main performance influencing characteristic of large linear algebra objects and proposed hybrid representations mapped to an in-memory column store. A cost-model based density clustering has been proposed to optimize sparse storage structure depending on matrix topology and algorithmic pattern. The evaluation showed that overall performance can benefit from an architecture that combines multiple internal storage representations.

Challenges to our architecture involve the exploitation of efficient BLAS kernels, distribution strategies, and the development of a natural query and manipulation language.

Acknowledgements. We want to thank the group of Prof. Roth from the Institute of Theoretical Nuclear Physics at TU Darmstadt (http://theorie.ikp.physik.tu-darmstadt.de/tnp/index.php) for providing us sparse matrix data and algorithmic insights into their domain-specific problem setting.

References

1. Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., et al.: SystemML: Declarative machine learning on MapReduce. In: ICDE, pp. 231–242. IEEE (2011)
2. Garcia-Molina, H., Salem, K.: Main memory database systems: an overview. IEEE Trans. Knowl. Data Eng. **4**(6), 509–516 (1992)
3. Brown, P.G.: Overview of SciDB: large scale array storage, processing and analysis. In: SIGMOD, pp. 963–968. ACM (2010)
4. Kersten, M., Zhang, Y., Ivanova, M., Nes, N.: SciQL, a query language for science applications. In: EDBT/ICDT Workshop on Array Databases, pp. 1–12. ACM (2011)
5. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., et al.: MAD skills: new analysis practices for big data. VLDB **2**(2), 1481–1492 (2009)
6. Stonebraker, M., Madden, S., Dubey, P.: Intel "Big Data" science and technology center vision and execution plan. SIGMOD Rec. **42**(1), 44–49 (2013)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. Symb. Comput. **9**(3), 251–280 (1990)
8. Vassilevska Williams, V.: Breaking the coppersmith-winograd barrier (2011)
9. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. CCPE **14**(10), 805–839 (2002)
10. Amossen, R.R., Pagh, R.: Faster join-projects and sparse matrix multiplications. In: ICDT, pp. 121–126. ACM (2009)
11. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms **1**(1), 2–13 (2005)
12. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments. SIAM J. Sci. Comput. **34**(4), 170–191 (2012)
13. Yzelman, A.N., Bisseling, R.H.: Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. SIAM J. Sci. Comput. **31**(4), 3128–3154 (2009)
14. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations, Version 2 (1994)
15. Roth, R.: Importance truncation for large-scale configuration interaction approaches. Phys. Rev. **C79**, 064324 (2009)
16. Amossen, R.R., Campagna, A., Pagh, R.: Better size estimation for sparse matrix products. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX 2010. LNCS, vol. 6302, pp. 406–419. Springer, Heidelberg (2010)
17. Morton, G.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company (1966)
18. Hinneburg, A., Keim, D.A.: A general approach to clustering in large databases with noise. Knowl. Inf. Syst. **5**(4), 387–415 (2003)
19. Vary, J.P., Maris, P., Ng, E., Yang, C., Sosonkina, M.: Ab initio nuclear structure the large sparse matrix eigenvalue problem. J. Phys. Conf. Ser. **180**(1), 012083 (2009)