David Kernert, Frank Köhler, Wolfgang Lehner

SLACID - sparse linear algebra in a column-oriented in-memory database system

Erstveröffentlichung in / First published in:

SSDBM '14: Conference on Scientific and Statistical Database Management, Aalborg 30.06. – 02.07.2014. ACM Digital Library, Art. Nr. 11. ISBN 978-1-4503-2722-0

DOI: https://doi.org/10.1145/2618243.2618254

SLACID - Sparse Linear Algebra in a Column-Oriented In-Memory Database System

David Kernert
Technische Universität Dresden
Database Technology Group
Dresden, Germany
david.kernert@sap.com

Frank Köhler
SAP AG
Dietmer-Hopp-Allee 16
Walldorf, Germany
frank.koehler@sap.com

Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT
Scientific computations and analytical business applications are often based on linear algebra operations on large, sparse matrices. With the hardware shift of the primary storage from disc into memory it is now feasible to execute linear algebra queries directly in the database engine. This paper presents and compares different approaches of storing sparse matrices in an in-memory column-oriented database system. We show that a system layout derived from the compressed sparse row representation integrates well with a columnar database design and that the resulting architecture is moreover amenable to a wide range of non-numerical use cases when dictionary encoding is used. Dynamic matrix manipulation operations, like online insertion or deletion of elements, are not covered by most linear algebra frameworks. Therefore, we present a hybrid architecture that consists of a read-optimized main and a write-optimized delta structure and evaluate the performance for dynamic sparse matrix workloads by applying workflows of nuclear science and network graphs.

Categories and Subject Descriptors
E.1 [**Data Structures**]: Arrays, Tables; G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems; H.2.8 [**Database Applications**]: Scientific databases

General Terms
in-memory databases, linear algebra

1. INTRODUCTION
Linear algebra in the context of database systems has recently been discussed in the research community, as it is a fundamental pillar of analytical algorithms. Matrices and matrix operations are used in a variety of use cases in the science and business world. Among these application fields are: nuclear physics [29], genome analysis [25], electrical, mechanical and chemical engineering [35], economical correlation analysis, machine learning and text mining [19], and graph algorithms [28], to mention only a few.

In the era of big data and data deluge in business and science environments, data replication from database management systems (DBMS) into external linear algebra systems, for instance Matlab or R, becomes more and more time- and memory-consuming. As a consequence, data should only reside in a single system, which for business environments usually is a relational DBMS. However, disk-based DBMS's exhibit poor performance of random access patterns on big data sets, such as linear algebra operations on very large matrices. The decrease in RAM prices over the last years laid the foundation for the shift of the database storage from hard disc into main memory, which resulted in a considerable performance boost of analytical queries on large data sets [18]. With the data residing in RAM, it has become worthwhile to investigate how structures and algorithms of numerical libraries can be integrated into the database engine. Besides the change in database system design due to the emerging hardware trends, the introduction of a column-oriented database design [9] has shown performance advantages on analytical workloads in contrast to conventional row-oriented approaches. As we show in this work, a columnar storage layout yields an easy adoption of known sparse matrix structures.

We outline two major limitations of using of a conventional DBMS for linear algebra applications: first, random access on hard disc and unsuitable data structures and operators result in a poor performance. The second restriction is usability. Since relational DBMS's do not provide appropriate data objects, such as matrices and vector, data scientists often rely on hand-written and highly specialized solutions. But rather than being responsible for maintaining hardware-dependent solutions, many scientists would prefer to work on a more conceptional level. A DBMS with integrated support for matrices as first class citizens could serve as a framework for scalable linear algebra queries, and supersedes the need for copying data to an external algebra system.

The integration of linear algebra operations into the database system imposes the following requirements:

- **Avoidance of data transfer**. With the data persisted and kept consistently in a single database system with integrated linear algebra functionality, the expensive copying into external systems becomes dispensable.

- **Single source of truth**. The absence of redundant copies of data in external systems avoids data inconsistencies. Moreover, the corresponding meta data of data sets can be updated synchronously and consistently with the raw data.

- **Efficient implementation**. Data scientists require a system that is able to compete with existing high performance systems, which usually are optimized for the platform hardware. Efficient algorithms for linear algebra have been researched thoroughly for decades, so there is no need to re-invent the wheel. Carefully tuned library algorithms can be reused as a kernel for medium-sized matrices[1] [23, 34].

- **Manipulation of data**. In several analytic workflows, large matrices are no static objects. Single elements, rows, columns, or matrix subregions should be able to be read, updated or deleted by the user.

- **Standardized user API**. Users from science environments desire to have an declarative and standardized language for matrix manipulation primitives and linear algebra operations.

To address each of these requirements, we present an architecture for sparse matrices that seamlessly integrates with a column-oriented in-memory DBMS, and provide an application interfaces that allows workflows from science and business environments to be run efficiently on our system. Our main contributions are:

- **Mutable sparse matrix architecture**. We present a matrix architecture with a columnar layout by taking advantage of well-known, ordered sparse matrix data structures. Moreover, we show how a two-layered main-delta storage can be exploited to provide dynamic matrix manipulation in constant time, without being penalized by a reordering of the optimized main matrix representation.

- **Matrix application interface**. Similar to the data manipulation language of transactional, relational systems, we sketch an application interface to access and manipulate matrices.

- **Applicability to non-numeric use cases**. We show how relational tables can reinterpreted as sparse matrices, and analytical queries can be rewritten to exploit efficient linear algebra algorithms.

- **Evaluation**. We implemented different matrix representations and evaluate the performance of our architecture against alternative approaches using real world applications of science and network graphs.

The remainder of this paper is structured as follows: Section 2 provides an overview of recent research about the integration of array and sparse matrix structures into databases. In section 3 we discuss different representations for large, sparse matrices. Section 4 presents our physical storage architecture. A brief introduction about different matrix access patterns, manipulation types and linear algebra primitves is provided in section 5. Two example workflows are described in 6, which are taken as a benchmark for our extensive evaluation in section 7.

2. RELATED WORK

This section is subdivided into four topics of related work where we see intersections with our work.

[1]We refer to medium-sized matrices as data volumes which fit into the memory of a single machine

2.1 Linear Algebra in Databases

Within the past years, a lot of research has been pursued to integrate multidimensional arrays and linear algebra operations into database systems. A popular work to mention within this area is MAD Skills [16], which shows how plain SQL can be utilized to calculate linear algebra expressions, although the authors add user defined functions (UDFs) and infix operators to make the query look more natural. Moreover they admit that SQL terms are rather suitable to pair up scalar values than treating vectors as “whole objects” and do not fit the natural way of thinking of a data scientist with a mathematical background. It is also stated 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. Their work is continued by introducing a UDF library for data mining algorithms on SQL DBMS’s, which however requires a DBMS supporting C++-written UDFs [21].

Some commercial data warehouse vendors claim to offer linear algebra operations that have been pushed down to the database engine [1, 2], e.g. in conjunction with support for the R statistical language. However, details about their systems are rare and they are – to the best of our knowledge – restricted to dense linear algebra and not amendable to non-numeric use cases. Another approach based on Hadoop is SystemML [19], where basic linear algebra primitives are addressable via a subset of the R language with a scalable MapReduce back-end.

2.2 Array DBMS

Approaching from a scientific point of view, there are array-based DBMS’s like RasDaMan [13] and SciDB [15, 33], which come with query languages that are designed for array processing, e.g. RASQL, AQL [3] or SciQL [24]. However, many array DBMS’s and the associated languages came up in domain-specific environments, for example multidimensional image processing, and are often geared towards the particular workflows of this domain, and not towards common linear algebra operations. SciDB [33] addresses this discrepancy by providing three different programming interfaces, each with a different flavor. In contrast, our work comprises an interface that could be integrated in many languages, hence, is rather independent from the particular language design. Another major difference of our work to array-based DBMS’s is that our system smoothly integrates with the relational world, since we use the data structures of a columnar RDBMS engine. This obviates the urge to transfer or convert the present data into arrays, which is often infeasible, e.g. for large, sparse graph data.

In order to combine linear algebra with in-memory, column-oriented DBMS’s, we recently proposed a two-layered architectural model [23]. The model foresees a logical component in the databases engine, which contains the language interface, linear algebra expression parsing and a logical optimization. The physical component utilizes the persistence infrastructure of the database system and maps sparse matrices to a columnar storage layout. This paper ties in with the physical part, where we will describe how two-dimensional matrices are efficiently integrated in a column-oriented database architecture and show how concepts from sparse matrix technology can be exploited on general dictionary-encoded value columns.

2.3 BLAS and Sparse Structures

As we strive for a solution that is able to compete with hand-tuned implementations, it is essential to take a glance outside the database world, where efficient linear algebra computation has been thoroughly researched for several decades. From a performance perspective, Stonebraker et al. [34] propose the reuse of carefully optimized external C++ libraries as user defined functions for linear algebra calculations, but leave the problem of resource management and suitable data structures in this *hybrid* world yet unsolved.

It is commonly agreed that a tuned BLAS implementation is the best choice for computations on 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 the Automatically Tuned Linear Algebra Software (ATLAS [4]), Intel Math Kernel Library (MKL [5]) or AMD Core Math Library (AMCL [6]). BLAS has been extended by the Linear Algebra Package (LAPACK [11]) and its distributed version ScaLAPACK [14], providing various solvers for linear equation systems and matrix factorization, which are however restricted to dense matrices.

Since sparse matrix algebra has been less established than dense in numerical computing applications, some library providers just started in the last decade to provide implementations for sparse matrices. However, sparse matrix operations on large, distributed systems is yet a topic of research [36, 10, 37]. It is widely known that there are various ways to store a sparse matrix, and each of them might be the best for a certain scenario. The efficiency of a storage representation highly 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 of the different types of sparse storage representations is given in Saad et al. [30]. Among the most used formats are the compressed sparse row (CSR) [12] and compressed sparse column (CSC) representations of matrices. A further description of the CSR representation is given in section 3.4, as it plays a key role in our indexed columnar database storage layout.

2.4 Generalization to Other Scenarios

Database-integrated sparse matrix representations and linear algebra operations may also be used for scenarios of other fields that contain matrix-like data structures. The duality between the representation of graphs as a set of vertices and edges and its adjacency matrix is well-known in the graph science community. Nevertheless, graph algorithms and efficient numerical linear algebra computing have been evolving rather separately. Recent research [22] has been pursued on the similarity between graphs and sparse matrices, and outlines that graph algorithms could benefit from the highly efficient, array-like access patterns of sparse linear algebra. For example, the Boost CSR graph [7] provides a compressed sparse row-based C++ graph implementation, which however has the disadvantage of being immutable. In contrast, our sparse matrix architecture described in 4 is not limited by immutability.

3. MATRIX STORAGE ARCHITECTURE

In this section we face the architectural question of how a large sparse matrix should be represented in a columnar database system. We therefore consider different columnar data structures for matrices with regard to their integrability into an in-memory DBMS.

The challenge of many analytical database systems, which strive for both quick query execution and immediate updates, is the dualism of read- and write-optimized structures. Therefore, we examine the representations according to the following two criteria: optimization for read access and the complexity of manipulations, i.e. the mutability of the data structure. Since these opposed characteristics are unlikely to be satisfiable achieved by a single structure, we use a main-delta approach. The separation of an abstract storage layer into two different physical representations, which is normally an optimized (compressed) static and an mutable delta structure, has already been applied in recent database systems [17].

We describe four different representations for matrices with respect to their applicability in our main-delta architecture, which is then presented in detail in section 4.

3.1 Matrix Table

A straightforward way of storing matrices in a RDBMS is to translate matrix rows to table rows and matrix columns to table columns. This approach results in a $m \times n$-sized table for a $m \times n$ matrix, as shown in Fig. 1a. In a column-oriented DBMS this would be reflected as n separate column storage containers. However, this representation often reaches its limitations if matrices are very wide, since the number of table columns in common DBMS's is usually restricted. The apparent advantage, that the matrix table representation is intuitive because it preserves the logical two-dimensionality of a matrix, loses its justification when the matrix size grows to an extent where displaying the matrix interactively is simply not feasible anymore.

Moreover, the matrix table is a dense representation, which makes it unusable for sparse matrices, unless the individual columns are compressed. Compressing the individual columns would decrease memory consumption, but usually adds the decompression to the algorithm execution runtime. The advantage of individual column compression in conventional business tables becomes superfluous as the columns of a matrix tend to be of similar structure.

3.2 Single Value Column

Another way of representing a matrix is to put every value (including zeros) adjacently into one large, consecutive value sequence. This translates into a single table column (see Fig. 1b) and internally results in a large value container.[2]

For this representation, a 2D to 1D linearization is needed. This mapping is implicitly performed on regular 2D-arrays in most programming languages, since the memory is sequentially addressable either way. In our example we use a row-by-row sequence, which is effectively a linearization according to the row-major order. The position of each matrix element in the sequence can be calculated using its 2D coordinates and the matrix $m \times n$ dimensions, i.e. the position of an element (i, j) in the sequence is $i \cdot n + j$. The advantage of this uncompressed representation is obviously that reads and writes are of constant complexity, whereas the disadvantage lies in the static memory consumption of $O(m \cdot n)$, independent of the sparsity of the matrix.

It should be mentioned that the single value column representation is clearly not relational, since positional referencing within a single table column elements is usually not supported and contradicts the relational thought of having an unordered set of relations. We will however assume that a logical layer for addressing single matrix elements in the DBMS exists and use the uncompressed 1D array representation as a comparison measure in our evaluation.

[2]In order to avoid misunderstandings with a matrix column, *container* refers to the storage structure for a column in a column-store DBMS.

Final edited form was published in "SSDBM '14: Conference on Scientific and Statistical Database Management. Aalborg 2014", Art. Nr. 11, ISBN 978-1-4503-2722-0
https://doi.org/10.1145/2618243.2618254

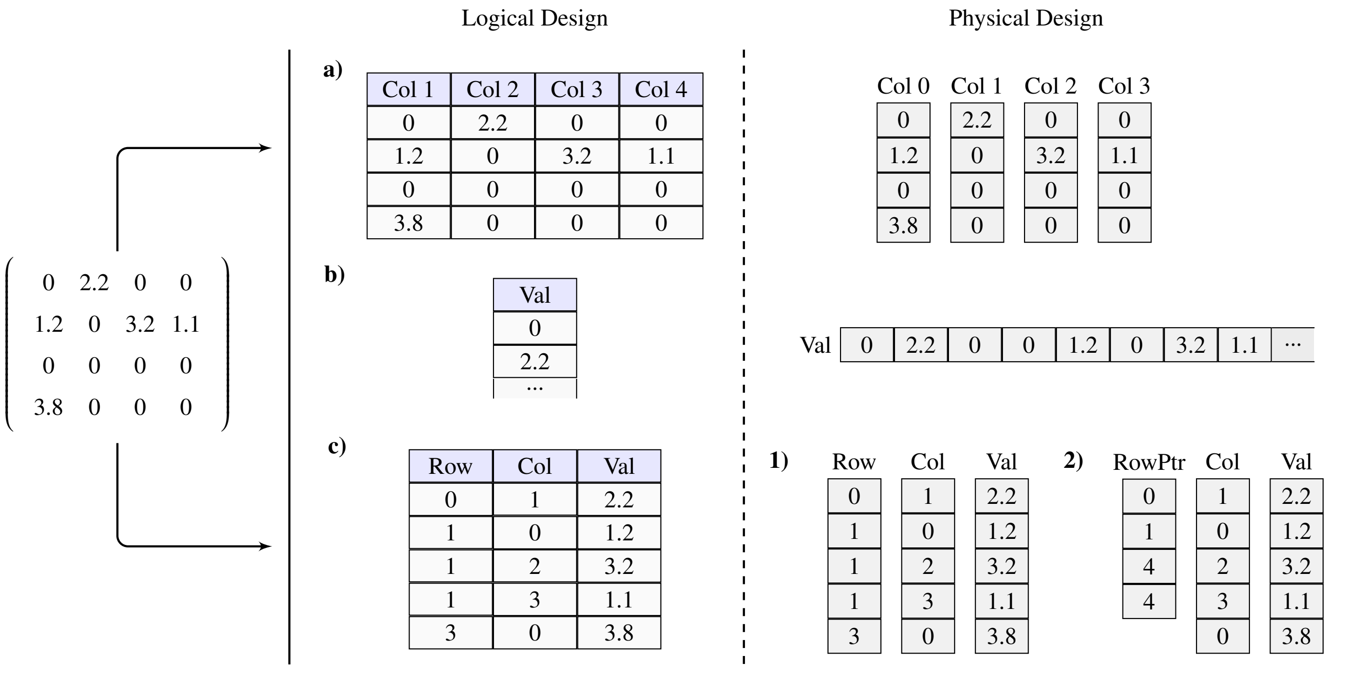

Figure 1: Overview of the different approaches to represent a matrix logically in a DBMS (middle) and their internal representations in a columnar storage (right): a) Matrix table, b) single value column, c1) triple table, c2) CSR representation

3.3 Triple Table

A third way of representing a matrix is as a collection of triples (Fig. 1c), where each triple contains the row and column coordinate, and the value of the corresponding matrix element: $\langle row, col, val\rangle$. The *row* and *col* attributes form a composite primary key, thus duplicate matrix elements are avoided. This variant turns out to be effective if the matrix is sparse, because only the non-zero elements have to be kept and the order of the rows is generally arbitrary.

In a column-oriented database, this triple table is represented as separate containers in the storage layer. Each of the containers has the length N_{nz}, which is equal to the number of non-zero matrix elements, resulting in a total memory consumption of $O(3N_{nz})$. To find an element in the unsorted, not indexed triple table a full column scan ($O(N_{nz})$) is required. The insertion of additional non-zero matrix elements is performed in constant time as they can just be appended to the end of the respective physical container, which makes the triple representation suitable as delta structure in our architecture. Further compression of the triple table can be achieved by sorting it according to one of the coordinates. The thereby resulting adjacent chunks of identical numbers in the corresponding container can then be compressed. This, however, influences the update and algorithmic behavior, so that the compressed format is considered as a separate representation.

3.4 CSR Representation

The compressed sparse row and compressed sparse column format [12, 30] are well-known sparse matrix structures in the numerical algebra community. For the sake of simplicity, we confine this description to the CSR representation (Fig. 2), which we briefly recapitulate. The CSC representation of a matrix A is equal to the CSR representation of the transposed matrix A^T, and vice versa.
The CSR representation is effectively a compression of the row-major-ordered triple representation. The row-major order allows

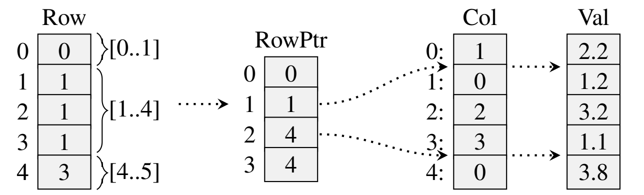

Figure 2: The CSR representation. Left: Compression of the row container. Right: The row pointer vector (RowPtr). The access path to the first matrix row is sketched.

replacing the row container by a row pointer vector (RowPtr) which contains only the start positions of each matrix row, as shown in Fig. 2. The central characteristic of the row pointer vector is that it also acts as an index, since a look-up for a row coordinate value provides the physical positions of the matrix row in the remaining containers of the triple table. As an example, to get all elements of the first row, every triple from the row start position $RowPtr[1]$ up to the end position $RowPtr[2] - 1$ is returned. The row pointer vector of an $m\times n$ matrix has thus the size $m+1$, where the $(m+1)^{\text{th}}$ element denotes the end position of the m^{th} row in the column and value containers. The total memory consumption is $O(2N_{nz}+(m+1))$, thus usually less than that of the triple format, because the inequation $(m + 1) \leq N_{nz}$ is only violated if the matrix contains rows of zeros.

In the original CSR implementation [12], the materialized row container is discarded and replaced completely by the row pointer vector. As contrasted to the uncompressed triple representation, the complexity for the inverse access, i.e. , finding the row coordinate x to a table position i is not constant. For this operation the interval of the start positions $I : [I_V, I_{V+1}]$ where $i \in I$ has to be determined. However, this can be easily performed using a binary search in an asymptotic complexity of $O(\ln(m + 1))$, as the row pointer vector is sorted. In contrast to many naive conventional compression techniques that are used columnar stores, a partial or complete decompression of the row pointer vector is not necessary. The row access in $O(1)$ and the (average) single element access in $O(\ln \sqrt{N_{nz}})$ time makes it a reasonable choice for our static main storage structure.

3.5 Impact of the Linearization Order

As the linearization order plays an important role in most of the abovementioned representations, it is essential to know that certain algorithmic patterns favor certain orders. The row-major and column-major ordering are biased linearization techniques. For instance, a row-major order would not be chosen as internal layout when it is likely that complete columns are accessed. A single column iteration translates into a *jump* memory access pattern on a row-major order, since the addresses are separated by the row width and spread over the complete memory section, whereas it yields in a cache-efficient, locally restricted and sequential memory scan on a column-major order. Although there are non-biased linearizations, such as the recursive Morton order [27] or the Hilbert curve, they exhibit poor cache locality for one-directional algorithmic patterns, e.g., BLAS level 2 operations [8].

Furthermore, the linearization order defines the compression into either CSR (row-major) or CSC (column-major). With the row pointer vector as index, algorithms with a row-centric pattern obviously benefit from a CSR structure whereas column-centric algorithms would favor a CSC-based approach. This introduces obviously a bias in algorithmic performance, but according to our notion the majority of algorithms usually are one-directional, i.e , they can be expressed in a way that accesses only one of the two dimensions. Examples are the matrix-vector multiplication or the graph breadth-first search, which we sketch in section 6. However, if an algorithm favors a CSC structure, but the matrix is available in CSR representation, then often an alternative algorithm working on the transposed structure can be used, since $A_{\text{CSR}} = (A^T)_{\text{CSC}}$.

Nevertheless, in contrast to the sole use of numerical libraries or common algebra systems, where the user is required to define the matrix representation in advance, and has to be aware of the algorithmic access and manipulation patterns, a DBMS that accommodates query statistics can act as advisor to reorder the matrix representation, as proposed in [23].

4. SYSTEM ARCHITECTURE

In this section, we present a novel approach for supporting mutable sparse matrices, which seamlessly integrates with a column-oriented DBMS. We show in particular how the mutability of large matrix data sets, which is one of our main requirements, is provided without losing the advantage of optimized storage structures. Moreover, we underline that the efficient indexing method of the CSR representation can be exploited by arbitrary tables, when integer dictionary-encoding is used.

We point out that all of the following data structures of our architecture are contained in RAM. In particular in sparse linear algebra, random access patterns are usual, which becomes clearer when we sketch example algorithms in section 6. For not being penalized by hard disk accesses, we chose a main memory database environment [18], which does not pose a limitation, since recent in-memory systems [17] are nowadays reaching storage scales of 1 TB and more.

In recent DBMS's [17] the storage architecture of each column is separated into a static main structure, which is compressed and read-optimized for online analytical processing (OLAP), and an incremental delta structure, which is write-optimized for online transactional processing (OLTP). The delta storage is merged into the main storage periodically, and each of these merge steps includes a reorganization, the original purpose of which is to improve the compression ratio by reordering the table rows. In our system, we exploit the reorganization to sort the columns by their values. This step is transparent to the user and can be implemented in a way so that online query execution performance is not affected. The internal algorithms are usually dependent on the representation and are therefore executed separately on the main and the delta storage.

4.1 Static Main

The static main component of our architecture contains a data representation that is optimized for read operations, and moreover to the patterns of sparse matrix algorithms. Our evaluation in section 7 shows that a CSR (CSC) representation turns out to be beneficial in a variety of use cases, especially in the applications we present in section 6, which are related to the sparse matrix-vector multiplication. Besides its efficiency and applicability in many numerical libraries, the CSR representation integrates well into a column-oriented DBMS, since the row pointer vector is at the same time both an index for the triple table and the compressed version of the row container itself.

We want to emphasize that the CSR representation, and thus the derived *CSR index*[3] are not limited to applications with matrices. Every database table accommodating a relation $R = \{a_1, a_2, ...\}$ of at least two attributes and arbitrary values, except the null value ($a_i \neq \texttt{NULL}$), can be transformed into an internal CSR structure, if the values are translated into subsequent integer values. We consider a column-store DBMS [17] that uses by default dictionary encoding for every column in order to reduce the memory consumption and improve the scan performance, since table values in business environments are predominantly reoccurring strings. That means, each table value of arbitrary format is assigned an integer $id \in \{0, 1, ..., n_{\text{values}} - 1\}$, so that only the integer value IDs are materialized and kept in the containers. This circumstance allows for creating a CSR-based representation for a large group of use cases.

The dictionary encoding is sketched in Fig. 3, which shows the internal database representation of a social network table. The dictionary of the *Name* attribute is sorted in ascending order by its values, which are then assigned consecutive integers, starting with zero. As a result, the materialized container for a table column consists only integer values. The single precondition for the applicability of our CSR-based representation is the ordering of the table. The sorting of every container in the corresponding table according to the value ids of the leading attribute, which is *Name* in Fig. 3 and *Row* for a matrix table, is performed during the reorganization step following a delta merge. After sorting the table, the CSR index is created.

It is noteworthy that in the original form of CSR [12] a two-level nested sorting is used to achieve a strictly row-major ordering of a two-dimensional matrix: the sort order is first by row, then by column values. However, the latter is not required to create the CSR index, although a subordering of the column leads to an increased algorithmic performance. This can be explained by cache effects: during a matrix vector multiplication, the column coordinates refer

[3]The CSR index corresponds to the row pointer vector

Final edited form was published in "SSDBM '14: Conference on Scientific and Statistical Database Management. Aalborg 2014", Art. Nr. 11, ISBN 978-1-4503-2722-0
https://doi.org/10.1145/2618243.2618254

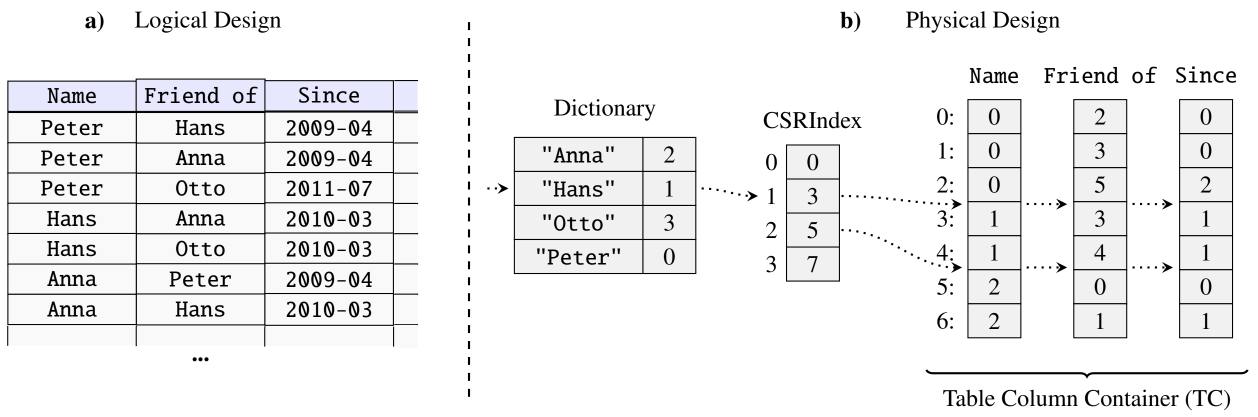

Figure 3: The dictionary-encoded columnar storage architecture of a social network graph table that contains *string* and *date* values. a) The logical view of the table in the database with two attributes (*Name, Friend of*) that denote the graph topology, and an auxiliary attribute (*Since*). b) The internal representation that consists of dictionaries and integer columns.

to positions in the target array. If they were randomly ordered, many cache-lines would have to be evicted and reloaded again, whereas an ascending order leads to cache-friendly writes.

4.2 Incremental Delta

The sorted characteristic of the optimized CSR representation makes it a static structure that is not mutable in constant time. Hence, in common numerical algebra workflows the representation has to be rebuilt after manipulating the matrix, even for slight changes like the single insert of an additional nonzero matrix element. This results in a $O(N \ln N)$ sorting overhead that becomes particularly expensive for very large matrices in a dynamic workload, like the example workflow described section 6.

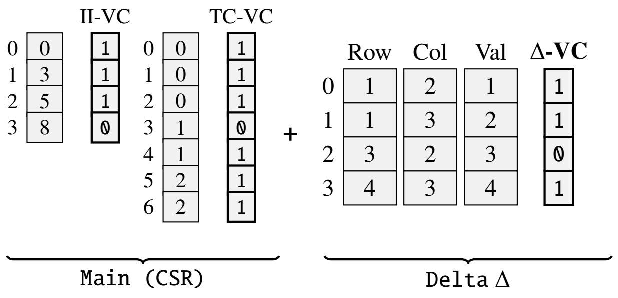

Figure 4: Column-oriented architecture containing a static main and an incremental delta structure.

Thus, our architecture (Fig. 4) foresees an updatable, incremental delta structure that coexists with the static main structure.

Inserts of non-zero elements are processed by simply appending a $\langle row, col, val \rangle$ triple to the unsorted delta structure. Updates of already existing nonzero elements are performed in-place, i.e., either in main or delta . For a single element update, this requires a binary search on the column container on the main and a scan on the delta structure, thus on average $O(\ln \sqrt{N_{nz}^M} + N_{nz}^\Delta)$ time.

Deletions of elements require an additional structure to keep track of the deleted elements. For this purpose our architecture contains *validity control* (VC) bitvectors for the main table, the CSR index and the delta structure. For every deleted element, the bit of the corresponding container position in the respective main (II-VC) or delta bitvecor (Δ-VC) is unset. Moreover, if a complete matrix row is removed, for instance row k, then the corresponding bit at position k of the TC-VC bitvector is unset.

5. APPLICATION PROGRAMMING INTERFACE

In classical database workloads, tables are commonly manipulated dynamically by inserting, updating or deleting data elements. However, the dynamic characteristic of relational database workflows also holds for large sparse matrix applications, as we describe for the nuclear science use case in section 6. Hence, it is a valid assumption that sparse matrices are not just queried in a single-pass, but rather modified in-between subsequent query executions as part of an analytical workflow, for instance that of section 6.1. Therefore, our system offers the user an interface with which sparse matrix data can be manipulated in a similar manner as relational tables with data manipulation language (DML) commands. This interface can then be utilized by the application programmer to construct user-specific UDFs for complex linear algebra workflows. In this section, we discuss basic manipulation primitives for matrix data from a logical perspective and describe how an matrix application programming interface (API) could look like.

We propose a database system that contains matrices as first-class citizens, for instance by extending SQL with a matrix data type, similar to the array type presented in [24]. Thus, matrices are defined in the data definition language (DDL) as such with the specification of its dimensions, which are stored as meta data in the system. The following API can then be exposed as built-in procedures that process matrix data types.

5.1 Access Patterns

As a basis for the following algorithms and examples, we briefly introduce the application programming interface for referencing matrix elements and regions. Each of the two matrix dimensions can be queried by providing either a point, range or no restriction. Based on this assumption, we define the subarray referencing matrix which is shown in Fig. 5.

To fetch single elements or matrix subregions we define the command

Final edited form was published in "SSDBM '14: Conference on Scientific and Statistical Database Management. Aalborg 2014", Art. Nr. 11, ISBN 978-1-4503-2722-0
https://doi.org/10.1145/2618243.2618254

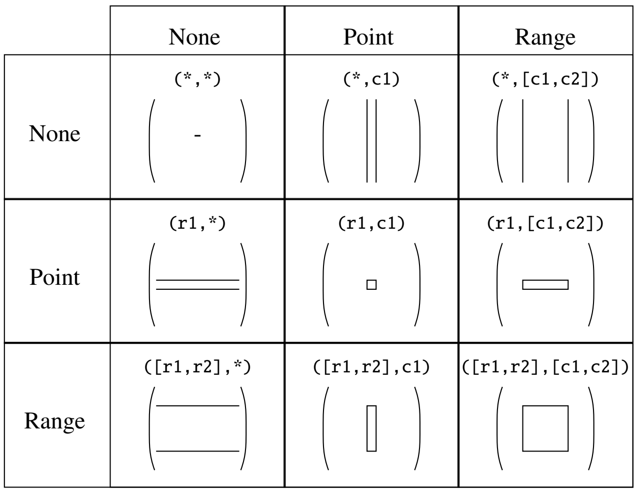

Figure 5: Matrix subarray access patterns

- **get**: is the counterpart of the relational *select ... where* statement, where the filter condition is replaced by a topological reference according to patterns shown in Fig. 5. For example, `get A(5,3)` returns a single matrix element, `get A(*,3)` references the third column and `get A(1,*)` the first row of matrix `A`. Two-dimensional submatrices are returned by defining their row and column range, such as `get A([2,5],[3,5])` to retrieve the rectangular region between the edge elements `A(2,3)` and `A(5,5)`. The complete matrix is referenced by providing no restriction in both dimensions, thus `A(*,*)`.

5.2 Data Manipulation Primitives

From the relational SQL perspective, the DML comprises commands to insert, delete, and update elements. The difference to a logical matrix context is that every single element of the matrix space $m \times n$ does in fact exist, independent of its value, including zero elements. Thus, there is no other interpretation of inserting a single matrix element than updating the already existing zero element of the matrix at the corresponding position. In the same way a deletion of a single element is rather described as setting the nonzero value to zero. However, if a complete row or column, or a submatrix is inserted with dimensions of either $m \times k$ or $k \times n$, then an insert can also be interpreted as an expansion of the matrix by k rows or columns, respectively. In the same way, a deletion of regions spanning the whole row- or column range can be seen as an effective shrinking of the matrix. To remove this ambiguity, we define the following commands:

- **set:** sets any single element or region in the matrix space $m \times n$ and overrides the previous value of the corresponding matrix region. As an example, `set A(9,3) value 5.0` sets a value at position `(9,3)`, whereas `set A([2,2],[3,3]) values (0.0, 0.0, 0.0, 0.0)` sets all the values of the square submatrix to zero.

- **delete:** only applies to either a $m \times k$ (k rows) or a $k \times n$ (k columns) subregion of the corresponding $m \times n$ matrix. It affects the matrix dimension in such a way that the adjacent parts are shifted to the first free place which was formerly populated by a deleted row/column. Thus, the resulting matrix has either the dimension of $(m-k)\times n$ or $m\times(n-k)$, respectively. For instance, `delete A(*,3)` executed on a 4×4 matrix `A` deletes the third column which changes the dimensions of `A` to 4×3.

- **insert:** is the logical counterpart of the delete operation. The insertion of either k rows or k columns results in matrix dimensions of either $(m + k) \times n$ or $m \times (n + k)$.

- **copy:** copies single elements, complete rows, columns or submatrices from any source position to a target position. If the target position exceeds the matrix bounds, then the copy operation only applies to $m \times k$ or $k \times n$ subregions. The overflowing rows or columns then affect the matrix dimension in the same way as an insert. The copy operation is derived by a consecutive `get` and `set` operations, if the target position stays within the matrix bounds, and by a get and set/insert operation if the target position exceeds the matrix bounds.

- **flip:** exchanges a $k \times l$ subregion from a source position to a target position, which must not exceed the matrix bounds. The flip can not be implemented solely by consecutive get and set commands, since either the target or source region has to be buffered temporarily.

Next to these basic commands, one can indeed define a variety of further operations, such as transpose. However, manipulations of a sparse matrix, such as insertion of elements, is not foreseen in common algebra systems such as Matlab [20], whereas they can be seamlessly integrated in our matrix architecture. Setting single elements in a matrix is a fundamental operation in a variety of applications, for example in LU-decomposition methods, such as Gaussian Elimination or the Doolittle algorithm [26]. Moreover, some analytical workflows tend to remove complete matrix rows or columns. In the example from nuclear science described in section 6.1, chunks of k rows and columns are deleted from the sparse matrix. More obvious examples are graph algorithms, where the elimination of a graph vertex corresponds to the removal of the adjacency matrix row (and the respective column).

5.3 Linear Algebra Primitives

In addition to access and manipulation commands, data scientists require basic linear algebra primitives to construct their application algorithms. Desired primitives include, but are not limited to multiplication, addition or inversion of matrices. We describe how a sparse matrix-vector multiplication operator `spGemv` is implemented on our columnar main-delta architecture. Sparse matrix-vector multiplication serves as building block for many applications, for example eigenvalue problems or cosine similarity calculations.

Let

$$y = A \cdot x$$

be a matrix-vector multiplication with $x \in \mathcal{R}^m$ and $A \in \mathcal{R}^{m\times n}$. For illustration purposes we consider the transposed equation

$$y^T = x^T \cdot A^T$$

which can be written as follows

$$y^T = (x_1 \;\; x_2 \;\; x_3 \;\; ...) \cdot \begin{pmatrix} A_{11} & A_{12} & A_{13} & \\ A_{12} & A_{22} & A_{23} & ... \\ A_{13} & A_{23} & A_{33} & \\ & ... & & \end{pmatrix}$$

$$\begin{aligned} y^T = \; & x_1 \cdot (A_{11} \;\; A_{12} \;\; A_{13} \;\; ...) \; + \\ & x_2 \cdot (A_{21} \;\; A_{22} \;\; A_{23} \;\; ...) \; + \\ & ... \end{aligned}$$

From an implementation perspective, the linearity of the operation enables the independent, sequential processing on the main-delta data layout. With $A = A^M + A^\Delta$, one obtains the superposition

$$y^T = x^T \cdot (A^M + A^\Delta)^T = \underbrace{x^T \cdot A^{M,T}}_{①} + \underbrace{x^T \cdot A^{\Delta,T}}_{②}$$

We sketch the implementation of the `spGemv` sparse matrix-vector multiplication operator based on our main-delta architecture in algorithm 1.

Algorithm 1 The `spGemv` operator

```
 1: function spGemv(Matrix $A$, Vector $x$)
 2:     $y \leftarrow 0$                                   ▷ $y$: Target Vector
 3:     Main, Delta ← getImpl($A$)
    ▷ Main part ①
 4:     for $x_i \neq 0 \in x$ do
 5:         start ← Main.CSRIndex[$i$]
 6:         stop ← Main.CSRIndex[$i + 1$]−1
 7:         switch Main.iivc[$i$] do
 8:             case 0 continue
 9:             case 1
10:                 for start ≤ $k$ < stop do
11:                     if Main.ivvc[$i$] = 0 then continue
12:                     col ← Main.Col[k]
13:                     $y$[col] ← $y$[col] + $x_i \times$ Main.Val[k]
    ▷ Delta part ②
14:     for $0 \leq i <$ Delta.Size do
15:         if Delta.vc[$i$] = 0 then continue
16:         col ← Delta.Col[i]
17:         $y$[col] ← $y$[col] + $x$[Delta.Row[i]] × Delta.Val[i]
18:     return $y$
```

The first part of the code (line 4-13) shows the main multiplication part using the CSR index. For each non-zero vector element x_i, a index lookup for the corresponding matrix row i provides the start and end position of the containers in the main structure. Moreover, the II-VC bitvector is checked for each x_i in order to skip the meanwhile deleted matrix rows. The same check is performed in the inner loop for each element using the IV-VC bitvector before the target vector y is written.

The second part (line 14–17) of the algorithm iterates over the valid elements of the incremental delta structure and adds the product results to the respective element holder in the target vector. It is noteworthy that by using this implementation, neither the first nor the second part of algorithm 1 requires a search scan in contrast to naive column store approaches.

6. APPLICATIONS AND WORKFLOWS

We present two sparse matrix applications from different domains and show how they can be run our system: the lanczos algorithm for numerical eigenvalue calculation, taken from a theoretical nuclear physics analysis, and an inclusive breadth-first search on network graphs.

6.1 Nuclear Energy State Analysis

We briefly sketch a workflow from theoretical nuclear physics, which we use as a benchmark in the evaluation. In this analysis, the energy states of an atomic nucleus are determined by an eigenvalue calculation of a large, sparse Hamiltonian matrix that stems from a preprocessed nuclear physics theory simulation. The eigenvalue calculation is based on the Lanczos method sketched in algorithm 2.

Algorithm 2 Lanczos with importance truncation

```
 1: function getEnergyStates(Hamiltonian $H$)
 2:     · · ·                              ▷ $H$: Symmetric $n \times n$ Matrix
 3:     while not isConverged($\lambda_k$) do
 4:         $r1, r2, c1, c2$ ← selectDel($H$)
 5:         delete($H, r1, r2, c1, c2$)
 6:         $\lambda_k$ ← lanczos($H$)
 7:     · · ·

 8: function lanczos(Hamiltonian $H$)
 9:     $T, v_k, v_{k-1}$ ← initialize($T, v_k, v_{k-1}$)          ▷ $v_i$: vectors
10:     for $1 \leq i < m$ do
11:         $w$ ← spGemv($H, v_k$)
12:         update($T, v_k, v_{k-1}, w$)
13:     $\lambda$ ← diagonalize($T, v_k, v_{k-1}$)       ▷ $T$: tridiagonal matrix
14:     return $\lambda$
```

The pseudocode is written in an abstract higher level language and shows the definition of the user defined functions getEnergyStates and lanczos. Depending on the DBMS, UDFs can either be written in SQL, or in a programming language like C++, R, or others. For our implementation, we used the *L* language [31] of the SAP HANA database, which is close to C. However, we emphasize that our work is generally independent from the language. The only requirement is that it includes the previously defined matrix API, which references our low-level operators that are implemented in the database engine. Furthermore, algorithm 2 contains calls to other UDFs, selectDel, update and diagonalize, which are defined elsewhere. We omit their definition and further domain-specific details of the code in order to outline the API calls to the database system.

The getEnergyStates procedure resembles the importance truncation method as described in Roth et al. [29]. The dots denote pre- and postprocessing steps of the analysis which contain domain-specific parameters and are left out for the sake of clarity. A crucial part of the analysis is a quantum state selection that we sketched via the selectDel function, which returns the coordinates of the matrix rows and columns that are selected for truncation. Since the Hamiltonian matrix is symmetric, the operation comprises the deletion of row-column pairs which is executed by calling the delete command of our matrix interface (line 5). After the deletion, the Lanczos function is called again. These two steps are repeated until a goodness criteria is achieved. Finally, the resulting eigenvalues λ were returned, which refer to the nuclear energy states.

The lanczos function itself is an iterative method, which effectively consists of a matrix-vector multiplication (line 11) and an update part (line 12). It processes the resulting vectors and forms an orthonormal basis of the eigenspace. For more details about the Lanczos method, we refer to the work of Stewart [32]. In the context of this paper, it is sufficient to know that the bottleneck of the lanczos function is the matrix-vector multiplication in line 11, which in this case is a call to our algorithm 1.

6.2 Breadth-First Search

In this part we show how queries on relational data from a non-numeric environment, especially with a graph-like topology, can be evaluated by exploiting our sparse matrix architecture. As an example we sketch an *inclusive* breadth-first search. We call it inclusive, because it returns all vertices that are discovered on paths with a length up to a certain traversal depth. However, this does not pose a restriction, since the algorithm can be rewritten in such a way that exclusively paths of a certain lenght are returned.

The breadth-first search is inherently similar to a matrix vector multiplication (algorithm 1), which is explained by the dualism between a graph and its adjacency matrix. This connection has already been topic of various works, and a comprehensive insight into this relation and its impact on graph algorithms is provided by the work of Kepner et. al. [22].

Fig. 3 shows an example table that represents friend connections of a social network graph. It sketches two attributes *Name* and *Friend_of* which denote the start and the target vertices of the graph, and thus the sparse adjacency matrix. Furthermore it contains an additional attribute *Since*, but we want to emphasize that the table may contain an arbitrary number of additional property attributes since they have no effect on the leading topological attributes. A common query on such a social networks table would for instance be: *'who are the friends of the friends of person X?'*, which is in fact a breadth-first search with depth=2 and start node *X*.

Algorithm 3 Breadth-First Search

```
 1: procedure BFSEARCH(GraphName, first, depth)
 2:     P ← 0                                    ▷ P: result set
 3:     Q ← first                                ▷ Q: set of vertices to visit
 4:     $x$ ← CONVERT(Q)
 5:     while depth>0 do
 6:         $y$ ← SPGEMV(GraphName,x)
 7:         $x \leftarrow y$
 8:         depth ← depth − 1
 9:     P ← CONVERT (y)
10:     return P
```

Algorithm 3 shows the breadth-first algorithm that internally calls algorithm 1. It effectively wraps an iterative matrix-vector multiplication by converting the start vertex set into a vector x and the target vertices vector y back into a result set. Internally, algorithm 1 multiplies the x_i values with the sparse matrix values $(A)_{ij}$, which usually refer to the edge weights. However, the graph must not necessarily have weighted edges. The additional floating point operation is nevertheless rather cheap, so that for unweighted graph edges, the value container *Main.Val* in line 13, algorithm 1, might just be filled with dummy values. A typical example for an algorithm working on weighted graphs is page rank [28]. It should be mentioned that we will also use a modified version of algorithm 1 in the evaluation. The modified version uses sparse vectors for x and y, i.e. , a tuple list of $\langle row, val\rangle$ pairs instead of a dense array. This is in particular a reasonable choice for very sparse matrices, since the number of non-zero entries of the product vector y depends on the matrix population density $\rho = N_{nz}/(n \times n)$ and the number of non-zero elements of x.

7. EVALUATION

In this section, we first compare the static query execution performance of our main-delta architecture against the different matrix representations of section 3. Thereafter, we evaluate how the performance on dynamic workloads using the example of section 6.1 improves against naive approaches.

The system for our prototype implementation contains an Intel Xeon X5650 CPU with 48 GB RAM. As there are currently no standardized benchmarks for large scale linear algebra operations in a database context, it is difficult to provide a comprehensive comparison against other systems. Therefore, we took the real world example workflows described in section 6 and compared how our presented architecture performs against alternative variants that could be implemented in a database system. In the context of this evaluation, we implemented a single-threaded version of the algorithms. However, as each of the structures is horizontally partionable, we expect that a parallelization does not change the qualitative results for the applied algorithms.

Table 1: Sparse matrices and graphs of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density (rounded) of each matrix. All matrices are square ($n \times n$.)

Name	Matrix Type	Dim.	N_{nz}	$\rho[\%]$
Mat1	NCSM	800	309 K	47.2
Mat2	NCSM	3440	2.930 M	24.7
Mat3	NCSM	17040	42.962 M	14.8
Gra1	Slashdot Netw.	77360	905 K	0.01
Gra2	Roadnet CA	1,971 K	5.533 M	10^{-6}

Tab. 1 lists the matrix data sets which we use in the evaluation. These include three Hamiltonian matrices from a nuclear science simulation of different scale (Mat1, Mat2 & Mat3), a social (Gra1) and a street network graph (Gra2). The Hamiltonian matrices stem from a no core shell model (NCSM) simulation and were provided by the theoretical nuclear physics research group of the Technical University of Darmstadt[4]. The graphs are taken from the SNAP graph library[5].

7.1 Static Algorithm Execution

Fig. 6 shows the relative performance comparison of algorithm 1 using our columnar main-delta sparse matrix architecture against the following representations:

- **Pure CSR representation**: We take the immutable CSR representation as a baseline for the algorithmic performance for the static experiments.
- **Triple Representation**: The pure triple representation serves as naive alternative approach for database-integrated algebra on mutable sparse matrices.
- **Dense Representation**: The dense representation has been included in all of the following cases only for illustrational purposes only, since it is not scaling and for most sparse matrices its memory consumption is order of magnitudes higher, as for example $10^5 x$ for Gra2.

In this experiment, we subsequently set values of random matrix elements. We chose randomly the matrix coordinates in order to

[4]http://theorie.ikp.physik.tu-darmstadt.de/tnp/index.php
[5]http://snap.stanford.edu/data/index.html

Final edited form was published in "SSDBM '14: Conference on Scientific and Statistical Database Management. Aalborg 2014", Art. Nr. 11, ISBN 978-1-4503-2722-0
https://doi.org/10.1145/2618243.2618254

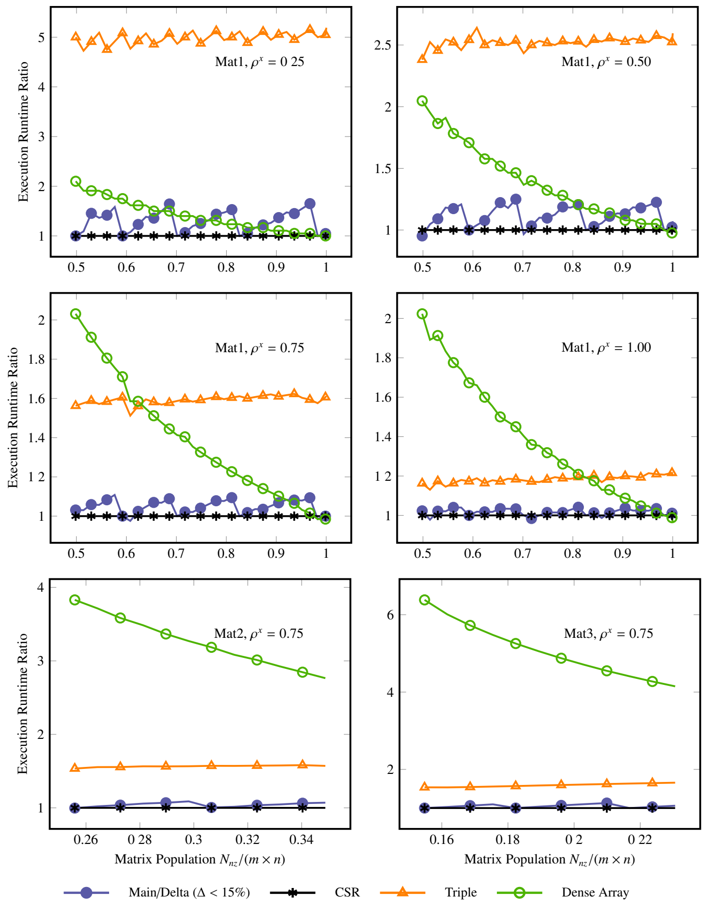

Figure 6: Algorithm 1 runtime performance comparison of the main-delta architecture against static CSR, a pure triple table and a dense array using different matrices and x-vectors with a varying population density ρ^x. For each plot, the corresponding sparse matrix was filled with nonzero elements between consecutive query executions, which results in an increasing matrix population density along the x-axis. The delta merge threshold was set to $\Delta_T = 15\%$

get an unbiased perception of the `set` performance. The varying matrix population density ρ is denoted along the x-axis of the plots, which reaches up to a complete occupation (Mat1) with nonzero elements ($\rho = 100\%$). The saw-tooth line belongs to the main-delta representation. Its shape is reasoned by the dynamically growing number of non-zero elements. All elements that are inserted into the main-delta architecture are at first accommodated by the delta triple representation. Thus, the delta size continuously increases until a certain occupation threshold Δ_T is reached, which we set to 15%. Then, the delta part is merged into the main structure and the delta occupation shifts back to zero. The main-delta execution time is then effectively a superposition of the triple and CSR representation, i.e. $T_{\text{m/d}} = T_{\text{CSR}}((1-\Delta)\rho) + T_{\text{triple}}(\Delta\rho)$. The sort overhead for the pure CSR representation is not included in the static measurement, but it is taken into consideration in the second part of the evaluation.

The first four plots in of Fig. 6 differ in the number of nonzero elements of the vector x that takes part in the multiplication. This variable, which we call $\rho^x = N^x_{nz}/m$, has a significant influence on the algorithm 1 performance and becomes even more significant for algorithm 3 on the graph Gra1 data set. With increasing ρ^x the runtime performance of the triple representation approaches to that of CSR, which also explains why the slope of saw-tooth decreases with increasing ρ^x. If every element x_i is nonzero, the advantage of having the CSR index disappears, since each matrix row has to be visited either way. Hence, the remaining benefit of the CSR representation is solely its row-major ordering, which leads to a better cache locality. It is worthwhile mentioning, that even for completely dense matrices, the dense representation does not result in a better performance than using a CSR representation. This could be explained with the sequential single-pass access pattern of algorithm 1, which enables prefetching of both the column and the value container. Finally, the $O(N_{nz})$ behavior of algorithm 1 results in a $1/\rho$-convergence of the dense performance relative to CSR.

We carried out a similar measurement using the inclusive breadth-first search (algorithm 3) on both graphs (Gra1 and Gra2). Therefore, we left the graph matrices unmanipulated and solely varied the search depth parameter of algorithm 3, which is denoted along the x-axis of the plots. We remark that the main-delta architecture is (up to a negligibly deviation) equal to CSR in this measurement, since we consider an isolated query execution on static data under the condition that all data has been merged and is residing in the main structure.

Fig. 7 presents the execution runtimes of CSR and triple representation, each with the dense and the sparse version of the intermediate result vectors x, y. The noticeable influence of the x vector population density ρ^x, which referes to the number of discovered vertices Q in algorithm 3, on the overall algorithmic performance was already observed in the previous measurements in Fig. 6. This dependency is even more significant for the inclusive breadth-first search, since the start x vector only contains a single non-zero element. In this case, the dense array-based x implemention (DI) iterates over every zero entry, which is why algorithm 3 performs obviously worse for small depths than using the list-based sparse x implementation (SpI). However, there is a turning point, where the vector density of x reaches a certain density threshold ρ^x_T, the exact value of which depends on the details of the respective implementation. In our experiment, the turning point is reached between depth two and three for the social graph Gra1. In the analogous measurement on Gra2, the turning point depth is at a considerably larger depth, which exceeds the x-range of the plot.

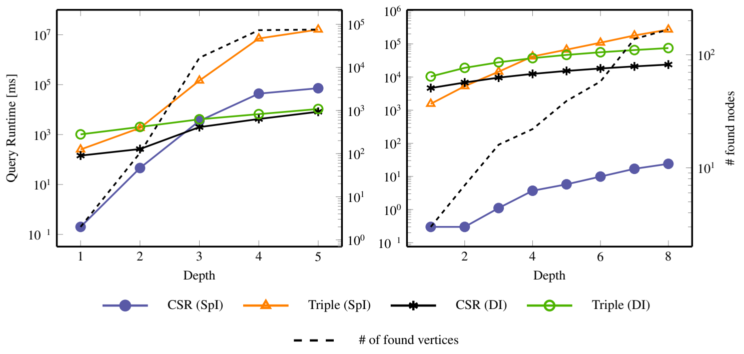

Figure 7: Comparison of the execution duration of algorithm 3 on graph Gra1 (left) and Gra2 (right) between a CSR and a triple table with respective sparse (SpI) and dense (DI) intermediate structures. The x-axis denote the depth parameter of algorithm 3.

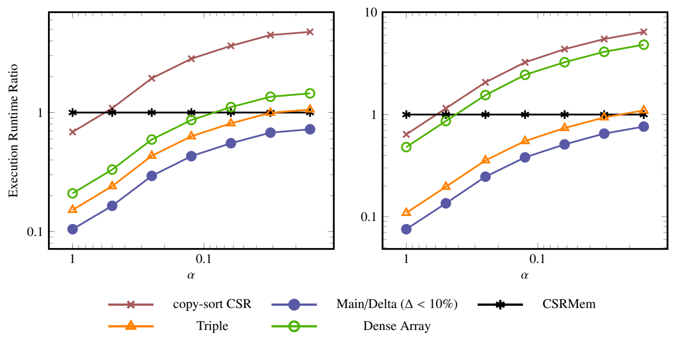

Figure 8: Comparison of the overall query throughput with $N_{\text{read}} + N_{\text{write}} = (1 + \alpha) \cdot 50$ using the different approaches relative to CSRMem on Mat1 (left) and Mat3 (right).

Finding the right spot to internally switch from a sparse to a dense representation of vector x provides clearly a tweak option that could be part of an optimizer. Nevertheless, we observe that independent from the intermediate vector representation, the inclusive breadth-first search using the CSR representation outperforms the naive triple approach by up to four orders of magnitude (right plot of Fig. 6, SpI). This can be reasoned with the index character of CSR, which is of particular importance for hypersparse problems, which are regularly found in graph contexts.

7.2 Dynamic Workload

In this part, we measured the throughput of dynamic workloads on large, sparse matrix data and compared our main-delta architecture against four different alternative approaches. These include the triple, the dense representation, and the following:

- **CSRMem**: This approach is is taken as a baseline for the following comparisons. A cached CSR version of the sparse matrix is kept in memory and is only rebuilt when an read query is requested, after a manipulation of the matrix has been executed.
- **Copy-sort CSR**: It is fair to compare against a naive approach, which includes copying and ordering of the data before each algorithm execution request. This is commonly done in science and analytic workflows, in order to transfer the data and bring it in shape for a third party system, where the actual calculations are executed.

In the first experiment, we interleave consecutive single element inserts, which we call *write* queries, with periodic executions of algorithm 1, which we call *read* queries. Moreover, we varied the ratio of the number of read queries N_{read} to the number of interleaved writes N_{write} according to following formula: $N_{\text{read}} + N_{\text{write}} = (1 + \alpha)N_{\text{read}}$, where α is the insert-to-query ratio $N_{\text{write}}/N_{\text{read}}$ which takes values from 0.02 to 1.

Fig. 8 presents the resulting relative query throughput of a mixed query workflow performed on matrices Mat1 and Mat3. To put it in other words, it shows the speedup factor of the overall execution time for $N_{\text{read}} + N_{\text{write}} = (1 + \alpha) \cdot 50$ queries using our main-delta architecture and other approaches compared relative to CSRMem. For $\alpha \to 1$, the main-delta architecture outperforms the naive CSRMem and copy-sort CSR approaches by orders of magnitude, whereas the difference to the triple and dense array representation is similar as in the static comparison, since these structures are both mutable and not significantly affected by interleaved inserts.

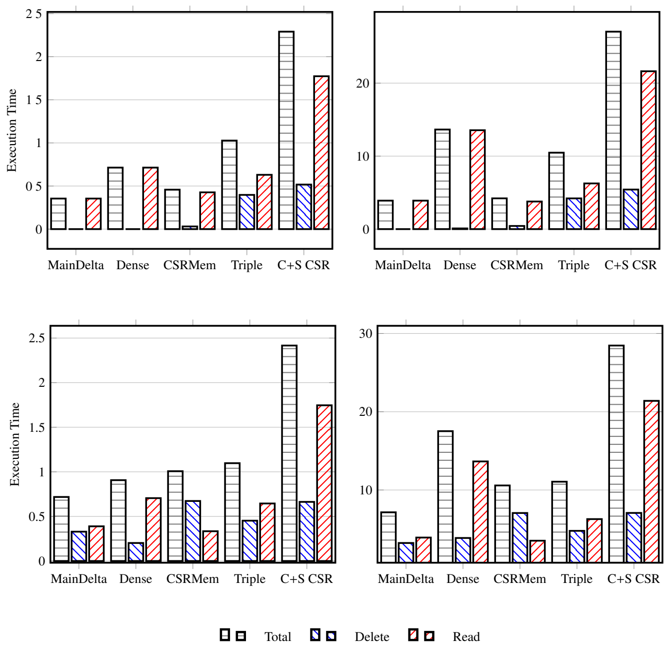

Figure 9: Average duration of a matrix k-rows (upper plot) and a k-column (lower plot) delete operation and the following algorithm 1 (20 times repeated) execution query. The left bar denotes the total query execution time. The operations were performed on matrix Mat1 (left) and Mat2 (right) with $k = 0.01m$ and $k = 0.01n$, respectively.

In the second experiment, we chose a scenario close to the workflow from theoretical nuclear physics that is sketched in algorithm 2. Although the original scenario comprises row- and column deletions, we decided to split the experiment in a row- and an column-exclusive deletion variant, in order to measure the impact of linearization on the respective deletions.

Fig. 9 shows the average duration of a k-rows delete operation ($k = 0.01m$), which is followed by a read query. It is observable that the deletion costs of our main-delta architecture are negligible, and the algorithm execution performance is nearly as good as the pure CSR representation. The low costs of the delete row operand on the dense and CSRMem representations can be explained by its efficient implementation, which essentially consists of copying of a contiguous memory section. The analogous measurements for column delete operations shows that our main-delta approach has the best deletion performance on Mat2. When the matrix is larger, the impact of cache misses by jumping over the row-major-ordered dense array increases significantly.

8. SUMMARY AND CONCLUSIONS

We have presented an approach to seamlessly integrate sparse matrices into a column-oriented in-memory database system with integrated API-level support for accessing, manipulating, and performing elementary linear algebra operations. The evaluation has shown that the algorithmic performance of our hybrid architecture, consisting of the read-optimized CSR main and the write-optimized, mutable triple delta representation, outperforms naive approaches, and deviates only negligible from using CSR-only. Moreover, the integer dictionary encoding of the columnar database architecture allows a flawless transition from pure numerical matrices to general structured data, for example graph data. From a system perspective, the use of the well-known CSR representation in the database engine opens the door to employ efficient numerical C++ libraries [5, 6, 4] as kernel, and hence, dispenses the need of time- and memory-consuming data transfers to third party software.

To put it in a nutshell, we showed that introducing mutability of sparse matrices without losing algorithmic performance yields an overall benefit for users of dynamic sparse matrix workloads. We have used database technologies to improve the overall performance for graph algorithms and science workflows, which bridges the gap between linear algebra and relational DBMS's.

9. ACKNOWLEDGMENTS

We thank the group of Prof. Roth from the Institute of Theoretical Nuclear Physics at TU Darmstadt for providing us insights into their domain-specific problem setting. Moreover, we express our gratitude to our fellow colleagues in Walldorf for fruitful discussions and support.

10. REFERENCES

[1] IBM Netezza Data Warehouse Appliances, http://www.ibm.com/software/data/netezza/.

[2] Oracle R Enterprise, http://www.oracle.com/.

[3] Array Query Language, http://www.xldb.org/arrayql/.

[4] Automatically Tuned Linear Algebra Software ATLAS, http://math-atlas.sourceforge.net/.

[5] Intel® Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[6] AMD® Core Math Library, http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

[7] Compressed Sparse Row Graph (CSR Graph), The Boost C++ Libraries, http://www.boost.org.

[8] Basic Linear Algebra Subprograms, http://www.netlib.org/blas/.

[9] D. J. Abadi, M. Stonebraker, E. Lau, A. Lin, and et al. C-Store: A Column-Oriented DBMS. *VLDB*, pages 553–564, 1995.

[10] R. R. Amossen and R. Pagh. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT*, pages 121–126. ACM, 2009.

[11] E. Anderson, Z. Bai, C. Bischof, S. Blackford, and et al. *LAPACK Users' Guide*. SIAM, third edition, 1999.

[12] R. Barrett, M. Berry, T. F. Chan, J. Demmel, , and et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[13] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and et al. The Multidimensional Database System RasDaMan. In *SIGMOD*, pages 575–577. ACM Press, 1998.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, and et al. *ScaLAPACK Users' Guide*. SIAM, 1997.

[15] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, pages 963–968. ACM, 2010.

[16] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and et al. MAD Skills: New Analysis Practices for Big Data. *VLDB*, 2(2):1481–1492, Aug. 2009.

[17] F. Färber, N. May, W. Lehner, P. Groß e, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[18] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, Dec. 1992.

[19] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, and et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, pages 231–242. IEEE, 2011.

[20] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, Jan. 1992.

[21] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, and et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *VLDB*, 5(12):1700–1711, Aug. 2012.

[22] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, Tools. SIAM, 2011.

[23] D. Kernert, F. Köhler, and W. Lehner. Bringing Linear Algebra Objects To Life in a Column-Oriented In-Memory Database. In *IMDM*, 2013.

[24] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a Query Language for Science Applications. In *EDBT/ICDT Workshop on Array Databases*, pages 1–12. ACM, 2011.

[25] M. Li, J. H. Badger, X. Chen, and et al. An Information-Based Sequence Distance and its Application to Whole Mitochondrial Genome Phylogeny. *Bioinformatics*, 17(2):149–154, 2001.

[26] R. Mittal and A. Al-Kurdi. LU-Decomposition and Numerical Structure for Solving Large Sparse Nonsymmetric Linear Systems. *J. Comput. Appl. Math.*, 43(1–2):131 – 155, 2002.

[27] G. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. IBM, 1966.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[29] R. Roth. Importance Truncation for Large-Scale Configuration Interaction Approaches. *Phys.Rev.*, C79:064324, 2009.

[30] Y. Saad. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations, Version 2, 1994.

[31] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*, pages 731–742. ACM, 2012.

[32] G. Stewart and USAF Office of Scientific Research. *Lanczos and Linear Systems*. Comp. Sc. Tech. Rep. Series. University of Maryland, 1991.

[33] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[34] M. Stonebraker, S. Madden, and P. Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Rec.*, 42(1):44–49, May 2013.

[35] K. Teranishi, P. Raghavan, J. Sun, and P. Michaleris. An Evaluation of Limited-Memory Sparse Linear Solvers for Thermo-Mechanical Applications. *Int. J. Num. Meth. Eng.*, 74(11):1690–1715, 2008.

[36] V. Valsalam and A. Skjellum. A framework for High-Performance Matrix Multiplication based on Hierarchical Abstractions, Algorithms and Optimized Low-Level Kernels. *CCPE*, 14(10):805–839, 2002.

[37] R. Yuster and U. Zwick. Fast Sparse Matrix Multiplication. *ACM Trans. Algorithms*, 1(1):2–13, July 2005.