A Relational Matrix Algebra and its Implementation in a Column Store

Oksana Dolmatova
University of Zürich
Zürich, Switzerland
dolmatova@ifi.uzh.ch

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Michael H. Böhlen
University of Zürich
Zürich, Switzerland
boehlen@ifi.uzh.ch

ABSTRACT

Analytical queries often require a mixture of relational and linear algebra operations applied to the same data. This poses a challenge to analytic systems that must bridge the gap between relations and matrices. Previous work has mainly strived to fix the problem at the implementation level. This paper proposes a principled solution at the logical level. We introduce the *relational matrix algebra* (RMA), which seamlessly integrates linear algebra operations into the relational model and eliminates the dichotomy between matrices and relations. RMA is closed: All our relational matrix operations are performed on relations and result in relations; no additional data structure is required. Our implementation in MonetDB shows the feasibility of our approach, and empirical evaluations suggest that in-database analytics performs well for mixed workloads.

ACM Reference Format:
Oksana Dolmatova, Nikolaus Augsten, and Michael H. Böhlen. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *Proc of the 2020 ACM SIGMOD International Conference on Management of Data, June 14-19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389747

1 INTRODUCTION

Many data that are stored in relational databases include numerical parts that must be analyzed, for example, sensor data from industrial plants, scientific observations, or point of sales data. The analysis of these data, which are not purely numerical but also include important non-numerical values, demand mixed queries that apply relational and linear algebra operations on the same data.

SIGMOD'20, June 14-19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389747

Dealing with mixed workloads is challenging since the gap between relations and matrices must be bridged. Current relational systems are poorly equipped for this task. Previous attempts to deal with mixed workloads have focused on the implementation level, for example, by introducing ordered data types; by storing matrices in special relations or key-value structures; or by splitting queries into their relational and matrix parts. This paper resolves the gap between relations and matrices.

We propose a principled solution for mixed workloads and introduce the *relational matrix algebra* (RMA) to support complex data analysis within the relational model. The goal is to (1) solve the integration of relations and linear algebra at the logical level, (2) so achieve independence from the implementation at the physical level, and (3) prove the feasibility of our model by extending an existing system. We are the first to achieve these goals: Other works focus on facilitating the transition between the relational and the linear algebra model. We eliminate the dichotomy between matrices and relations by seamlessly integrating linear algebra into the relational model. Our implementation of RMA in MonetDB shows the feasibility of our approach.

We define linear operations over relations and systematically process and maintain non-numerical information. We show that the relational model is well-suited for complex data analysis if ordering and contextual information are dealt with properly. RMA is purely based on relations and does not introduce any ordered data structures. Instead, the relevant row order for matrix operations is computed from *contextual information* in the argument relations. All relational matrix operations return relations with *origins*. Origins are constructed from the contextual information (attribute names and non-numerical values) of the input relations and uniquely identify and describe each cell in the result relation.

We extend the syntax of SQL to support relational matrix operations. As an example, consider a relation $rating$ with schema $(User, Balto, Heat, Net)$, that stores users and their ratings for the three films ("Balto", "Heat", and "Net", one column per film). The SQL query

```
SELECT * FROM INV(rating BY User);
```

orders the relation by users and computes the inversion of the matrix formed by the values of the ordered numerical

columns. The result is a relation with the same schema: The values of attribute *User* are preserved, and the values of the remaining three attributes are provided by matrix inversion (see Section 5 for details). The origin of a numerical result value is given by the user name in its table row and the attribute name of its column.

At the system level, we have integrated our solution into MonetDB. Specifically, we extended the kernel with relational matrix operations implemented over binary association tables (BATs). The physical implementation of matrix operations is flexible and may be transparently delegated to specialized libraries that leverage the underlying hardware (e.g., MKL [2] for CPUs or cuBLAS [1] for GPUs). The new functionality is introduced without changing the main data structures and the processing pipeline of MonetDB, and without affecting existing functionality.

Our technical contributions are as follows:

- We propose the *relational matrix algebra* (RMA), which extends the relational model with matrix operations. This is the first approach to show that the relational model is sufficient to support matrix operations. The new set of operations is *closed*: All relational matrix operations are performed on relations and result in relations, and no additional data structure is required.
- We show that matrix operations are *shape restricted*, which allows us to systematically define the results of matrix operations over relations. We define *row and column origins*, the part of contextual information that describes values in the result relation, and prove that all our operations return relations with origins.
- We implement and evaluate our solution in detail. We show that our solution is feasible and leverages existing data structures and optimizations.

RMA opens new opportunities for advanced data analytics that combine relational and linear algebra functionality, speeds up analytical queries, and triggers the development of new logical and physical optimization techniques.

The paper is organized as follows. Sect. 2 discusses related work. We introduce basics in Sect. 3 and introduce relational matrix algebra (RMA) in Sect. 4. We show an application example in Sect. 5, discuss important properties of RMA in Sect. 6, and its implementation in MonetDB in Sect. 7. We evaluate our solution in Sect. 8 and conclude in Sect. 9.

2 RELATED WORK

Relational DBMSs offer simple linear algebra operations, such as the pair-wise addition of attribute values in a relation. Some operations, e.g., matrix[1] multiplication, can be expressed via syntactically complex and slow SQL queries. The set of operations is limited and does not include operations whose results depend on the row order. For instance, there are no SQL solutions for inversion or determinant computation. Complex operations must be programmed as UDFs. Ordonez et al. [26] suggest UDFs for linear regression with a matrix-like result type. The UTL_NLA package [25] for Oracle DBMS offers linear algebra operations defined over UTL_NLA_ARRAY. UDFs provide a technical interface but do not define matrix operations over relations. No systematic approach to maintain contextual information is provided.

Luo et al. [20] extend SimSQL [8], a Hadoop-based relational system, with linear algebra functionality. RasDaMan [6, 22] manages and processes image raster data. Both systems introduce *matrices* as ordered numeric-only attribute types. Although relations and matrices coexist, operations are defined over different objects. Linear operations are not defined over unordered objects and they do not support contextual information for individual cells of a matrix.

SciQL [33, 34] extends MonetDB [23] with a new data type, `ARRAY`, as a first-class object. An array is stored as an object on its own. Arrays have a fixed schema: The last attribute stores the data values of a matrix, all other attributes are dimension attributes and must be numeric. Arrays come with a limited set of operations, such as addition, filtering, and aggregation, and they must be converted to relations to perform relational operations. The presence of contextual information and its inheritance are not addressed.

The MADlib library [16] for in-database analytics offers a broad range of linear and statistical operations, defined as either UDFs with C++ implementations or Eigen library calls. Matrix operations require a specific input format: Tables must have one attribute with a row id value and another array-valued attribute for matrix rows. Matrix operations return purely numeric results and cannot be nested.

Hutchison et al. [17] propose LARA, an algebra with tuple-wise operations, attribute-wise operations, and tuple extensions. LARA defines linear and relational algebra operations using the same set of primitives. This is a good basis for inter-algebra optimizations that span linear and relational operations. LARA offers a strong theoretical basis, works out properties of the solution, and allows to store row and column descriptions during the operations. The maintenance of contextual information is not considered for operations that change the number of rows or columns.

LevelHeaded [4, 5], an engine for relational and linear algebra operations, uses a special key-value structure: Each object has keys (dimension attributes) and annotations (value attributes). Dimension and value attributes are stored in a trie and a flat columnar buffer, respectively. Linear operations are available through an extended SQL syntax. Key values guarantee contextual information for rows. However,

[1] Some approaches support multi-dimensional arrays. Since we target linear algebra, we focus on two dimensions and use the term *matrix* throughout.

the trie key structure restricts relational operations: For example, aggregations of keys and join predicates over non-key attributes (i.e., subselects in SQL) are not allowed.

SciDB [31] is a DBMS that is based on arrays. Matrices and relations are implemented as nested arrays. SciDB focuses on efficient array processing and performs linear algebra operations over arrays. SciDB supports element-wise operations and selected linear operations, such as SVD. The system also offers relational algebra operations on arrays but cannot compete with relational DBMSs such as MonetDB in terms of performance. A systematic approach to maintain contextual information is not considered.

Statistical packages, such as R [27] and pandas [21], offer a broad range of linear and relational algebra operations over arrays. Each cell may be associated with descriptive information, but this information is not always inherited as part of operations (e.g., usv). No systematic solution for associating contextual information with numeric results is provided. The most important relational operations are supported, but even basic optimizations (e.g., join ordering) are missing.

The R package RIOT-DB [32] uses MySQL as a backend and translates linear computations to SQL. RIOT-DB addresses the main memory limitations of R. The optimization of constructed SQL statements yields inter-operation optimization. However, it is difficult (or sometimes impossible) to express linear algebra operations in SQL, and only a few simple operations, such as subtraction and multiplication, are discussed.

AIDA [11] integrates MonetDB and NumPy [3] and exploits the fact that both systems use C arrays as an internal data structure: To avoid copying NumPy data to MonetDB, AIDA passes pointers to arrays. Data copying is still needed to pass MonetDB results to NumPy since MonetDB does not guarantee that multiple columns are contiguous in memory, which is required by NumPy. AIDA offers a Python-like procedural language for relational and linear operations. Sequences of relational operations are evaluated lazily, which allows AIDA to combine and optimize sequences of relational operations. The optimization does not include linear algebra operations.

SystemML [14] offers a set of linear algebra primitives that are expressed in a high-level, declarative language and are implemented on MapReduce. SystemML includes linear algebra optimizations that are similar to relational optimizations (e.g., selecting the order of execution of matrix multiplications). The system considers only linear algebra operations.

3 PRELIMINARIES

This section presents notation for relations and matrices, and introduces the basic matrix algebra operations.

r

O	V	W
A	30	1
C	22	5
B	10	1

d

	1
1	D
2	B

e

	1	2
1	1	3
2	2	4

$d \square e$

	1	2	3
1	D	1	3
2	B	2	4

Figure 1: Relation r; matrices d, e, and $d \square e$

3.1 Relations

A relation r is a set of tuples r_i with schema $\mathcal{R}$. A schema, $\mathcal{R} = (A, B, \ldots)$, is a finite, ordered set of attribute names. A tuple $r_i \in r$ has a value from the appropriate domain for each attribute in the schema. We write $r_i.A$ to denote the value of attribute A in tuple r_i and $r.A$ to denote the set of all values $r_i.A$ in relation r. Ordered subsets of a schema, $\mathbf{U} \subseteq \mathcal{R}$, are typeset in bold. $|r|$ is the number of tuples in relation r.

Let r be a relation and $\mathbf{U} \subseteq \mathcal{R}$ be attributes that form a key of $\mathcal{R}$. We write $r^{\mathbf{U},k}$ to denote the k^{th} *tuple* of relation r sorted by the values of attributes $\mathbf{U}$ (in ascending order):

$$r_i = r^{\mathbf{U},k} \iff r_i \in r \wedge \\ |\{r_j \mid r_j \in r \wedge r_j.\mathbf{U} < r_i.\mathbf{U}\}| = k - 1 \quad (1)$$

The *column cast* $\triangledown U$ creates an ordered set L from the sorted values of an attribute U that forms a key in relation r:

$$L = \triangledown U \iff |L| = |r| \wedge \\ \forall 1 \le i \le |r| (L[i] = r^{U,i}.U) \quad (2)$$

The column cast is used to generate a schema from a set of values. We use this for operations `tra`, `usv`, and `opd` (see Table 2). The column cast is applicable if the cardinality of a list of attributes $\mathbf{U}$ is one.

Example 3.1. Consider relation r in Figure 1. The third tuple of relation r sorted by the values of attribute V is $r^{(V),3} = (A, 30, 1)$, the column cast of O is $\triangledown O = (A, B, C)$, and the values of attribute W are $r.W = \{1, 5, 1\}$.

We use set notation and apply it to bags. Bags can be ordered or unordered. To emphasize the difference, parentheses are used for ordered bags (or lists), e.g., (3,2,3), and curly braces for unordered bags, e.g., {3,2,3}. When transitioning from unordered to ordered bags, the order is specified explicitly.

3.2 Matrices

An $n \times k$ matrix m is a two-dimensional array with n rows and k columns. $|m|$ is the number of rows, $\#m$ the number of columns. The element in the i^{th} row and the j^{th} column of matrix m is $m[i, j]$; the i^{th} row is $m[i, *]$; the j^{th} column is $m[*, j]$.

We consider the operations from the R Matrix Algebra [28]: element-wise multiplication (EMU), matrix multiplication (MMU), outer product (OPD), cross product (CPD), matrix addition (ADD), matrix subtraction (SUB), transpose (TRA), solve equation (SOL), inversion (INV), eigenvectors (EVC), eigenvalues (EVL), QR decomposition (QQR, RQR), SVD – single value decomposition (DSV, USV, VSV), determinant (DET), rank (RNK), and Choleski factorization (CHF). Note that QR and SVD return more than one matrix, therefore we split the operations: QQR and RQR return matrix Q and matrix R of the QR decomposition, respectively; DSV, USV, and VSV return vector D with the singular values, matrix U with the left singular vectors, and matrix V with the right singular vectors of SVD, respectively.

The *matrix concatenation* of matrices m and n with k rows each returns a matrix h with k rows. The i^{th} row of h is the concatenation of the i^{th} row of m and the i^{th} row of n.

$$h = m \,\square\, n \iff |h| = |m| \wedge \forall 1 \leq i \leq |h|(h[i,*] = m[i,*] \circ n[i,*]) \quad (3)$$

The *schema cast* $\Delta\mathbf{U}$ of attributes $\mathbf{U}$ creates a matrix m (with a single column) from the attribute names of $\mathbf{U}$:

$$m = \Delta\mathbf{U} \iff \#m = 1 \wedge |m| = |\mathbf{U}| \wedge \forall 1 \leq i \leq |\mathbf{U}|(m[i,1] = \mathbf{U}[i]) \quad (4)$$

Example 3.2. Consider attributes $\mathbf{U} = (D, B)$. Matrix d in Figure 1 is the result of the schema cast $d = \Delta\mathbf{U}$. The result of concatenating matrix d and matrix e is $d \,\square\, e$. Note that the row and column numbers (cells shaded in gray) in the matrix illustrations are not part of the matrix.

Matrix operations are *shape restricted*, i.e., the number of result rows is equal to the number of *rows* of one of the input matrices (r), the number of *columns* of one of the input matrices (c), or *one* (1). The same holds for the number of result columns.

The dimensionality of result matrices defines the *shape type* of matrix operations. We write r_1 if the result dimensionality is equal to the number of rows in the first matrix, r_2 if the result dimensionality is equal to the number of rows in the second matrix, and r_* if the result dimensionality is equal to the number of rows in the first and second matrix (i.e., $\mathrm{r}_1 = \mathrm{r}_2$). The same notation holds for the number of columns. Table 1 summarizes the shape types of matrix operations.

Example 3.3. Matrix multiplication has shape type (r_1,c_2), which states that the number of result rows is equal to the number of rows of the first argument matrix, and the number of columns is equal to the number of columns of the second argument matrix. Matrix addition has shape type (r_*,c_*), which states that the number of result rows is equal to the number of rows of the first matrix and the number of rows of the second matrix.

Table 1: Shape types of matrix operations

Cardinalities	Shape type	Operations
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert i_1 \times i_1 \rvert$	(r_1,r_1)	USV
$\lvert i_1 \times j_1 \rvert, \lvert i_2 \times j_1 \rvert \rightarrow \lvert i_1 \times i_2 \rvert$	(r_1,r_2)	OPD
$\lvert i_1 \times i_1 \rvert \rightarrow \lvert i_1 \times i_1 \rvert$	(r_1,c_1)	INV, EVC, CHF
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert i_1 \times j_1 \rvert$	(r_1,c_1)	QQR
$\lvert i_1 \times j_1 \rvert, \lvert j_1 \times j_2 \rvert \rightarrow \lvert i_1 \times j_2 \rvert$	(r_1,c_2)	MMU
$\lvert i_1 \times i_1 \rvert \rightarrow \lvert i_1 \times 1 \rvert$	(r_1,1)	EVL
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert i_1 \times 1 \rvert$	(r_1,1)	VSV
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert j_1 \times i_1 \rvert$	(c_1,r_1)	TRA
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert j_1 \times j_1 \rvert$	(c_1,c_1)	RQR , DSV
$\lvert i_1 \times j_1 \rvert, \lvert i_1 \times j_2 \rvert \rightarrow \lvert j_1 \times j_2 \rvert$	(c_1,c_2)	CPD
$\lvert i_1 \times j_1 \rvert, \lvert i_1 \times 1 \rvert \rightarrow \lvert j_1 \times 1 \rvert$	(c_1,c_2)	SOL
$\lvert i_1 \times j_1 \rvert, \lvert i_1 \times j_1 \rvert \rightarrow \lvert i_1 \times j_1 \rvert$	(r_*,c_*)	EMU, ADD, SUB
$\lvert i_1 \times i_1 \rvert \rightarrow \lvert 1 \times 1 \rvert$	(1,1)	DET
$\lvert i_1 \times j_1 \rvert \rightarrow \lvert 1 \times 1 \rvert$	(1,1)	RNK

All operations of the matrix algebra are *shape restricted*. This follows directly from the definitions of the matrix operations [15]. The first column of Table 1 lists the relevant cardinalities from these definitions. We use shape restriction to determine the inheritance of contextual information. Shape restriction has also been used in size propagation techniques [7] for the purpose of cost-based optimization of chains of matrix operations.

4 RELATIONAL MATRIX ALGEBRA

To seamlessly integrate matrix operations into the relational model, we extend the relational algebra to the *relational matrix algebra* (RMA). For each of the matrix operations we define a corresponding relational matrix operation in RMA: emu, mmu, opd, cpd, add, sub, tra, sol, inv, evc, evl, qqr, rqr, dsv, usv, vsv, det, rnk, chf. We use upper case for matrix operations (e.g., TRA) and lower case for RMA operations (e.g., tra). RMA includes both relational algebra and relational matrix operations. The new operations behave like regular operations with relations as input and output.

For each argument relation, r, of a relational matrix operation one parameter must be specified: The *order schema* $\mathbf{U} \subseteq \mathcal{R}$ imposes an order on the tuples for the purpose of the operation. The attributes of the order schema must form a key[2]. The attributes of relation r that are not part of the order schema $\overline{\mathbf{U}}$, i.e., $\overline{\mathbf{U}} = \mathcal{R} - \mathbf{U}$ form the *application schema*. The application schema identifies the attributes with the data to which the matrix operation is applied.

The order schema $\mathbf{U} \subseteq \mathcal{R}$ splits relation r into four non-overlapping areas: *Order schema* $\mathbf{U}$; *order part* $r.\mathbf{U}$; *application schema* $\overline{\mathbf{U}}$; and *application part* $r.\overline{\mathbf{U}}$. The parts of r that do not

[2]Attributes that neither belong to the order schema nor the application schema must be dropped explicitly with a projection (or added to the order schema, thus, forming a super key).

include matrix values, i.e., the order and application schemas ($\mathbf{U}$ and $\overline{\mathbf{U}}$) and the order part ($r.\mathbf{U}$), form the *contextual information* for application part $r.\overline{\mathbf{U}}$. Intuitively, the order schema and application schema provide context for columns while the order part provides context for rows.

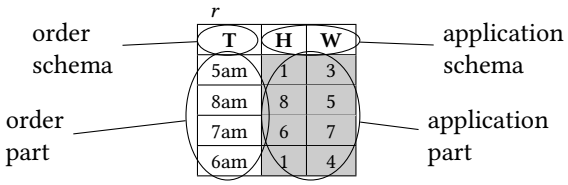


Figure 2: Structure of a relation instance

Example 4.1. Order schema $\mathbf{U} = (T)$ splits relation r in Figure 2 into four parts: Order schema $\mathbf{U} = (T)$, application schema $\overline{\mathbf{U}} = (H, W)$, order part $r.\mathbf{U} = r.(T) = \{5am, 8am, 7am, 6am\}$, and application part $r.\overline{\mathbf{U}} = r.(H, W) = \{(1, 3), (8, 5), (6, 7), (1, 4)\}$.

4.1 Matrix and Relation Constructors

Figure 3 summarizes our approach for the inv operation and example relation $r' = \sigma_{T>6am}(r)$: (1) Two matrix constructors define matrices m and n that correspond to order and application part of r, respectively; (2) INV inverses matrix n resulting in matrix h; and (3) the relation constructor combines $m \square h$ and $\mathcal{R}$ into result relation v.

Definition 4.2. (*Matrix constructor*) Let r be a relation, $\mathbf{U}$ be an order schema. The matrix constructor $\mu_{\mathbf{U}}(r)$ returns a matrix that includes the values of $r.\mathbf{U}$ sorted by $\mathbf{U}$:

$$m = \mu_{\mathbf{U}}(r) \iff |m| = |r| \wedge \forall 1 \leq i \leq |r|(m[i, *] = r^{\mathbf{U}, i}.\mathbf{U})$$

We use the complement notation $\overline{\mu}_{\mathbf{U}}(r)$ to denote the matrix that includes the values of $r.\overline{\mathbf{U}}$ sorted by $\mathbf{U}$.

Example 4.3. Consider Figure 3 with relation instance $r' = \sigma_{T>6am}(r)$ and schema $\mathcal{R} = (T, H, W)$. The matrix constructor $\overline{\mu}_T(r')$ returns matrix n.

Definition 4.4. (*Relation constructor*) Let m be a matrix with unique rows, and $\mathcal{R}$ be a relation schema with $\#m$ attributes. The relation constructor $\gamma(m, \mathcal{R})$ returns relation r with schema $\mathcal{R}$:

$$r = \gamma(m, \mathcal{R}) \iff |m| = |r| \wedge \forall t(t \in r \iff \exists 1 \leq i \leq |m|(t = m[i, *]))$$

Example 4.5. In Figure 3, a relation constructor is applied to schema $\mathcal{R}$ and the concatenated matrices $m \square h$ to construct the result relation: $v = \gamma(m \square h, \mathcal{R})$.

Matrix and relation constructors map between relations and matrices. We use constructors and matrices to define relational matrix operations and to analyze their properties. At the implementation level, constructors are very efficient since they split and combine lists of attribute names and *do not* access the data (cf. Section 7).

4.2 Relational Matrix Operations

Relational matrix operations offer the functionality of matrix operations in a relational context. The general form of a unary relational matrix operation is $\mathsf{op}_{\mathbf{U}}(r)$, where $\mathbf{U}$ is the order schema. A binary operation $\mathsf{op}_{\mathbf{U};\mathbf{V}}(r, s)$ has an additional order schema $\mathbf{V}$ for argument relation s.

The result of a relational matrix operation is a relation that consists of (a) the *base result* of the corresponding matrix operation, and (b) *contextual information*, appropriately morphed from the contextual information of the argument relations to reflect the semantics of the operation.

Definition 4.6. (*Base result*) Consider a unary relational matrix operation $\mathsf{op}_{\mathbf{U}}(r)$. The matrix that is the result of matrix operation $\mathsf{OP}(\overline{\mu}_{\mathbf{U}}(r))$ is the *base result* of $\mathsf{op}_{\mathbf{U}}(r)$. The base result for binary operations is defined analogously.

Example 4.7. Consider $\mathsf{inv}_T(\sigma_{T>6am}(r))$ in Fig. 3. The base result is matrix h, which results from $\mathsf{INV}(\overline{\mu}_T(\sigma_{T>6am}(r)))$.

Table 2 defines the details of how contextual information is maintained in relational matrix operations. All definitions follow the structure illustrated in Figure 3. A result relation is composed from order parts, base result, and schemas with the help of a relation constructor. For example, inv is defined according to its shape tape in Table 2: $\mathsf{inv}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \square \mathsf{INV}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \overline{\mathbf{U}})$, where $\mu_{\mathbf{U}}(r)$ are the rows of the order part, $\mathsf{INV}(\overline{\mu}_{\mathbf{U}}(r))$ is the base result, and $\mathbf{U} \circ \overline{\mathbf{U}}$ is the result schema.

Operations that have a different number of rows than any of the input relations add a new attribute C to the result relation. This attribute C is for contextual information (cf. Example 4.8): Its values are either the attribute names of the application schema of an input relation or the operation name. The operations add, sub, emu require union compatible application schemas and non-overlapping order schemas. Operations usv, opd, and tra construct the application schema of the result from the order schema of an input relation. Therefore the cardinality of the order schemas $\mathbf{U}$ of tra and usv, and $\mathbf{V}$ of opd must be one.

Example 4.8. Consider Figures 2 and 4. Figure 4a illustrates the result of $\mathsf{qqr}_T(r)$. The values of T define the ordering of tuples for this operation. The values of H and W are the values of matrix Q computed as part of the QR decomposition. Figure 4b illustrates the result of $\mathsf{tra}_T(r)$. The column cast $\triangledown T$ of ordering attribute T provides names for the attributes

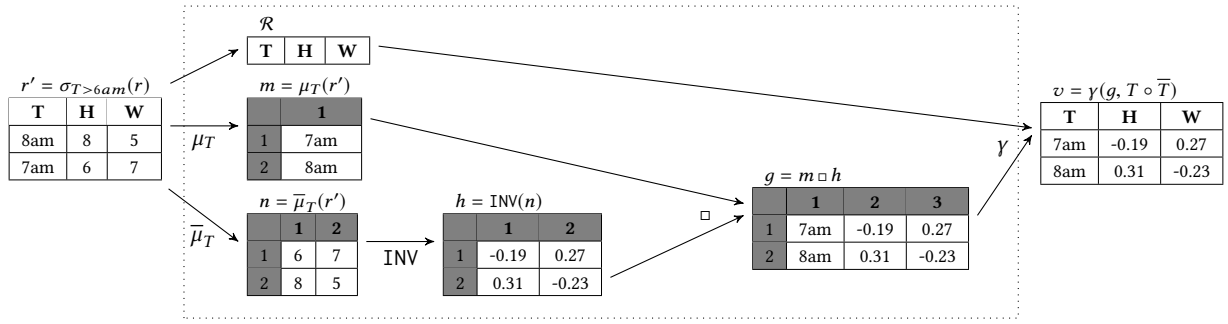


Figure 3: Structure of our solution for the inversion example, $v = \mathtt{inv}_T(\sigma_{T>6am}(r))$

Table 2: Splitting and morphing relations and matrices

Shape type	Operations	Definition
(r_1,r_1)	usv	$\mathsf{op}_U(r) = \gamma(\mu_U(r) \square \mathsf{OP}(\bar{\mu}_U(r)), \mathbf{U} \circ \triangledown\mathbf{U})$
(r_1,r_2)	opd	$\mathsf{op}_{U;V}(r,s) = \gamma(\mu_U(r) \square \mathsf{OP}(\bar{\mu}_U(r), \bar{\mu}_V(s)), \mathbf{U} \circ \triangledown\mathbf{V})$
(r_1,c_1)	inv, evc, chf, qqr	$\mathsf{op}_U(r) = \gamma(\mu_U(r) \square \mathsf{OP}(\bar{\mu}_U(r)), \mathbf{U} \circ \overline{\mathbf{U}})$
(r_1,c_2)	mmu	$\mathsf{op}_{U;V}(r,s) = \gamma(\mu_U(r) \square \mathsf{OP}(\bar{\mu}_U(r), \bar{\mu}_V(s)), \mathbf{U} \circ \overline{\mathbf{V}})$
$(r_1,1)$	evl, vsv	$\mathsf{op}_U(r) = \gamma(\mu_U(r) \square \mathsf{OP}(\bar{\mu}_U(r)), \mathbf{U} \circ (\mathsf{op}))$
(c_1,r_1)	tra	$\mathsf{op}_U(r) = \gamma(\Delta\overline{\mathbf{U}} \square \mathsf{OP}(\bar{\mu}_U(r)), (C) \circ \triangledown\mathbf{U})$
(c_1,c_1)	rqr, dsv	$\mathsf{op}_U(r) = \gamma(\Delta\overline{\mathbf{U}} \square \mathsf{OP}(\bar{\mu}_U(r)), (C) \circ \overline{\mathbf{U}})$
(c_1,c_2)	cpd, sol	$\mathsf{op}_{U;V}(r,s) = \gamma(\Delta\overline{\mathbf{U}} \square \mathsf{OP}(\bar{\mu}_U(r), \bar{\mu}_V(s)), (C) \circ \overline{\mathbf{V}})$
(r_*,c_*)	emu, add, sub	$\mathsf{op}_{U;V}(r,s) = \gamma(\mu_U(r) \square \mu_V(s) \square \mathsf{OP}(\bar{\mu}_U(r), \bar{\mu}_V(s)), \mathbf{U} \circ \mathbf{V} \circ \overline{\mathbf{U}})$
(1,1)	det, rnk	$\mathsf{op}_U(r) = \gamma(r \circ \mathsf{OP}(\bar{\mu}_U(r)), (C, \mathsf{op}))$

in the transposed relation. The result relation has a new attribute C whose values are the names of the attributes in the application schema of r. Note that all result relations come with sufficient contextual information for each value. For example, relation r in Figure 2 records that Humidity (H) was 1 at 6am, which is also recorded in the transposed relation in Figure 4b.

$\mathtt{qqr}_T(r)$

T	H	W
5am	0.1	0.5
6am	0.8	-0.4
7am	0.6	0.4
8am	0.1	0.7

(a) QR decomposition

$\mathtt{tra}_T(r)$

C	5am	6am	7am	8am
H	1	1	6	8
W	3	4	7	5

(b) Transpose

Figure 4: Examples of relational matrix operations

5 RMA IN ACTION

This section gives an application example with a mixed workload that combines relational and linear algebra operations. It maintains all data in regular relations and illustrates the importance of maintaining contextual information.

Consider relations u, f, and r in Figure 5. Relation u records name, state and year of birth of users; relation f records title, release year and director of films; relation r records user ratings for films. Tuple u_1 states that user Ann lives in California and was born in 1980; tuple f_1 states that film Heat was directed by Lee and was released in 1995; tuple r_1 states that Ann's ratings for Balto, Heat, and Net are, respectively, 1.5, 2.0, and 0.5.

u (user)

	User	State	YoB
u_1	Ann	CA	1980
u_2	Tom	FL	1965
u_3	Jan	CA	1970

f (film)

	Title	RelY	Director
f_1	Heat	1995	Lee
f_2	Balto	1995	Lee
f_3	Net	1995	Smith

r (rating)

	User	Balto	Heat	Net
r_1	Ann	1.5	2.0	0.5
r_2	Tom	0.0	0.0	1.5
r_3	Jan	4.0	1.0	1.0

Figure 5: Example database

The task is to determine how similar each of Lee's films is to any other film, based on the ratings from California users. The covariance [19] is used to compute this similarity.

In addition, we need relational algebra operations (e.g., selection σ, aggregation ϑ, rename ρ, and join $\bowtie$) to retrieve selected ratings and films, aggregate ratings, and combine information from different tables. The key observation is that a mixture of matrix and relational operations is required to determine the similarities of the ratings.

The solution in Figure 6 includes three key steps: Data preparation ($w1$), covariance computation ($w2$-$w7$), and retrieving Lee's films together with all similarities ($w8$). Note the seamless integration of linear and relational algebra[3]: The entire process frequently switches between linear and relational operations.

$$w1 = \pi_{U,B,H,N}(\sigma_{S='CA'}(u \bowtie r))$$
$$w2 = \vartheta_{AVG(B),AVG(H),AVG(N)}(w1)$$
$$w3 = \pi_{U,B,H,N}(\mathsf{sub}_{U;V}(w1, \rho_V(\pi_U(w1)) \times w2))$$
$$w4 = \mathtt{tra}_U(w3)$$
$$w5 = \mathtt{mmu}_{C;U}(w4, w3)$$
$$w6 = w5 \times \rho_M(\vartheta_{COUNT(*)}(w1))$$
$$w7 = \pi_{C,B/(M-1),H/(M-1),N/(M-1)}(w6)$$
$$w8 = \pi_{T,B,H,N}(\sigma_{D='Lee'}(w7 \bowtie_{C=T} f))$$

Figure 6: Computing the similarity of the ratings

In the following we discuss the algebra expressions in Figure 6. First, we join u and r to select ratings from California users ($w1$). Next, we compute the covariance using its standard definition [19]: $cov(X,Y) = \frac{1}{n-1}[(X-E[X])*(Y-E[Y])^{\mathsf{T}}]$. The expectation of an attribute, e.g., $E(H) = \vartheta_{AVG(H)}(\ldots)$, is computed via aggregation ($w2$). *Relational matrix operations*, sub, tra and mmu, are used to subtract ($X - E[X]$), transpose ($^{\mathsf{T}}$), and multiply ($*$) relations ($w3$, $w4$, $w5$). Next, we compute the unbiased covarinace ($w6$, $w7$). Finally, we join $w7$ and f to select Lee's films.

Figure 7 illustrates relations $w3$, $w4$, and $w8$. Consider transpose $\mathtt{tra}_U(w3)$ with order schema U and application schema $\overline{U} = (B,H,N)$. The result of this operation is a relation $w4$ with schema (C,*Ann*,*Jan*). The values of attribute C are the attribute names in the application schema of $w3$. Note that each operation preserves schema and ordering information as the crucial parts of contextual information. This makes it possible to interpret the tuples in result relation $w8$. For example, tuple z_1 states that Lee's film Balto has the smallest covariance to film Net.

6 PROPERTIES OF RMA

This section defines two crucial requirements for relational matrix operations. *Matrix consistency* guarantees that the result of a relational matrix operation can be reduced to the result of the corresponding matrix operation. *Origins* guarantee that each result relation includes sufficient inherited contextual information to relate argument and result relation. We prove that each relational matrix operations is matrix consistent and returns a relation with origins.

[3]We use the first character of the attribute name to refer to attributes.

w3

U	B	H	N
Ann	-1.25	0.5	-0.25
Jan	1.25	-0.5	0.25

w4

C	Ann	Jan
B	-1.25	1.25
H	0.5	-0.5
N	-0.25	0.25

w8

	T	B	H	N
z_1	B	1.56	-0.62	-2.5
z_2	H	-0.62	0.25	1

Figure 7: Steps during the computation

6.1 Matrix Consistency

Matrix consistency ensures that the result relation includes all cell values that are present in the base result and the order of rows in the base result can be derived from contextual information in the result relations. First, we define *reducibility* to transition from relations to matrices.

Definition 6.1. (*Reducibility*) Let r be a relation, $\mathbf{U}$ be an order schema. Relation r is *reducible* to matrix m iff m can be constructed from the attribute values of $\overline{\mathbf{U}}$ in relation r sorted by $\mathbf{U}$: $r \rightarrow_{\mathbf{U}} m \iff \overline{\mu}_{\mathbf{U}}(r) = m$

Example 6.2. Consider Fig. 3 with relation $r' = \sigma_{T>6am}(r)$, matrix n, and order schema T. From Example 4.3 we have $\overline{\mu}_T(r') = n$. Relation r' is reducible to matrix n since n can be constructed from the values of H and W in the argument relation sorted by T, i.e., $r' \rightarrow_T n$.

Definition 6.3. (*Matrix consistency*) Consider a unary matrix operation $\mathtt{OP}(m)$. The corresponding relational matrix operation op is *matrix consistent* iff for all relations r that are reducible to matrix m, the result relation $\mathtt{op}_{\mathbf{U}}(r)$ is reducible to $\mathtt{OP}(m)$:

$$\forall r, m, \mathbf{U}(r \rightarrow_{\mathbf{U}} m \implies \exists \mathbf{U}'(\mathtt{op}_{\mathbf{U}}(r) \rightarrow_{\mathbf{U}'} \mathtt{OP}(m)))$$

A binary relational matrix operation is matrix consistent if its result is reducible to the result of the corresponding binary matrix operation.

Example 6.4. Consider Figures 2 and 8 with relation r, matrix g, matrix $\mathtt{RQR}(g)$ and relation $\mathtt{rqr}_T(r)$.

- $r \rightarrow_T g$: Relation r is reducible to matrix g
- $\mathtt{rqr}_T(r) \rightarrow_C \mathtt{RQR}(g)$: relation $\mathtt{rqr}_T(r)$ is reducible to matrix $\mathtt{RQR}(g)$

6.2 Origins of Result Relations

The result of a relational matrix operation is a relation that, in addition to the base result, includes a *row origin* and a *column origin*. Origins (1) uniquely define the relative positioning of

g

	1	2
1	1	3
2	1	4
3	6	7
4	8	5

RQR(g)

	1	2
1	-10.1	-8.8
2	0.0	-4.6

$\mathtt{qqr}_T(r)$

C	H	W
H	-10.1	-8.8
W	0.0	-4.6

Figure 8: Example of matrix consistency

result values, (2) give a meaning to values with respect to the applied operation, and (3) establish a connection between argument relations of an operation and its result relation.

Example 6.5. Consider inversion and result relation v in Figure 3. Values 7*am* and 8*am* show that (1) value -0.19 precedes value 0.31 because 7*am* precedes 8*am*; (2) -0.19 is the inversion value for humidity and for time 7*am*; (3) value -0.19 in relation v is connected to value 6 in the argument relation since they have the same origins (7*am* and H).

Origins are either inherited order or application schemas from argument relations, or constants. The shape type of an operation determines the cardinality of inherited contextual information. The indices in the shape type specify the input relation, from which an origin is inherited. For example, if the first element of the shape type is c_1, the row origin is the schema cast of the application schema of the first argument relation. Indices $_*$ and $_2$ apply only to binary operations.

Definition 6.6. (*Origins*) Consider a unary, $v = \mathtt{op}_{\mathbf{U}}(r)$, or binary, $v = \mathtt{op}_{\mathbf{U};\mathbf{V}}(r, s)$, matrix consistent operation with shape type (x,y), base result m, and attribute list $\mathbf{U}'$ such that $v \rightarrow_{\mathbf{U}'} m$. Consider Table 3. $v.\mathbf{U}'$ is a *row origin* iff $v.\mathbf{U}'$ is equal to *ro* for the given shape type x. $\overline{\mathbf{U}'}$ is a *column origin* iff $\overline{\mathbf{U}'}$ is equal to *co* for the given shape type y.

Table 3: Definition of origins for shape type (x,y)

x	*ro*
r_1	$r.\mathbf{U}$
r_2	$s.\mathbf{V}$
c_1	$\Delta\overline{\mathbf{U}}$
c_2	$\Delta\overline{\mathbf{V}}$
r_*	($r.\mathbf{U}$, $s.\mathbf{V}$)
c_*	($\Delta\overline{\mathbf{U}}$, $\Delta\overline{\mathbf{V}}$)
1	$'r'$

y	*co*
r_1	$\triangledown\mathbf{U}$
r_2	$\triangledown\mathbf{V}$
c_1	$\overline{\mathbf{U}}$
c_2	$\overline{\mathbf{V}}$
r_*	$\triangledown\mathbf{U}$
c_*	$\overline{\mathbf{U}}$
1	$'op'$

Example 6.7. Figure 9 illustrates relation r and the origins for operations

- $\mathtt{rnk}_H(\pi_{H,W}(r))$ with shape type (1,1)
- $\mathtt{usv}_T(r)$ with shape type (r_1,r_1)
- $\mathtt{qqr}_{W,T}(r)$ with shape type (r_1,c_1)

Column origins (*co*) are marked by rectangles (all values inside a rectangle form together the column origin for the relation). *Row origins* (*ro*) are marked by ellipses.

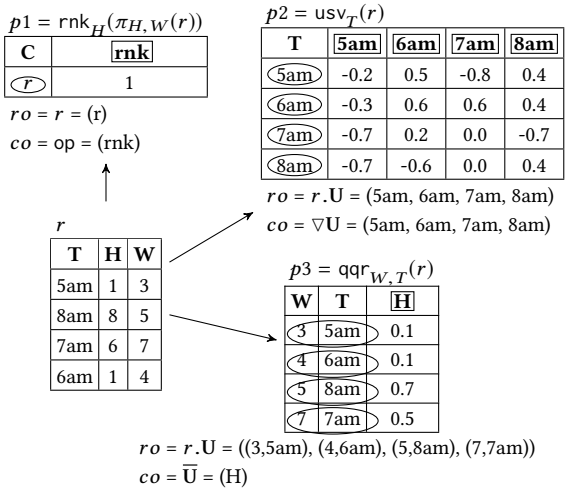


Figure 9: Examples of origins

For $p2 = \mathtt{usv}_T(r)$, we have $\mathbf{U} = (T)$, $\overline{\mathbf{U}} = (H, W)$, $\mathbf{U}' = (T)$, and $\overline{\mathbf{U}'}$ = (5am, 6am, 7am, 8am). The shape type of usv is (r_1, r_1) (see Table 2) this makes $p2.T = r.T$ a row origin and $\triangledown T$ = (5am, 6am, 7am, 8am) a column origin.

6.3 Correctness

THEOREM 6.8. *All relational matrix operations return matrix consistent relations with a row and column origin.*

We provide the proof for the theorem in our technical report [10].

The following example uses a sequence of tra operations to illustrate the importance of origins. A result relation with origins inherits sufficient contextual information, such that each value can be interpreted. Origins also carry sufficient information about the order of rows, such that in sequences of relational matrix operations no ordering information is lost between operations.

Example 6.9. Consider a relation instance r that is reducible to matrix n. Figure 10 illustrates the matrix expression TRA(TRA(n)) and the corresponding relational matrix expression $\mathtt{tra}_C(\mathtt{tra}_T(r))$. Operation $_T\mathtt{tra}(r)$ returns relation $r1$, which in addition to the application schema (attributes 5*am*, 6*am*, 7*am*, 8*am*) also includes attribute C, which is preserved together with the application schema.

7 IMPLEMENTATION

We discuss the integration of our solution into MonetDB. The implementation of relational matrix operations includes

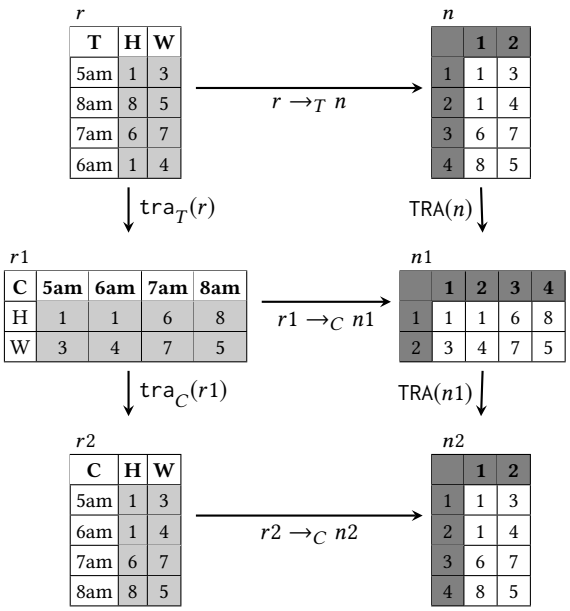


Figure 10: Origins and matrix consistency

the processing of contextual information and the computation of the base result. Contextual information is handled inside MonetDB, while the computation of the base result can be done in MonetDB or delegated to external libraries (e.g., MKL). The integration of each relational matrix operation requires extensions throughout the system, but does not change the query processing pipeline and no new data structures are introduced. To extend MonetDB with addition, QR decomposition, linear regression, and the transformation of numerical data to the MKL format we touch 20 (out of 4500) files and add 2500 lines of code.

7.1 MonetDB

MonetDB stores each column of a table as a binary association table (BAT). A BAT is a table with two columns: Head and tail. The head is a column with object identifiers (OID), while the tail is a column with attribute values. All attribute values of a tuple in a relation have the same OID value. Thus, a tuple can be constructed by concatenating all tail values with the same OID. MonetDB operations manipulate BATs, and relational operations are represented and executed as sequences of BAT operations. Example BAT operations are $B_1 * B_2$, B_1/B_2, and $B_1 - B_2$ for element-wise multiplication, division, and subtraction, and $sum(B)$ to sum the values in BAT B.

One important BAT operation is *leftfetchjoin* ($\downarrow$), which returns a BAT with OIDs sorted according to the order of OIDs of another BAT from the same relation. For instance, $X \downarrow Y$ returns BAT X, whose OIDs have the same order as OIDs of BAT Y. $X \downarrow X$ denotes X sorted by its own values.

7.2 RMA Integration

As a first step, we have extended the SQL parser to make the relational matrix operations available in the FROM clause of SQL [9]. The syntax `(r BY U)` specifies ordering for an argument relation r. As an example, consider relations r and s and ordering attributes **U** and **V**. The unary operation $\texttt{inv}_{\mathbf{U}}$ and the binary operation $\texttt{mmu}_{\mathbf{U;V}}$ are expressed as:

```
SELECT * FROM INV(r BY U);
SELECT * FROM MMU(r BY U, s BY V);
```

These basic constructs can be composed to build more complex expressions. For instance, folding $w5$, $w6$ and $w7$ from Figure 6 yields the RMA expression:

$$\pi_{C,B/(M-1),H/(M-1),N/(M-1)}(\texttt{mmu}_{C;U}(w4, w3) \times \rho_M(\vartheta_{COUNT(*)}(w1)))$$

The SQL translation of this expression is:

```
SELECT C, B/(M-1), H/(M-1), N/(M-1)
FROM MMU(w4 BY C, w3 BY U) AS w5
     CROSS JOIN
     ( SELECT COUNT(*) AS M FROM w1 ) AS t;
```

Algorithm 1 processes a node that represents a unary relational matrix operation $\texttt{op}_{\mathbf{U}}(r)$ and translates it to a list of BAT expressions. In lines 2 - 7, the BATs of relation r are split, sorted, and morphed to get BATs X with row origins and BATs Y with the application part. *Splitting* (lines 2 and 4) divides a relation into two parts: The application part, on which the matrix operations are performed, and the contextual information, which gives a meaning to the application part. BATs B are split into application part and order part according to **U**. *Sorting* (lines 3 and 4) determines the order of the tuples for a specific matrix operation. The order schema **U** is used to sort the BATs: BATs in **U** are sorted according to their values while the other BATs in B are sorted according to the OIDs of the BATs in **U**. The order is established for each operation based on the contextual values in the relation. *Morphing* (lines 5-7) morphs contextual information so that it can be added to the base result. Finally, the matrix operation is applied to Y (line 8). *Merging* (line 9) combines the result of a matrix operation with relevant contextual information and constructs the result relation with row and column origins. Merging and splitting are efficient operations that work at the schema level and do not access the data.

Example 7.1. Figure 11 illustrates Algorithm 1 for $v = \texttt{inv}_T(\sigma_{T>6am}(r))$. *Splitting*: input list $B = (T, H, W)$ is split into order list $D = (T)$ and application list $B \setminus D = (H, W)$. *Sorting*: BAT T is sorted, producing G. Then, (H, W) are sorted according to G returning $(H \downarrow T, W \downarrow T)$. *Morphing*:

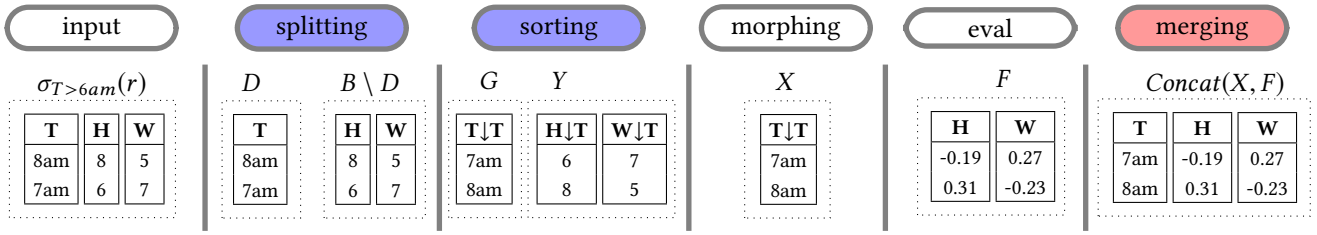


Figure 11: Splitting, sorting, morphing, merging for query $v = \texttt{inv}_T(\sigma_{T>6am}(r))$

Algorithm 1: UnaryRMA(op, **U**, r)

```
1 B ← BATs(r); Y ← {} ;
2 D ← {b|b ∈ B, b is in order schema U} ;
3 G ← sort(D) ;
4 for b ∈ B \ D do Y ← Y ∪ b↓G;
5 if ShapeType(op) ∈ {(r,r), (r,c), (r,1)} then X ← G;
6 else if ShapeType(op) ∈ {(c,r), (c,c)} then X ← newBAT(Y);
7 else X ← newBAT(r);
8 F ← eval(op, Y);
9 return Concat(X, F);
```

Algorithm 2: INV(B)

```
1 n ← B.length; BR ← IDmatrix(n);
2 for i = 1 to n do
3     v1 ← sel(Bi, i); Bi ← Bi/v1; BRi ← BRi/v1;
4     for j = 1 to n do
5         if i ≠ j then
6             v2 ← sel(Bj, i); Bj ← Bj − Bi ∗ v2;
7             BRj ← BRj − BRi ∗ v2;
8 return BR;
```

Since inv is of shape type (r_1,c_1), row contextual information is the order schema: $X = T \downarrow T$. *Merging*: BATs X are concatenated with BATs F to form the result.

7.3 Computing the Base Result

Line 8 of Algorithm 1 calls the procedure that computes the matrix operation. The computation can be done either in the kernel of MonetDB or by calling an external library (e.g., MKL [2]). Calling an external library requires copying data from BATs to the external format and copying the result back. The query optimizer decides about external library calls based on the complexity of the operation, the amount of data to be copied, and the relative performance of the matrix operation in MonetDB compared to the external library.

The no-copy implementation of matrix operations in the kernel of MonetDB is performed over BATs directly. Essentially, standard algorithms must be reduced to BAT operations. The process of reducing is highly dependent on the operation. The goal is to design algorithms that access entire columns and minimize accesses to single elements of BATs. To achieve this standard, value-based algorithms must be transformed to vectorized BAT operations.

Algorithm 2 illustrates the reduction for the Gauss Jordan elimination method for the INV computation. The algorithm takes a list of BATs $B = (B_1, B_2, .., B_n)$ and returns the inversion as a list of BATs BR of the same size. Function $IDmatrix(n)$ creates a list of BATs that represents the identity matrix of size $n \times n$. The selection operation $sel(B, i)$ returns the i^{th} value in B. With the exception of the sel operation, all operations are standard MonetDB BAT operations that are also used for relational queries. For example, the operation on $B_i \leftarrow B_i/v_1$; divides each element of a BAT by a scalar value.

8 PERFORMANCE EVALUATION

Setup. All runtimes are averages over 3 runs on an Intel(R) Xeon(R) E5-2603 CPU, 1.7 GHz, 12 cores, no hyper-threading, 98 GB RAM (L1: 32+32K, L2:256K, L3:15360K), Debian 4.9.189.

Competitors. We empirically compare the implementations of our relational matrix algebra (*RMA+*[4]) with the statistical package *R*, the array database *SciDB*, and two state-of-the-art in-database solutions, *AIDA* [11] and *MADlib* [16]. (1) We implemented *RMA+* in MonetDB (v11.29.8) with two options for matrix operations: (a) BATs (RMA+BAT): No-copy implementation in the kernel of MonetDB; (b) MKL (RMA+MKL): Copy BATs to an MKL (v2019.5.281) [2] compatible format (contiguous array of doubles), then copy the result back to BATs. We execute linear operations (add, sub, emu) on BATs and use MKL for more complex operations. When the matrices do not fit into main memory we switch to BATs. Due to the full integration of RMA+, MonetDB takes care of core usage and work distribution, and all cores are used for relational and for matrix operations. (2) *SciDB* [31] uses an array data model, and queries are expressed in the

[4]The implementation can be found here: https://github.com/oksdolm/RMA

high-level, declarative language AQL (Array Query Language) [30]. SciDB uses all available cores. (3) *AIDA* is a state-of-the-art solution for the integration of matrix operations into a relational database and was shown to outperform other solutions like Spark or the pandas library for Python [11]. AIDA executes matrix operations in Python and offers a Python-like syntax for relational operations, which are then translated into SQL and executed in MonetDB (v11.29.3). AIDA uses all cores both in MonetDB and in Python. We also integrate our solution into MonetDB, which makes AIDA a particularly interesting competitor. (4) MADlib [16] (v1.10) provides a collection of UDFs for PostgreSQL (v9.6) for in-database matrix and statistical calculations. MADlib does not use multiple cores, which affects its overall performance. (5) The *R* package (v3.2.3) is highly tuned for matrix operations and is a representative of a non-database solution. R performs all relational operations with data.tables structures and transforms the relevant columns to matrices to compute the matrix operations. An alternative approach to use character matrices for all operations is very inefficient (cf. Section 8.5). R uses all cores for matrix operations but runs relational operations on a single core.

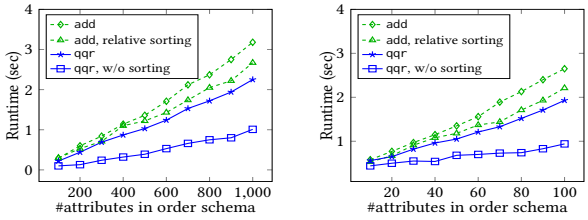
Data. BIXI [18] stores trips and stations of Montreal's public bicycle sharing system, years 2014-2017. DBLP [24] stores authors with their publication counts per conference as well as conference rankings. The synthetic dataset used in the experiment to measure the effect of sparsity includes values between 0 and 5,000,000. All other synthetic datasets include real-valued numeric attributes with uniformly distributed values between 0 and 10,000.

8.1 Maintaining Contextual Information

A salient feature of our approach is that contextual information is maintained during matrix operations. We analyze the scalability of maintaining context and study an optimization that avoids sorting. To this end, we generate relations with a single application column and an increasing number of order columns. We compute `add` and `qqr` on these relations. Since `add` and `qqr` are inexpensive for single column matrices, the main cost is the maintenance of the order part.

To handle contextual information we split, sort, morph, and merge lists of BATs (cf. Section 7.2). Sorting is the most expensive operation. Fortunately, sorting is not always necessary. For example, permuting the input rows for the `qqr` operation will affect the order of the result rows, but will not change their values. Therefore, sorting is not required. In element-wise operations like `add`, `emu`, or `sol`, only the relative order of the rows in the two input relations matters. Thus, only the order part of the second relation requires sorting (to get the same order).

Figure 12 shows the results. (1) Handling contextual information is efficient and scales to large numbers of attributes. (2) The optimized operators that (partially) avoid sorting clearly outperform their non-optimized counterparts.



(a) 100K tuples **(b) 1M tuples**

Figure 12: Handling contextual information

Note that a number of operations (`cpd`, `sol`, `rqr`, `dsv`, `tra`, `det`, `rnk`) do not preserve row context since the number of rows changes. Instead, a single column with predefined values (operation name or attribute names of the application schema) is created, which is negligible in the overall runtime.

8.2 Wide and Sparse Relations

Wide relations. Current databases scale better in the number of tuples than in the number of attributes. We test our RMA+ implementation in MonetDB on wide relations. We generate relations with 1000 tuples, one order attribute, and a varying number of application attributes. In Table 4, we increase the number of attributes from 1K to 10K and measure the runtime of the `add` operation. MonetDB can handle wide relations with several thousands of attributes, even though the runtime per column increases with the attribute number.

Table 4: `add` over wide relations in RMA+

#attr	1K	2K	3K	4K	5K	6K	7K	8K	9K	10K
sec	0.6	2.2	4.8	8.8	13.4	20	27	36	47	62

Sparse relations. We analyze the effect of MonetDB's built-in compression on relations with many zeros. We add two relations (5M tuples, one order, 10 application attributes) with uniformly distributed non-zero values (range 1-5M). In Table 5 we increase the percentage of zero values (position of zeros is random) and measure the runtime: The `add` operation on sparse matrices is up to two times faster than the same operation on dense matrices. Thus, RMA+ leverages MonetDB's compression features.

Table 5: `add` over sparse relations in RMA+

%	0	10	20	30	40	50	60	70	80	90	100
sec	1.68	1.60	1.49	1.41	1.33	1.25	1.16	0.99	0.94	0.89	0.76

8.3 RMA+ vs. Non-Database Approaches

We study the scalability of RMA+ to large relations and compare to *R* as a non-database solutions for matrix operations. In Table 6 we measure the runtime for `qqr` on tables with up to 100M tuples and 70 attributes in the application schema. For relations up to a size of 50Mx40, RMA+ delegates the matrix computation to MKL; the runtime includes copying the data. RMA+ is consistently faster than R since MKL can better leverage the hardware. R fails for sizes above 50Mx40 since it runs out of memory. In RMA+ we switch to the BAT implementation, which leverages the memory management of MonetDB. The Gram-Schmidt `qqr` baseline [13] that we implemented over BATs is slower than the MKL algorithm (e.g., 834 vs. 61.4 sec for 50Mx40), which explains the increase in runtime. RMA+ scales to large relations that do not fit into memory (e.g., relation size 100Mx70 requires 56GB).

Table 6: Runtimes of `qqr` in seconds in R and RMA+

	10 attr		40 attr		70 attr	
System	**R**	**RMA+**	**R**	**RMA+**	**R**	**RMA+**
5M tup	3.5	2.1	20	6.6	47	11.6
50M tup	37	21.3	221	61.4	fail	2018
100M tup	74	40	fail	1690	fail	4064

8.4 RMA+ vs. Array Databases

We study the performance of RMA+ vs. SciDB [31] as a representative of array databases. We compute `add` on two matrices with 10 columns and a varying number of rows, followed by a selection [5]. The resulting runtimes for Ubuntu are shown in Table 7. RMA+ outperforms SciDB by more than an order of magnitude. RMA+ performs addition directly over pairs of relations, while SciDB must compute a so-called array join [30] over the input arrays in order to add their values.

Table 7: `add` followed by a selection: RMA+ vs. SciDB

#tuples	1M	5M	10M	15M
RMA+	4.6s	24.4s	1m18s	1m39s
SciDB	1m21s	7m6s	13m2s	18m23s

8.5 Overhead of Data Transformation

We investigate the overhead of data transformation for various matrix operations in a mixed relational/matrix scenario.

RMA+ is free to execute matrix operations directly on BATs or rearrange the numerical data in main memory and delegate the matrix operations to specialized packages like MKL [2]. R does not enjoy this flexibility: R uses the matrix data type for matrix operations and the data.tables storage structure for relational operations. While data.tables supports simple linear operations like linear model construction, the data must be transformed to the matrix type for more complex operations like `CPD`, `OPD`, or `MMU`. Matrices cannot store a mix of numerical and non-numerical values, which is required when working with tables; R offers character matrices, but they are very inefficient, e.g., joining trips and stations in the BIXI dataset takes 40 sec for the character matrix type and less than 2 sec for data.tables.

Figure 13 shows the percentage of time spent for data transformations on relations with 50 columns and a varying number of rows (100k to 500k). For R we measure the time of transforming the relation from data.table to matrix and back as a percentage of the overall query time, which includes the actual matrix operation. For RMA+ we measure the time share for copying the data from a list of BATs to a contiguous, one-dimensional array for MKL, and for copying the result back; the overall runtime in addition includes the matrix computation in MKL (but excludes the MonetDB query pipeline of query parsing, query tree creation, etc.).

#rows	(#columns = 50)					
500K	81	75	64	21	7	7
300K	79	77	63	21	7	7
100K	84	74	69	23	9	10
	ADD	EMU	MMU	QQR	DSV	VSV

(a) Data.table and matrix

#rows	(#columns = 50)					
500K	92	92	86	53	44	43
300K	91	91	86	55	45	40
100K	86	86	80	48	37	35
	ADD	EMU	MMU	QQR	DSV	VSV

(b) List of BATs and 1D array

Figure 13: Data transformation share: (a) R, (b) RMA+

Clearly, the overhead of transforming data matters for both R and RMA+. We draw the following conclusions: (a) Transforming data between data structures is costly. (b) For simple operations like `ADD` and `EMU`, the transformation overhead dominates the overall runtime (up to 92%). (c) For complex operations, the performance of the matrix operation dominates the overall runtime.

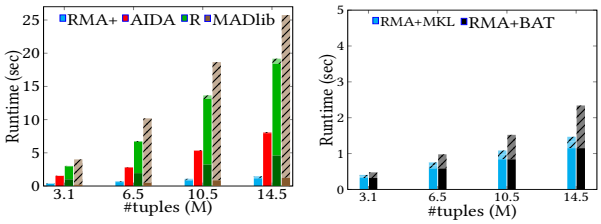
8.6 Efficiency for Mixed Workloads

We analyze four workloads that require a mix of relational operations and matrix operations, and we compare our implementation of RMA (RMA+) to its competitors (R, AIDA, MADlib). The workloads stem from applications on our real-world datasets and differ in the complexity of relational vs. matrix part. On the BIXI dataset, we compute (1) the linear regression between distance and duration for individual trips, and (2) journeys connecting up to 5 trips; on DBLP we compute the (3) covariance between conferences based on the publication counts per conference and author; (4) on a synthetic dataset based on BIXI we count trips per rider.

[5] We run this experiment on Ubuntu 14.04 since SciDB does not support Debian; Ubuntu runs on a server with 4 cores and 16GB of RAM.

(1) Trips – Ordinary Linear Regression. Trips in BIXI include start date and start station, end date and end station, duration, and a membership flag for the rider; stations have a code, a name, and coordinates. At the level of relations, we need to perform the following data preparation steps: (a) Aggregate the trips and select those trips that were performed at least 50 times; (b) join trips and stations to retrieve the station coordinates and compute the distance. We use the OLS method [29] to compute the linear regression between distance and duration. OLS uses cross product, matrix multiplication, and inversion: $\texttt{MMU}(\texttt{INV}(\texttt{CPD}(A, A)), \texttt{CPD}(A, V))$, where A is the matrix with the independent variables, and V is the vector with the dependent variable.

Figure 14a shows the runtime results for trips reported in the years 2014 (3.1M trips), 2014-2015 (6.1M trips), 2014-2016 (10.5M trips), and 2014-2017 (14.5M trips), respectively. The input data consists of numeric and non-numeric types such as date and time. We break the runtime down into data preparation (solid area of the bar) and matrix computation time (dashed light area) for RMA+, R, and AIDA; for R we also show the load time from a CSV file (dark area). RMA+ and AIDA outperform R and MADlib in all scenarios. R performs poorly on the relational operations of the data preparation step: The join implementation of R does not leverage multiple cores, and R lacks a query optimizer, which adversely affects the relational performance. MADlib is outperformed by all other solutions due to the slow computation of the linear regression. RMA+ outperforms AIDA on all datasets. Although both RMA+ and AIDA compute the relational operations in MonetDB, RMA+ is up to 6.3 times faster: While AIDA passes pointers to access numerical Python data in MonetDB, this does not work for other data types (e.g., date, time, string) due to different storage formats [12]. Therefore, expensive data transformations must be applied.

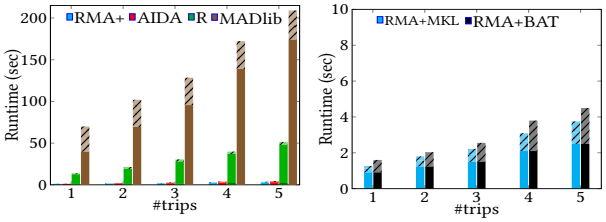


(a) System Comparison **(b) RMA+BAT vs RMA+MKL**

Figure 14: Trips (Ordinary Linear Regression)

(2) Journeys – Multiple Linear Regression. We compose trips that meet in a station into journeys. We start from 15M one-trip journeys of the form (start station, end station, duration); all attributes are numerical. During data preparation, we perform joins to create journeys of up to five trips, select those that appear at least 50 times, and join stations with their coordinates to compute the distances between subsequent stations in a journey. At the matrix level, we do a multiple linear regression analysis with the distances as independent variables and the overall duration as the dependent variable.

Figure 15a shows the runtime for journey lengths of 1 to 5 trips (i.e., 1 to 5 independent variables). The solid part is the time for data preparation (relational operations); the dashed light part is the time for multiple linear regression (matrix operations). RMA+ and AIDA again outperform R on the relational part of the query. The relational part operates on purely numerical data and AIDA shows comparable join performance to RMA+. MADlib spends about two third of the relational runtime on distance computations and is therefore slower than its competitors also on the relational part.



(a) System Comparison **(b) RMA+BAT vs RMA+MKL**

Figure 15: Journeys (Multiple Linear Regression)

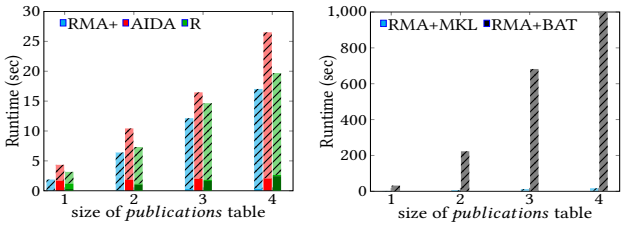
(3) Conferences – Covariance Computation. We compute the covariance between conferences with A++ rating to lower rated conferences based on the number of publications per author and conference. The data includes two tables. *ranking* stores a rating (e.g., A++, A+, B) for each conference. *publication* stores the number of publications per author and conference; the first attribute is the author, the other attributes are conference names (i.e., the result of SQL PIVOT over a count-aggregate by conference and author). The query computes the covariance matrix on *publication* and joins the result with *ranking* to select A++ conferences.

We measure the runtime for *publication* tables of increasing sizes: (1) 337363x266 (i.e., 337363 authors and 266 conferences), (2) 550085x519, (3) 722891x744, and (4) 876559x882. The *ranking* table stores 882 tuples. Note that the number of result rows of covariance is identical to the number of input columns, e.g., covariance of *publications* with 266 columns returns a relation (or matrix) of size 266x266.

Figure 16a shows the runtime results for RMA+, R, and AIDA. MADlib runs for 77, 429, 1086, resp. 1814 seconds on the different relation sizes and, thus, is omitted from the figure. In all systems, the covariance computation dominates the overall runtime with at least 90%. Since AIDA does not support covariance, we implement covariance via cross product [19] in all algorithms except MADlib, which has a `cov()` function but does not support cross product. For the cross

product in RMA+ we use the routine `cblas_dsyrk()` since the result of multiplication is symmetric, in AIDA we use `a.t@a`, in R we use `crossproduct`[6].

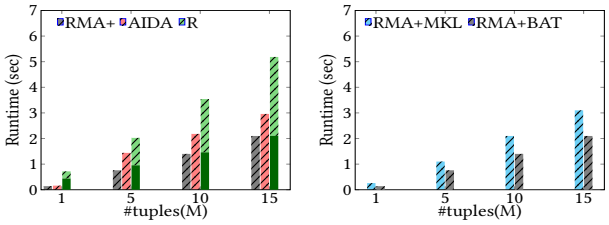
Note that the covariance computations in AIDA and R do not return contextual information. In order to join the result with *ranking* and to select all A++ conferences, the conference names must be manually added as a new column.



(a) Systems Comparison **(b) RMA+BAT vs RMA+MKL**

Figure 16: Conferences (Covariance Computation)

(4) Trip Count. In Figure 17 we compute the number of trips per rider to 10 different destinations. Each tuple in the input relations stores a rider and the number of trips to each of the 10 locations for one year. We use add on the relations of two different years to get the trip count for a period of two years. We vary the number of riders from 1M to 15M and measure the runtime. Since add is a simple operation, RMA+ uses the no-copy implementation on BATs (RMA+BAT). RMA+ is faster than AIDA and R because it does not transfer data to Python (as AIDA) and does not translate data.tables to matrices (as R). MADlib takes 23, 119, 299, resp. 480 seconds for the different input sizes and, thus, is again omitted from the figure.



(a) Systems Comparison **(b) RMA+BAT vs RMA+MKL**

Figure 17: Trip Count (Matrix Addition)

RMA+BAT vs. RMA+MKL. Following our policy, RMA+ delegates matrix operations to MKL (RMA+MKL) in Figures 14a, 15a, and 16a (the operations are complex and we do not run out of memory), and uses the no-copy implementation on BATs (RMA+BAT) in Figure 17a (add is a linear operation). We compare RMA+BAT to RMA+MKL in all scenarios. RMA+MKL outperforms RMA+BAT for the queries on trips (factor 1.8-3.8, cf. Figure 14b) and journeys (factor 1.4-1.9, cf. Figure 15b). For the conference query, RMA+MKL is 24 to 70 times faster since the cross product requires single element access and operates on relations with a large number of attributes. For the trip count, RMA+BAT outperforms RMA+MKL in all settings (cf. Figure 17b). Although elementwise addition is highly efficient in MKL, the transformation overhead cannot be amortized.

[6]We do not use function cov() since it uses a single core only and is slower.

8.7 Discussion

The key learnings from our empirical evaluation are the following: (1) RMA+ excels for mixed workloads that include both standard relational and matrix operations. (2) Only RMA+ can avoid data transformations in mixed workloads; data transformations may be costly and consume more than 90% of the overall runtime. (3) For complex matrix operations, however, transforming the data to a suitable format may pay off: In our approach, we are free to transform the data whenever beneficial. (4) In terms of scalability to large relations/matrices, our solution outperforms all competitors since it relies on the memory management of the database system for both the standard relational and the matrix operations. (5) Finally, the handling of contextual information, a feature of RMA, is efficient and can leverage optimizations that avoid expensive sortings.

9 CONCLUSION

In this paper, we targeted applications that store data in relations and must deal with queries that mix relational and linear algebra operations. We proposed the *relational matrix algebra* (RMA), an extension of the relational model with matrix operations that maintain important contextual information. RMA operations are defined over relations and can be nested. We implemented RMA over the internal data structures of MonetDB, and the implementation does not require changes in the query processing pipeline. Our integration is competitive with state-of-the-art approaches and excels for mixed workloads.

RMA opens new opportunities for cross algebra optimizations that involve both relational and linear algebra operations. It is also interesting to investigate the handling of wide tables, e.g., by storing them as skinny tables that are accessed accordingly or by combining operations to avoid the generation of wide intermediate tables.

ACKNOWLEDGMENTS

We thank Joseph Vinish D'silva for providing the source code of AIDA and helping out with the system. We thank the MonetDB team for their support. We thank Roland Kwitt for the discussion about application scenarios. The project was partially supported by the Swiss National Science Foundation (SNSF) through project number 407540_167177.

REFERENCES

[1] cuBLAS – NVIDIA BLAS library for GPUs. https://developer.nvidia.com/cublas.

[2] MKL – Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[3] NumPy. http://www.numpy.org, 2018.

[4] C. R. Aberger, A. Lamb, K. Olukotun, and C. Re. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *ICDE. IEEE*, 2018.

[5] C. R. Aberger, S. Tu, K. Olukotun, and C. Re. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446, New York, NY, USA, 2016. ACM.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. *SIGMOD Rec.*, 27(2), June 1998.

[7] M. Boehm, A. Kumar, J. Yang, and H. V. Jagadish. *Data Management in Machine Learning Systems.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[8] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 637–648, New York, NY, USA, 2013. ACM.

[9] O. Dolmatova, N. Augsten, and M. H. Böhlen. Preserving contextual information in relational matrix operations. In *Proceedings of the 36th International Conference on Data Engineering*, ICDE '20. To appear.

[10] O. Dolmatova, N. Augsten, and M. H. Böhlen. A relational matrix algebra and its implementation in a column store (extended version). *CoRR*, abs/2004.05517, 2020.

[11] J. V. D'silva, F. De Moor, and B. Kemme. AIDA: Abstraction for advanced in-database analytics. *Proc. VLDB Endow.*, 11(11):1400–1413, July 2018.

[12] J. V. D'silva, F. De Moor, and B. Kemme. Keep your host language object and also query it: A case for SQL query support in RDBMS for host language objects. In Carlos Maltzahn and Tanu Malik, editors, *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*, pages 133–144. ACM, 2019.

[13] W. Gander. Algorithms for the QR-Decomposition. Technical report, ETH Zurich, April 1980.

[14] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[15] G. H. Golub and C. F. Van Loan. *Matrix Computations.* John Hopkins University Press, 3rd edition, 1996.

[16] J. M. Hellerstein, C. Re, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, August 2012.

[17] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. *CoRR*, abs/1703.07342, 2017.

[18] Kaggle Inc. BIXI Montreal (public bicycle sharing system). https://www.kaggle.com/aubertsigouin/biximtl, 2019.

[19] R.A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis.* Applied Multivariate Statistical Analysis. Pearson Prentice Hall, 2007.

[20] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 523–534, April 2017.

[21] W. McKinney. pandas: a foundational python library for data analysis and statistics, 2011.

[22] D. Misev and P. Baumann. Homogenizing data and metadata retrieval in scientific applications. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*, DOLAP '15, pages 25–34, New York, NY, USA, 2015. ACM.

[23] MonetDB. Online MonetDB reference. https://www.monetdb.org/Home, 2017.

[24] University of Trier. DBLP computer science bibliography. https://dblp.uni-trier.de, 2019.

[25] Oracle. Online Oracle UTL_NLA Package Reference. https://docs.oracle.com/database/121/ARPLS/u_nla.htm, 2016.

[26] C. Ordonez. Building statistical models and scoring with UDFs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1005–1016, New York, NY, USA, 2007. ACM.

[27] R project. The R Project for Statistical Computing. https://www.r-project.org/, 2018.

[28] Quick R. R Matrix Algbera package overview. http://www.statmethods.net/advstats/matrix.html, 2017.

[29] C. Radhakrishna Rao and H. Toutenburg. *Linear Models, Least Squares and Alternatives.* Springer Series in Statistics. Springer-Verlag New York, 1995.

[30] Inc. SciDB. SciDB User's Guide Version 13.3.6203. In *SciDB User's Guide*, pages 1–258, 2013.

[31] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16. Springer-Verlag, 2011.

[32] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. *CoRR*, abs/0909.1766, 2009.

[33] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11. ACM, 2011.

[34] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052. ACM, 2013.