ArrayQL Integration into Code-Generating Database Systems

Maximilian E. Schüle
Technical University of Munich
m.schuele@tum.de

Tobias Götz
Technical University of Munich
tobias.goetz@tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

Array database systems offer a declarative language for array-based access on multidimensional data. Although ArrayQL formulates the operators for a standardised query language, the corresponding syntax is not fully defined nor integrated in a productive system. Furthermore, we see potential in a uniform array query language to fill the gap between linear and relational algebra.

This study explains the integration of ArrayQL inside a relational database system, either addressable through a separate query interface or integrated into SQL as user-defined functions. With a relational database system as the target, we inherit the benefits such as query optimisation and multi-version concurrency control by design. Apart from SQL, having another query language allows processing the data without extraction or transformation out of its relational form. This is possible as we work on a relational array representation, for which we translate each ArrayQL operator into relational algebra. This study provides an extended ArrayQL grammar specification to address each ArrayQL operator. In our evaluation, ArrayQL within Umbra computes matrix operations faster than state of the art database extensions and outperforms traditional array database systems on predicate evaluation and aggregations.

1 INTRODUCTION

Array database systems are developed for geo-temporal data and therefore specialised for multidimensional discrete data (MDD) [3]. Such data occurs within time-series of scientific experiments or real-world scenarios when processing images or indexing geographic or astronomical data [9, 17, 57]. These examples have in common that the tuples can be addressed using a multidimensional index out of coordinates and time [39, 45]. In contrast to relational database systems, array database systems are designed for index-based array access and excel in computing aggregations on numerical data. Popular array database systems are RasDaMan [3], MonetDB SciQL [59] and SciDB [7, 11]. As each one is shipped with its own query language, ArrayQL [30] is an attempt to standardise them as presented at XLDB 2012. Although the corresponding algebra [33] has been published, it is not fully covered by the corresponding draft of a grammar specification [30] needed in order to implement ArrayQL.

Even though array database systems are often based on relational ones, an interface for querying both does not exist. For example, RasDaMan supports relational database systems such as PostgreSQL as an underlying key-value store but archives the data as binary large objects (BLOB) only. SciQL is implemented

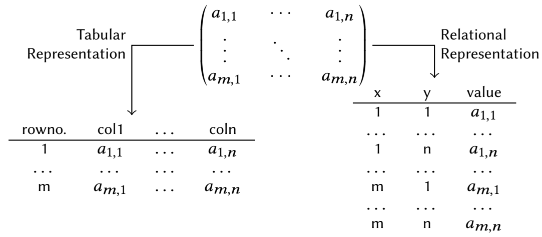

Figure 1: Storing arrays as database tables: either using a tabular representation (left) with the attributes as columns and an additional one defining the row order or using a relational representation (right) with the array as coordinate list.

within MonetDB and stores arrays along with tables in the same memory layout but does not enable cross-querying. However a uniform representation is needed to allow access from SQL *and* an array query language, Arrays have to be either stored as a coordinate list (*relational representation*) or tables have to carry an additional attribute that defines the row order (*tabular representation*, see Figure 1). A relational representation saves memory on sparse arrays as no entry is needed for values equal to zero. As the dimensions and the content are mapped to one attribute each, primitive data types are sufficient even for more than two dimensions. A tabular representation would require a nested array datatype to represent the third dimension.

Another use case for array-oriented data processing arises by the need of matrix operations [52] for data mining [1, 34, 38] and machine learning [22, 29, 31, 54]. The corresponding data is often stored and collected inside relational database systems [2, 6, 13, 48, 48], but its analysis depends on linear algebra, which database systems do not provide. Thus, the data gets extracted into separate tools such as R and Python, so analysis happens on past data, ignoring incoming tuples. We argue that array database systems are ideally suited for machine learning algorithms [41, 47], which essentially depend on data stored in tensors and their transformations [42, 43], making ArrayQL a worthwhile extension.

We claim that relational database systems will highly benefit from ArrayQL as a further query language, either embedded in SQL as user-defined functions or as a separate query interface.

We integrate ArrayQL within our code-generating database system Umbra [23, 36]. We decided in favour of a relational array representation allowing a direct mapping onto relational algebra at compile time. This requires an extension of the semantic analysis only, rather than a change to the underlying query engine. The extension accepts ArrayQL statements as part of SQL either as user-defined functions or via a separate interface. As an advantage, ArrayQL can work on SQL tables, and SQL has access to ArrayQL arrays. The extension does neither affect runtime nor the compile time of SQL queries. This study extends preliminary work about ArrayQL for linear algebra within Umbra [49] by an ArrayQL grammar specification, its application on real-world data and a comprehensive evaluation of array database systems. This study's specific contributions are:

- an extended grammar definition that supports the full ArrayQL algebra,
- a relational array representation including bounding boxes and validity maps for ArrayQL within database systems
- the translation of corresponding operators into relational algebra,
- the integration of ArrayQL into a code-generating database system with Umbra as target and
- an experimental evaluation using micro-benchmarks for linear algebra and real-world data for array operations.

This study comprises the following sections: Section 2 summarises existing work about array database systems and data analysis tools for relational algebra. Section 3 provides a complete ArrayQL grammar to address the ArrayQL operators. Section 4 presents the architecture when integrating ArrayQL within the beyond main-memory database system Umbra as the target. Section 5 introduces the ArrayQL algebra and its translation into relational algebra. Section 6 demonstrates the application of ArrayQL in conjunction with SQL and for linear algebra. Section 7 evaluates the proposed extension using micro-benchmarks for basic matrix algebra, queries on real-world data and the SS-DB benchmark.

2 RELATED WORK

Beside the array database systems RasDaMan, MonetDB SciQL and SciDB, this study considers also efforts towards integration of matrix algebra [21] into database systems.

2.1 Array Database Systems

Introduced in 1997, RasDaMan [3] was the first array database system developed for geo-spatial data. Even though it supports relational database systems as underlying storage engines, it stores data as BLOBs similar to a file system. Therefore, querying happens within RasDaMan only, for which it offers the array query language RasQL, including SQL-92 and embedded statements for multi-dimensional data. These statements access arrays and have been incorporated in the SQL/MDA:2019 standard[1] for multi-dimensional fields.

SciDB [7] is a database system that uses arrays as a first-class data model. Its declarative query language AQL is based on SQL and the array programming language APL. Another array database system is SciQL [59], which uses MonetDB's query engine and storage layout. Binary association tables (BAT), normally used to store columns, hold the array data. This allows one unified query interface to address either SQL tables or arrays.

2.2 Machine Learning Tools

Matrices are an example of multi-dimensional data. Its algebra attracts attention by the use for machine learning [12]. For this purpose, MATLANG [5], Lara [27] or BUDS [14] offer declarative query languages to produce optimised operator plans. Another machine learning tool is SystemML [4] that supports linear algebra and optimises query plans using cost models. We work on a sparse data model, for which sparsity estimation optimises the query plan [51]. Other work relies on such query plans when computing partial derivations [50].

2.3 Extensions for Database Systems

MADlib [18] operates on tables in a relational representation, which they call a sparse matrix, as well as on the array datatype. Also implemented as a datatype inside database systems, Kernert [25] enable native support for linear algebra on dense and sparse matrices. However, enabling a separate datatype has the downside of expensive transformations out of tables.

Luo et al. [32] argue that database systems form an excellent platform for linear algebra as this kind of computations can be expressed as a combination of operators within relational algebra. Due to the complexity of writing linear algebra computations in SQL and the overhead of the Volcano-style iterator model [15], they propose adding a vector and matrix datatype as database attributes and a small set of SQL language extensions for corresponding operations. Umbra eliminates the overhead of one function call per operator introduced by the Volcano-style iterator model as it generates low-level virtual machine (LLVM) code according to the producer-consumer model [35, 44, 46]. This allows pipelined processing, reduces the cost per tuple significantly and achieves nearly the performance of a hard-coded implementation. This study benchmarks the performance increase for linear algebra within such a code-generating database system in comparison to traditional (array) database systems. Although Umbra provides a datatype to store matrices as a part of relational tables, this study's motivation is to provide an array view on tables.

To allow linear algebra directly on database tables, relational matrix algebra (RMA) [10] extends MonetDB by operators for linear algebra. The linear operations can be addressed in SQL as table functions. But in contrast to our study, RMA interprets tables as matrices (tabular representation), limited to two-dimensional matrices, and requires a row-ordering as contextual information among linear operations. In contrast, SPORES [53] uses a relational representation for matrices only as an intermediate format to derive optimisations from relational algebra to SystemML. When we base our matrices directly on tables and translate all operations into relational algebra, we earn these optimisations [16, 19, 26, 55, 58] for free.

3 ARRAYQL GRAMMAR

The syntax draft of 2012 [30] proposed ArrayQL as a data definition and data query language on arrays as the principle data model. Common elements with SQL are the keywords and the syntax to create and access attributes, but in contrast, the statements are extended to specify dimensions. Beside arithmetic operations and basic array transformations, ArrayQL allows aggregations and joins. We add support for temporal tables, extend the join functionality and propose the syntax for an extension to a data modification language. In this section, we introduce the ArrayQL statements and give a syntax definition in Backus-Naur-Form (see Figure 2).

3.1 Data Definition Language

As a data definition language, ArrayQL allows the creation of arrays similar to SQL tables (see Listing 1). A create statement starts with the keyword `CREATE ARRAY` followed by the array name. As arguments inside parentheses, ArrayQL expects the keyword `DIMENSION` together with the array bounds and, as in SQL, the attributes per cell.

[1] https://www.iso.org/standard/69777.html

```
CREATE ARRAY m (i INTEGER DIMENSION [1:2],
 j INTEGER DIMENSION [1:2], v INTEGER);
```

Listing 1: Array creation statement.

As an alternative, creation is possible out of an existing array (see Listing 2).

```
CREATE ARRAY n FROM SELECT [i], [j], v FROM m;
```

Listing 2: Array creation out of existing array.

We suggest ArrayQL in conjunction with SQL insert statements to allow mixed queries. In conjunction with SQL, the ArrayQL create statement allows to create new tables or prepare existing ones for array-based processing (initialising the bounds and adding an index on the dimension attributes). When a new array has been created, SQL can access the corresponding table to insert elements like bulk-loading from CSV. Afterwards, ArrayQL as a data query language can process the filled array.

3.2 Data Query Language

ArrayQL is intended as a data query language to access and aggregate along the array's dimensions. Similar to SQL, an ArrayQL statement is composed out of a `SELECT`- and a `FROM`-clause and, optionally, one for `WHERE` and one for `GROUP BY` (see Listing 3). The `SELECT`-clause expects the indices for the dimensions (in brackets) as well as arithmetic expressions as attributes that form the result. The `FROM`-clause specifies the source array. As its arguments, arrays or compound statements, such as joins, table-functions or entire subqueries are allowed. The `WHERE`-clause filters each entry by a predicate. The `GROUP BY`-clause addresses the dimensions preserved after an aggregation.

```
SELECT [i], SUM(v)+1 FROM m WHERE v>0 GROUP BY i
```

Listing 3: Exemplary ArrayQL select statement.

Optionally, the indices can be rearranged, which is indicated by the dimension names inside brackets behind the source array. The keyword `AS` allows renaming of expressions as well as of input arrays. Filtering and grouping happen similar to SQL: filter expects a condition inside a `WHERE`-clause, grouping expects the dimensions considered for aggregation as part of the `GROUP BY`-clause.

In addition to the ArrayQL draft, we propose support of temporary arrays, explicit inner joins and various table functions. Temporary arrays are introduced by the keyword `WITH` (see Listing 4), similar to temporary tables in SQL.

```
WITH ARRAY temp AS (SELECT [i] as j, SUM(v+1) FROM m[j] WHERE v
     >0 GROUP BY j) SELECT * FROM temp;
```

Listing 4: Temporary table in ArrayQL.

Inner joins are part of the ArrayQL algebra in Section 5. Table functions are needed to apply linear algebra on arrays, and therefore, discussed in Section 6.

3.3 Data Modification Language

Beside manipulating existing arrays using SQL, we add basic support for ArrayQL update statements (see Listing 5). An ArrayQL update statement is introduced by `UPDATE ARRAY` and expects the array name and the dimensions whose values should be modified. An ArrayQL select statement or an explicit `VALUES`-clause containing the attributes specify the new values. Inside a `VALUES`-clause all attributes of one cell are enclosed by brackets.

```
⟨ArrayQlStmt⟩ ::= ⟨SelectStmt⟩
               |  ⟨CreateStmt⟩
               |  ⟨UpdateStmt⟩
⟨SelectStmt⟩ ::= ⟨WithArray⟩? 'SELECT' ⟨ExprList⟩ 'FROM' ⟨SubarrayList⟩
                 ⟨FilterExpr⟩? ⟨GroupByExpr⟩?
⟨CreateStmt⟩ ::= 'CREATE' 'ARRAY' ⟨Name⟩ ⟨CreateStyle⟩
⟨CreateStyle⟩ ::= 'FROM' ⟨SelectStmt⟩
               |  '(' ⟨ArrayDef⟩? ')'
⟨ArrayDef⟩ ::= ⟨AttrDef⟩ ( ',' ⟨AttrDef⟩ )*
            |  ⟨DimDef⟩ ( ',' ⟨DimDef⟩ )* ( ',' ⟨AttrDef⟩)*
⟨DimDef⟩ ::= ⟨Name⟩ ⟨Type⟩ 'DIMENSION' '[' ⟨Min⟩ ':' ⟨Max⟩ ']'
⟨AttrDef⟩ ::= ⟨Name⟩ ⟨Type⟩
⟨ExprList⟩ ::= ⟨SingleExpr⟩ (',' ⟨SingleExpr⟩)*
⟨SingleExpr⟩ ::= ⟨Expr⟩ ( 'AS' ⟨Name⟩ )?
              |  '[' ⟨Name⟩ ']' ( 'AS' ⟨Name⟩ )?
              |  '[' ⟨Min⟩ ':' ⟨Max⟩ ']' 'AS' ⟨Name⟩
              |  ⟨AggregationFunction⟩ '(' ⟨Name⟩ ')'
⟨SubarrayList⟩ ::= ⟨JoinExpr⟩ (',' ⟨JoinExpr⟩)*
⟨JoinExpr⟩ ::= ⟨JoinArray⟩ ( 'AS' ⟨Name⟩ )?
            |  ⟨SingleSubarray⟩
⟨JoinArray⟩ ::= ⟨SingleSubarray⟩ 'JOIN' ( ⟨SingleSubarray⟩ | ⟨JoinArray⟩ )
⟨SingleSubarray⟩ ::= ⟨Name⟩ ⟨Alias⟩?
                  |  ⟨Name⟩ '[' ⟨Expr⟩ (',' ⟨Expr⟩)* ']' ⟨Alias⟩?
                  |  ⟨Name⟩ '*' ⟨Name⟩
                  |  ⟨FunctionName⟩ '(' ')'
⟨WithArray⟩ ::= 'WITH' 'ARRAY' ⟨Name⟩ 'AS' '(' ⟨CreateStyle⟩ ')' ( ',' 'ARRAY'
                ⟨Name⟩ 'AS' '(' ⟨CreateStyle⟩ ')' )*
⟨FilterExpr⟩ ::= 'WHERE' ⟨BooleanExpr⟩
⟨GroupByExpr⟩ ::= 'GROUP' 'BY' ⟨Name⟩ ( ',' ⟨Name⟩ )*
⟨UpdateStmt⟩ ::= 'UPDATE' ⟨Name⟩ ⟨UpDim⟩* '(' ⟨UpExpr⟩ ')'
⟨UpDim⟩ ::= '[' ⟨Expr⟩ ']'
         |  '[' ⟨Min⟩ ':' ⟨Max⟩ ']'
⟨UpExpr⟩ ::= ⟨SelectStmt⟩
          |  'VALUES' '(' ⟨Expr⟩ ( ',' ⟨Expr⟩ )* ')' ( ',' '(' ⟨Expr⟩ ( ',' ⟨Expr⟩ )* ')' )*
```

Figure 2: ArrayQL grammar in EBNF.

```
UPDATE ARRAY m [1][1] (select [2][1],v FROM m);
UPDATE ARRAY m [1][1:2] (values (1),(2));
```

Listing 5: ArrayQL update statements.

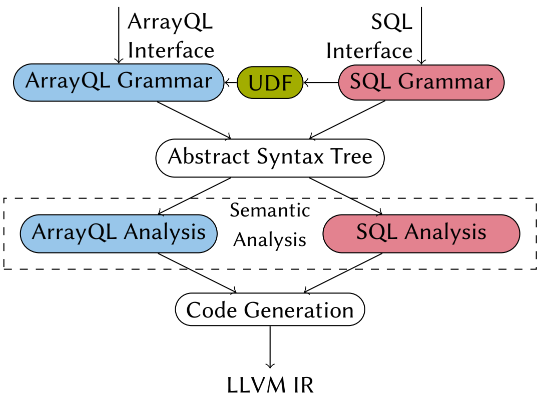

Figure 3: ArrayQL extension in Umbra: ArrayQL statements are parsed via a separate interface or as part of user-defined functions (UDF) in SQL; a separate file contains the ArrayQL grammar. An abstract syntax tree is generated, whereof the ArrayQL statements are analysed separately.

4 IN-DATABASE INTEGRATION

This section illustrates the changes made to the database system Umbra either in order to accept ArrayQL as user-defined functions or to offer a separate query interface. Afterwards, we explain design decisions made to suit the relational concept of a code-generating database system and to enable cross-querying without overhead.

4.1 Architecture

Umbra is a code-generating database system following the producer-consumer concept. First, it transforms the parsed SQL query into an abstract syntax tree, which is passed to the semantic analysis to create the operator plan. The operator plan gets optimised using estimated cardinalities as only the schema is known during compile-time. Afterwards, a translator class for each operator generates the LLVM code, for which the traditional tuple-flow is inversed. So instead of pushing tuples upwards a target operator, operators demand their sources to produce code. Each source then demands the parent operator, the consumer, to generate code for processing each tuple further.

Umbra supports user-defined functions, that are handled separately during semantic analysis. For each language, a separate grammar file together with one for its semantic analysis exists. This is shown in Figure 3: when adding ArrayQL to Umbra, either a separate interface accepts ArrayQL statements or they are parsed as part of a user-defined function in SQL. Afterwards, a common abstract syntax tree is generated, which is then analysed separately. Within the semantic analysis, we mostly rely on standard relational algebra operators, but we are also able to call customised operators. Because of this, we benefit from query optimisation such as operator reordering or predicate push-down.

4.2 Array Representation

Only the schema is known during compile-time, whereas the tuples can only be accessed during run-time. This interferes with a tabular array representation, as only the columns are part of the schema, and leads us to the relational representation. We store every n-dimensional array with m values per cell as a table with $n + m$ attributes. Stored as a coordinate list, the attributes for the indices are unique and form the primary key. This allows their indexing and fast retrieval later on.

ArrayQL differentiates between attributes and dimensions, which becomes obsolete in a relational representation as dimensions are mapped to attributes internally. This leads to more flexibility, since arbitrary attributes can be used as dimensions.

According to the ArrayQL algebra, an array consists of a *bounding box*, a *validity map* and the *content*. The bounding box defines the bounds for each dimension, whereas the validity map defines the visible cells within the bounds and the attributes per cell define the content. To define the bounding box, we simply insert a tuple for the lower as well as the upper bound upon array creation (see Figure 4). Within the bounding box, we consider an entry as valid if it exists and at least one attribute is not declared as NULL.

```
CREATE ARRAY m (
 i INTEGER DIMENSION [1:2],
 j INTEGER DIMENSION [3:4],
 v INTEGER);
```

$\rightarrow$

x	y	v
1	3	*NULL*
2	4	*NULL*

Figure 4: When an array is created with ArrayQL and the dimensions are specified, a corresponding relation is created with two initial tuples defining the bounding box.

4.3 Interaction with SQL

Depending on its signature, ArrayQL expressions, when used as part of a user-defined function, return either a table, e.g., `TABLE (x INT, y INT, v INT)`, or a single array attribute, e.g., `INT[][]` (see Listing 6). As a table function, it returns the relational array representation, that can be further processed in SQL. Otherwise, when the function is declared to return a single attribute, the result is cast to Umbra's array datatype.

```
CREATE FUNCTION exampletable() RETURNS TABLE (x INT, y INT, v
      INT) LANGUAGE 'arrayql' AS 'SELECT␣[x],␣[y],␣v␣FROM␣m';
CREATE FUNCTION exampleattribute() RETURNS INT[][] LANGUAGE '
      arrayql' AS 'SELECT␣[x],␣[y],␣v␣FROM␣m';
```

Listing 6: ArrayQL as part of a user-defined function returns either an SQL table or an SQL array.

5 ARRAYQL ALGEBRA

ArrayQL offers an algebra [33] that is similar to relational algebra and allows a mapping to SQL operators considering the underlying schema. The algebra offers nine operators (see Table 1), for which it defines content, validity maps and bounding box. In our relational form, one relation $a \subseteq \mathbb{I}^n \times \mathbb{R}^m$ with schema $sch(a) = \{\underline{i_1, \ldots, i_n}, r_1, \ldots, r_m\}$ represents one n-dimensional array $\mathfrak{a} \in (\mathbb{R}^m)^{|i_1| \times \ldots \times |i_n|}$ with m attributes of domain $\mathbb{R}$ as content. Its coordinates $(i_1, \ldots, i_n) \subseteq \mathbb{I}^n$ form the primary key and delimit the bounding box. We formulate the validity map of an array $\mathfrak{a}$ as set of indices $d_a \subseteq \mathbb{I}^n$ of valid entries. Transferred to SQL, all entries are valid, for which a tuple exists with not-null attributes. This section introduces the ArrayQL operators, the corresponding syntax and the translation into SQL operators.

5.1 Rename

The rename operator assigns a new name to either a dimension, attribute or a whole array. Similar to the rename operator ρ in SQL, it is introduced by a keyword (AS) behind expressions or tables.

```
SELECT [i] AS s, [j] AS t, v AS c FROM m[s,t];
```

Listing 7: Rename operator.

5.2 Function Application

The apply operator applies a function $f \in \mathbb{R}^m \rightarrow \mathbb{R}^o$ on certain attributes of each valid entry. This is translated to an arithmetic expression as part of an SQL projection $\pi_{i_1,\ldots,i_n,f(r_1,\ldots,r_m)}(a)$. As function application does not affect the validity map, no further adjustments are needed.

```
SELECT [i], [j], v+2 FROM m;
```

Listing 8: Function application: addition.

5.3 Filter

The filter operator invalidates cells for which a condition does not hold. This is called implicitly when accessing an array via indices or explicitly when checking the cell's value as part of the WHERE-clause. Both ways are translated into selections of relational algebra $\sigma_{p(v)}(a)$, as both dimensions and attributes are represented in SQL as attributes.

```
SELECT [i], [j], v FROM m WHERE v = 0.0;
SELECT [i] as i, [j] as j, * FROM m[i/2, j];
```

Listing 9: Explicit and implicit filter operator.

Operator	Input	Output	Validity Map	Relational Algebra
apply	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}, f \in (\mathbb{R} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$\pi_{i_1,\ldots i_n, f(v)}(a)$
combine	$\mathfrak{a}, \mathfrak{b} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a \uplus d_b = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$a ⟗_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n}(b)$
i. dim. join	$\mathfrak{a}, \mathfrak{b} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$(\mathbb{R}, \mathbb{R})^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a \cap d_b = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$a \bowtie_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n}(b)$
fill	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a \subseteq d_{out} = \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$\ldots 0_{\lvert a.i_1 \rvert, \ldots, \lvert a.i_n \rvert} \cdots$
filter	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}, p \in (\mathbb{R} \to \mathbb{B})$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_{out} \subseteq d_a \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$\sigma_{p(v)}(a)$
rebox	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}, i_1^l, i_1^u, \ldots, i_n^l, i_n^u \in \mathbb{I}$	$\mathbb{R}^{i_1^u - i_1^l \times \cdots \times i_n^u - i_n^l}$	$d_a \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert, d_{out} \subseteq i_1^u - i_1^l \times \cdots \times i_n^u - i_n^l$	$\sigma_{i_1^l \leq i_1 \leq i_1^u \wedge \cdots \wedge i_n^l \leq i_n \leq i_n^u}(a)$
reduce	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}, f \in (\mathbb{R}^{\lvert i_n \rvert} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_{n-1} \rvert}$	$d_a \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert, d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_{n-1} \rvert$	$\gamma_{i_1,\ldots,i_n, f(v)}(a)$
rename	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$\rho(a)$
shift	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}, i'_1, \ldots, i'_n \in \mathbb{I}$	$\mathbb{R}^{\lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert}$	$d_a = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$	$\pi_{i_1+i'_m,\ldots,i_n+i'_n,v}(a)$

Table 1: Operators of the ArrayQL algebra: the first column names the operator, the second column specifies the input arguments, the third column the output array, the fourth column defines the set of valid indices and the latter one the translation of ArrayQL operators into relational algebra. $i_{1 \ldots n}$ represents the attribute for the dimension in relational form, $\lvert i_{1 \ldots n} \rvert$ denotes the size of a dimension. We assume arrays having a single attribute $v \in \mathbb{R}$ only.

5.4 Index Manipulation: Shift and Rebox

Shift moves the indices, whereas rebox redefines the bounding boxes by enlarging or shrinking the array size. In our relational schema, shift is translated into an arithmetic expression as part of a projection, as it modifies each index by adding or subtracting the difference $i'_1, \ldots, i'_n \in \mathbb{I}$:

$$\pi_{i_1+i'_1,\ldots,i_n+i'_n,r_1,\ldots,r_m}(a).$$

```
SELECT [i] as i, [j] as j, b FROM m[i+1,j-1];
```

Listing 10: Shift operator.

For rebox, if the array size is shrunk, a conditional statement (selection) filters out each index, which is outside the new bounding box given as lower and upper bounds $i_1^l, i_1^u, \ldots, i_n^l, i_n^u \in \mathbb{I}$:

$$\sigma_{i_1^l \leq i_1 \leq i_1^u \wedge \cdots \wedge i_n^l \leq i_n \leq i_n^u}(a).$$

In any case, new array bounds have to be added afterwards (with a union operator).

```
SELECT [1:5] as i, [1:5] as j, * FROM m[i,j];
```

Listing 11: Rebox operator.

5.5 Fill

The fill operator creates an entry with the default value (0 for numerics) for the attributes of every invalid cell within the bounding box. This is useful for linear algebra with arrays as input matrices and has to be called by a keyword. Internally, it is translated to a call to `generate_series`, an outer join and a projection, only when enabled by the keyword `filled` in ArrayQL and needed before applying specific operations (see Section 6.2):

$$\pi_{COALESCE(a.r_1,0),\ldots}(a ⟗_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n}(\rho_b(0_{\lvert a.i_1 \rvert,\ldots,\lvert a.i_n \rvert}))).$$

```
SELECT FILLED [i], [j], * FROM m;
```

Listing 12: The keyword `FILLED` enables the fill operator.

5.6 Combining and Joining

ArrayQL defines three operators for joining arrays, namely combine, the inner dimension join and—its generalisation to attributes—the inner extended join.

5.6.1 Combine. Combine merges two arrays of the same dimensionality but distinct valid cells, so it concatenates arrays. All cells are valid that are at least valid in one input: $d_a \uplus d_b = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$. `NULL` is assumed for the attributes of a missing join partner. Combine acts like a full outer join, to which it is translated in relational algebra:

$$a ⟗_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n}(b).$$

```
CREATE ARRAY m2(x INTEGER DIMENSION [3:4], y INTEGER DIMENSION
      [1:2], v2 INTEGER);
SELECT [i] as i, [j] as j, v, v2 FROM m[i,j], m2[i,j];
```

Listing 13: Combine operator.

5.6.2 Inner Join. The inner dimension/extended join corresponds to the inner join:

$$a \bowtie_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n}(b).$$

All cells are valid, that are valid in both join partners: $d_a \cap d_b = d_{out} \subseteq \lvert i_1 \rvert \times \cdots \times \lvert i_n \rvert$. They differ, as the inner dimension join only allows dimensions as indices, whereas the inner extended join generalises the join predicate, so that attributes can be used to determine the index as well. As the usage of either combine or join is data-dependent and not known during compile-time, we add the keyword `JOIN` to explicitly perform an inner join. This differs from the original ArrayQL proposal where it shares the syntax with combine (which is called when an inner join cannot be applied).

```
SELECT [i] as i, [j] as j, v, v2 FROM m[i+2,j+2] JOIN m2[i-2,j
      -2];
```

Listing 14: Inner dimension Join.

5.7 Reduce for Aggregations

Reduce performs an aggregation over at least one dimension as needed by roll-up queries of analytical workloads. Reduce is introduced by the keywords *GROUP BY*, as known from SQL, followed by the preserved dimensions after reduction. Similarly, one aggregation function $f \in ((\mathbb{R}^m)^{\lvert i_n \rvert} \to \mathbb{R}^m)$ must be applied to all remaining attributes. These similarities allow a direct mapping to aggregations in relational algebra:

$$\gamma_{i_1,\ldots,i_n,f(v)}(a).$$

```
SELECT [i], sum(v) FROM m GROUP BY i;
```

Listing 15: Reduce operator for aggregation: summation

6 APPLICATION OF ARRAYQL

Array query languages have two major application areas: the common one is for use with geo-temporal data, the second one is applying linear algebra on data in relational form for statistical analysis. One important difference between the two domains concerns the handling of invalid values. Array database systems assume `NULL` for non-existing values, whereas these are interpreted as `0` within sparse matrices. To conform to the ArrayQL specification, we assume the geo-temporal use-case as default. To distinguish the other one, the keyword `filled` indicates matrix operations. This section explains both use-cases including table-function extensions.

6.1 Geo-Temporal Data

The intended purpose of ArrayQL is to allow index-based access to geo-temporal data. With our ArrayQL integration into a relational database system, mixed query types become possible. A table created in SQL (see Listing 16) can be accessed within ArrayQL (see Listing 17) and vice versa. ArrayQL interprets the attributes that form the table's primary key as indices. Accordingly, SQL has access to all array's dimensions as attributes.

```
CREATE TABLE taxidata(id TEXT, pickup_longitude INT,
      pickup_latitude INT, pickup_datetime DATE, dropoff_datetime
       DATE, trip_duration FLOAT, PRIMARY KEY(id,
      pickup_longitude, pickup_latitude));
INSERT INTO mytaxidata [..];
```

Listing 16: Table creation and data insertion using SQL.

```
SELECT [pickup_longitude],[pickup_latitude], SUM(trip_duration)
FROM mytaxidata GROUP BY pickup_longitude, pickup_latitude;
```

Listing 17: ArrayQL queries on an SQL table: the attributes that form the primary key serve as indices.

6.2 Linear Algebra with ArrayQL

The ArrayQL operators allow expressing basic mathematical expressions. Complex functions are realised by dedicated operators or user-defined functions. They are called as part of the `from`-clause where arbitrary table functions are accepted as input.

When creating an array, the bounding box defines the size of a matrix, the attributes determine the field of which each entry is a member. Operations on matrices obey the following pattern: scalar operations are part of the `select`-clause, matrix operations belong to the `from`-clause (see Figure 5).

$$(a_{ij}) = A \in \mathbb{R}^{2\times 3}$$

```
CREATE ARRAY A (
i INTEGER DIMENSION [1:2],
j INTEGER DIMENSION [1:3],
a FLOAT)
```

$$(a_{ij}) = A$$

```
SELECT [i],[j], A.a FROM A
```

Figure 5: Correlation of ArrayQL with matrices.

When performing linear algebra on arrays, the main difference to geo-temporal applications concerns the meaning of invalid entries. As arrays are interpreted as sparse matrices, values of non-existing entries are assumed to be zero. The fill operator has to be put implicitly in front of respective operations to consider operations that alter zero values. To enable this feature, we propose the keyword `filled` behind `select`. Having enabled this feature, the fill operator presented in Section 5.5 is called when necessary: this includes all arithmetic unary and binary functions but not joins. This functionality is enabled for all trigonometric, arithmetic and aggregate functions (see Listing 18). During semantic analysis, the fill operator is added to the operator tree before the function call is generated. The operator creates an array of the same dimensions containing zeros proceeded by an outer join with the original table on the indices. A `COALESCE` statement then replaces non-existing (null) values, thus within the array-bounds only.

Function	ArrayQL operator
addition	apply
scalar multiplication	apply
matrix multiplication	i.d.join, reduce
slice	rebox
subtraction	apply
transpose	rename

Table 2: Matrix algebra with ArrayQL.

```
SELECT FILLED [i], [j], v+2 FROM m;
SELECT FILLED [i],  max(v) FROM m GROUP BY i;
```

Listing 18: The fill operator is called before a function call, for example, of an arithmetic binary function or a unary aggregate function like a row-wise maximum function.

The ArrayQL algebra allows expressing basic mathematical expressions as scalar operations or compound statements (see Table 2). For operations not covered by the algebra, we add additional table functions. We demonstrate matrix operations expressed in ArrayQL, short-cuts to simplify their usage and their application for linear regression and training a neural network.

6.2.1 Scalar Operations. Scalar operations are part of the `select`-clause as arithmetic expressions (see Listing 19). They correspond to the apply operator, since scalar multiplication, addition or subtraction are each applied elementwise.

```
SELECT [i],[j],m.v*n.v FROM m,n;  -- multiplication
SELECT [i],[j],m.v+n.v FROM m,n;  -- addition
SELECT [i],[j],m.v-n.v FROM m,n;  -- subtraction
```

Listing 19: Scalar operations in ArrayQL.

6.2.2 Transpose and Slice. To transpose or slice an array, we can rely on basic index manipulations. Slicing an array corresponds to the rebox operator (see Listing 11), as it defines a sub-range for each index. For transposition $A^T = (a_{ij})^T = (a_{ji})$, ArrayQL does not perform an operation but renames the indices (see Listing 20) as the matrices are stored in a relational representation as a coordinate list.

```
SELECT [j] AS s, [i] AS t, * FROM m[s, t]
```

Listing 20: Transpose.

6.2.3 Matrix Multiplication. The text-book implementation of a matrix multiplication $(a_{i,k}) \cdot (b_{k,j}) = \sum_{k=1}^{n} a_{ik} b_{kj}$ can be expressed in ArrayQL. In relational algebra, with two tables $m\{[i,k,v]\}, n\{[k,j,v]\}$ each representing a matrix, the multiplication corresponds to an operator tree out of join, apply (for elementwise multiplication) and reduce (for final summation):

$$\gamma_{m.i,n.j,sum(m.v \cdot n.v)}(\pi_{m.v \cdot n.v}(m \bowtie_{m.k=n.k} n)).$$

ArrayQL allows to join over the dimension k implicitly (see Listing 21). Nevertheless, this query highly resembles its SQL counterpart (see Listing 22). In addition, it is not practical as the product and the summation have to be stated manually.

```
SELECT [i], [j], SUM(product) AS a FROM (
  SELECT [*:*] AS i, [*:*] AS j, [*:*] AS k, a.v * b.v AS
       product
  FROM m[i,k] a JOIN n[k,j] b) as ab GROUP BY i,j;
```

Listing 21: Text-book matrix multiplication in ArrayQL.

```
SELECT m.j AS i,n.j,SUM(m.v*n.v)
FROM a AS m INNER JOIN a AS n ON m.i=n.i
GROUP BY m.j, n.j;
```

Listing 22: Corresponding matrix multiplication in SQL.

6.2.4 Implemented Table Functions and Short-Cuts. To express linear algebra with ArrayQL, we introduce abbreviations for matrix operations. Matrix operations, either expressible in ArrayQL (like matrix multiplication or addition), or requiring a table function (such as for inversion) should belong to the `from`-clause. We implemented short-cuts to offer similar notations that resemble mathematical expressions and avoid writing prefix function calls (like `f()`). These short-cuts exist for operations not covered so far by the ArrayQL algebra as well as for compound operations to simplify their application (see Listing 23). Furthermore, this allows the `from`-clause to comprise larger statements out of matrices later needed for linear regression.

```
SELECT [i],[j],* FROM m+n;  -- addition
SELECT [i],[j],* FROM m^-1; -- inversion (table function call)
SELECT [i],[j],* FROM m*n;  -- multiplication
SELECT [i],[j],* FROM m^2;  -- power
SELECT [i],[j],* FROM m-n;  -- subtraction
SELECT [i],[j],* FROM m^T;  -- transposition
```

Listing 23: Short-cuts in ArrayQL.

6.2.5 Application. The presented operations allow solving numerical tasks based on matrix manipulations like solving linear regression numerically or training a neural network.

Linear regression can be expressed out of an input array $X \in \mathbb{R}^{i \times j}$, containing i tuples with j attributes, and a weight vector $\vec{w} \in \mathbb{R}^j$ to predict the labels $\vec{y} \in \mathbb{R}^i$ as follows: $X \cdot \vec{w} = \vec{y}$. Given a training dataset with corresponding labels $\vec{y}$, a closed-form expression out of transposition, matrix multiplication and inversion computes the optimal weight matrix:

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

As multiplication and transposition are expressible in ArrayQL, the corresponding short-cuts together with one for matrix inversion allow ArrayQL to process the closed-form expression. The query computes the weight matrix (see Listing 25) without writing nested subqueries as in SQL (see Listing 24).

```
SELECT tmp.i, SUM(tmp.sum*y.val) FROM (
 SELECT inv.i, m.i as j, sum(m.val*inv.sum)
 FROM matrixinversion(TABLE (
  SELECT a1.j AS i,a2.j,SUM(a1.val*a2.val)
  FROM m AS a1 INNER JOIN m AS a2 ON a1.i=a2.i
  GROUP BY a1.j, a2.j)
 ) AS inv INNER JOIN x ON inv.j=a.j GROUP BY inv.i, m.i
) AS tmp INNER JOIN y ON tmp.j=y.i GROUP BY tmp.i;
```

Listing 24: Linear regression in SQL.

```
SELECT [i],[j],* FROM ((m^T * m)^-1*m^T)*y
```

Listing 25: Linear regression in ArrayQL.

Supporting transposition and multiplication on arrays, ArrayQL is capable of expressing the forward pass of a fully connected neural network. A fully connected neural network with one hidden layer of size h requires two weight matrices $w_{hx} \in \mathbb{R}^{h \times |\vec{x}|}$ and $w_{oh} \in \mathbb{R}^{|L| \times h}$ and expects a feature vector as input. For the forward pass, matrix multiplications together with an activation function form a model function $m_{w_{hx},w_{oh}}(\vec{x}) \in \mathbb{R}^{|L|}$ that produces an output vector of probabilities:

$$sig(x) = (1 + e^{-x})^{-1},$$

$$m_{w_{hx},w_{oh}}(\vec{x}) = \underbrace{sig(w_{oh} \cdot \overbrace{sig(w_{hx} \cdot \vec{x})}^{a_{hx}})}_{a_{oh}}.$$

For preparation, we create the necessary tables for weights as well as the input matrix in SQL and define the sigmoid as an SQL function (see Listing 26). Afterwards, the forward pass is computed using an ArrayQL statement (see Listing 27).

```
CREATE TABLE input(i INT PRIMARY KEY, v FLOAT);
CREATE TABLE w_hx(i INT, j INT, v FLOAT, PRIMARY KEY (i,j));
CREATE TABLE w_oh(i INT, j INT, v FLOAT, PRIMARY KEY (i,j));
INSERT INTO ...
-- helper function
CREATE FUNCTION sig(i FLOAT) RETURNS FLOAT AS
 $$ SELECT 1.0/(1.0+exp(-i));$$ LANGUAGE 'sql';
```

Listing 26: Preparation for the neural network in SQL-92.

```
SELECT [i],[j], sig(v) as v FROM w_oh * (
 SELECT [i],[j], sig(v) as v FROM w_hx * input);
```

Listing 27: Forward pass in ArrayQL.

6.3 Logical Optimisations

Implemented within a relational database system, ArrayQL benefits from query optimisation by design. The operators undergo logical optimisations, inherited from the nature of the relational operators, including cost-based query plan reordering based on heuristics on the table sizes. This does not include mathematical optimisations that make use of, e.g., the symmetry property of matrices. We discuss the logical optimisations theoretically and show query plan reordering using multiplication of three matrices as an example.

6.3.1 Optimisation Steps. Database systems optimise queries logically by breaking up conjunctive predicates and pushing them down together with projections and changing the join order. We outline these optimisation steps with regard to ArrayQL operators.

- *Conjunctive predicate break-up and predicate push-down:* affects the filter and the rebox operator, as both are translated into selections. Filtering is similar to a selection. The rebox operator allows us to ignore all tuples outside the specified range.
- *Projection push-down*: concerns the apply and shift operator, that are both translated into projections. This is only beneficial for unbound attributes or when the query optimiser covers mathematical transformations.
- *Join ordering:* applicable to the combine operator and the inner dimension/extended join that are translated into joins.
- *Other:* Beside projections, also aggregations should consider mathematical transformations and be pushed down if possible, as required by reduce. The rename operator is only relevant for the data flow.

6.3.2 Cost-Based Query Plan Reordering. We demonstrate cost-based query plan reordering by an example of a three-way matrix multiplication. Given three matrices $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times o}, C \in \mathbb{R}^{o\times p}$, associativity allows computing their product $ABC \in \mathbb{R}^{m\times p}$ either as $(AB)C$ with $AB \in \mathbb{R}^{m\times o}$ or as $A(BC)$ with $BC \in \mathbb{R}^{n\times p}$. This results in two different operator plans (see Figure 6) with the two joins as common elements but with a different order concerning the aggregations. Although logical optimisation might push down the matrix subproduct out of projection and aggregation above the first join, the query optimiser must be aware of distributive properties. This allows the optimiser to transform the statement $\sum_{j'=1}^{j} a_{ij'} \sum_{k'=1}^{k} b_{j'k'} c_{k'l}$ over $\sum_{j'=1}^{j} \sum_{k'=1}^{k} a_{ij'} b_{j'k'} c_{k'l}$ to $\sum_{k'=1}^{k} c_{k'l} \sum_{j'=1}^{j} a_{ij'} b_{j'k'}$, when needed.

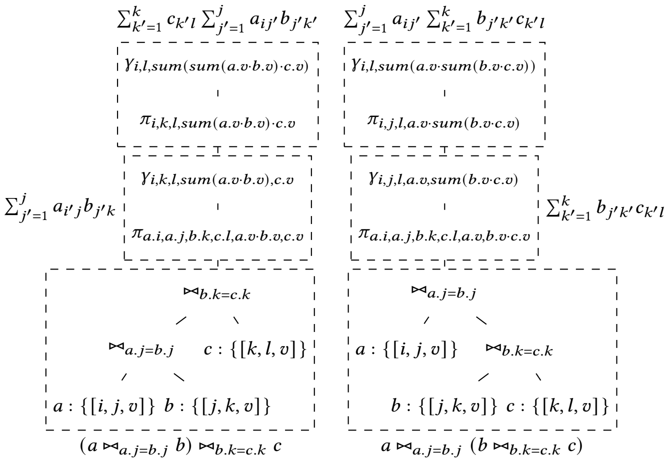

Figure 6: Unoptimised operator plan for three-way matrix multiplication of $(AB)C$ (left) and $A(BC)$ (right): the join on three relations might be reordered depending on the cardinalities. The projection and aggregation for the matrix subproduct can be pushed down above the first join.

HyPer [20, 24, 40] and Umbra are using index-based heuristics for join order optimisation [28]. As index-based heuristics exploit index structures to calculate join selectivities, they estimate join cardinalities more precisely. This is ideally suited to a relational matrix representation with an index on the dimensions used for joining. Given the densities $ds_a, ds_b, ds_{ab} \in [0, 1]$, the selectivity of a join with ds_{ab} as the single unknown parameter is given as $sel(|A \bowtie B|) = \frac{|A \bowtie B|}{|A||B|} = \frac{s_{ab}\cdot m\cdot o}{ds_a\cdot m\cdot n\cdot ds_b\cdot n\cdot o} = \frac{ds_{ab}}{n^2\cdot ds_a\cdot ds_b}$.

7 EVALUATION

For evaluation we measure the performance of ArrayQL under geo-temporal and linear algebra workloads. For the first use-case, we consider linear algebra extensions for database systems. For the latter, we compare the performance of ArrayQL in Umbra to those of popular array database systems. This section first discusses the performance of matrix operations before we proceed with the ArrayQL algebra on geo-temporal datasets.

System: All measurements have been conducted on a machine running Ubuntu 20.04 LTS, equipped with six Intel Core i7-3930K CPUs running at 3.20GHz, and offering 64 GB of main-memory.

Data: We benchmark with the New York taxi dataset[2], the science benchmark SS-DB [8] and randomly generated data.

[2] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

Competitors: The chosen competitors within popular array database systems are RasDaMan (version 10.0.0), MonetDB SciQL and SciDB (version 19.11) to benchmark geo-temporal applications. To benchmark linear algebra, we pick RMA as MonetDB's extension for linear algebra and MADlib (1.17.0 release) as an extension on top of PostgreSQL version 12.2.

7.1 Linear Algebra

This section discusses the performance of basic matrix operations as well as compound operations, either expressed in ArrayQL within Umbra or in SQL within its competitors, MADlib and RMA. MADlib provides two different representations as it distinguishes between arrays and matrices: for arrays, linear algebra operations are applied to the PostgreSQL array type, for matrices, operations expect a table in relational representation. Thus, MADlib's sparse relational representation corresponds to the underlying one for ArrayQL. In contrast, RMA uses a tabular representation. This section justifies the applicability of a relational representation for linear algebra without losing performance.

7.1.1 Matrix Algebra. This subsection presents the performance of micro-benchmarks (matrix addition and gram matrix computation) for the three matrix types (MADlib arrays, MADlib sparse matrices, RMA) against ArrayQL.

RMA's tabular representation depends on the database schema (the first dimension corresponds to the attributes, the second to the number of tuples). For benchmarking purposes, RMA provides a Python script, that creates the schema, inserts as many tuples as the specified size for the second dimension and creates SQL statements for matrix addition and gram matrix computation. For comparison, we add support to create statements for MADlib and ArrayQL and fill the relations with the same data.

We disable autocommit to measure execution time only, as it would slow down RMA dramatically. In the following, we measure the impact of the size and sparsity of a matrix on the runtime when performing matrix addition and gram matrix multiplication.

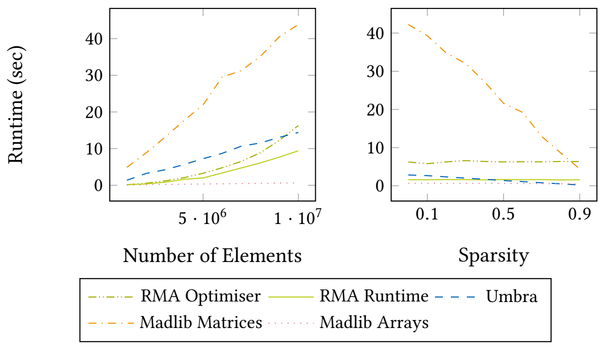

Figure 7: Evaluation of matrix addition: varying the number of elements in a dense array or the sparsity of an array with 10^6 elements.

Figure 7 shows the runtime needed for matrix addition ($X+X$), when varying the sparsity, and on dense arrays, when varying the input size. With increasing size, ArrayQL computes the matrix sum faster than RMA. RMA's compute time consists of optimisation and runtime, both increase with the size of a matrix. When varying the sparsity, MADlib matrices and Umbra benefit from sparse matrices, since zero values simply do not exist. RMA needs constant runtime with increasing sparsity as sparse and dense matrices consume the same space in a tabular representation.

Matrix addition on MADlib matrices performs the worst, whereas the same operation on MADlib arrays performs the best. This is reasonable, as the aggregation time needed to create arrays out of its relational form is not considered.

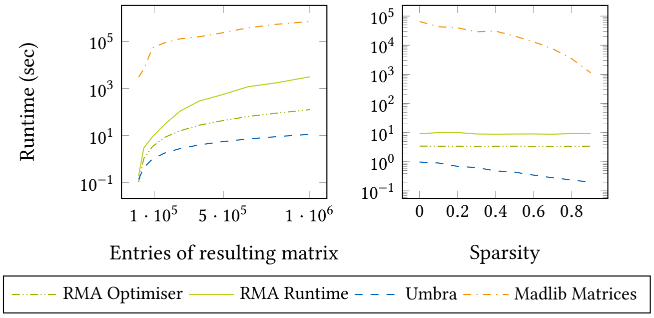

Figure 8: Evaluation of gram matrix computation: varying the number of elements in a dense array and the sparsity of a resulting matrix with 90000 entries.

Gram matrix computation ($X \cdot X^T$, see Figure 8) yields similar results: the higher the sparsity, the lower the runtime when handling MADlib matrices as well as within ArrayQL in Umbra. MADlib does not allow to transpose arrays, so gram matrix computation is not possible. Again, RMA needs constant compute time and, as the transposition is more expensive in a tabular representation, it is slower than Umbra.

When varying the input size, multiplication on MADlib matrices takes the most time. Multiplication in ArrayQL results in the shortest execution time as it is based on Umbra's relational algebra.

In summary, ArrayQL in Umbra benefits from sparse matrices as well as the performance of an in-memory database system. Therefore, our relational representation shows comparable performance to existing database extensions for linear algebra.

7.1.2 Linear Regression. We solved linear regression to benchmark composed matrix operations on a synthetic dataset. This section compares Umbra's performance with ArrayQL when using linear algebra only to the one of MADlib, which provides a dedicated table function to compute the optimal weights. Figure 9 shows the runtime of both systems when either varying the number of input tuples or the number of attributes. Our solution for ArrayQL—using matrix operations—outperforms MADlib's table function for linear regression only for a small number of input tuples or attributes. To further investigate the performance in Umbra, Figure 10 breaks down the runtime into the individual sub-operations. With increasing number of input tuples, the impact of the materialising inverse function on the runtime decreases as the inverted matrix $(X'^T X')^{-1}$ becomes relatively small. Most time is spent on the aggregation part of each matrix product (summation). Instead of using matrix algebra, a dedicated equation solve function can compute linear regression more efficiently. But calling an optimised equation solve function has the downside of materialising the input first. Nevertheless, this experiment has shown the ability to solve numerical problems when suitable table functions are available. The conception of a non-materialising table function for the matrix inversion [56] and a non-materialising equation solve function is for future work. The hash table used for summation can be further optimised: when the number of keys (indices) is known beforehand, the memory can be preallocated and the tuples prepartitioned for efficient access.

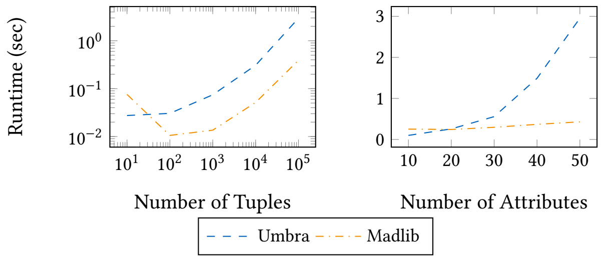

Figure 9: Runtime for solving linear regression when varying the number of tuples (50 attributes) or when varying the number of attributes (10^5 input tuples).

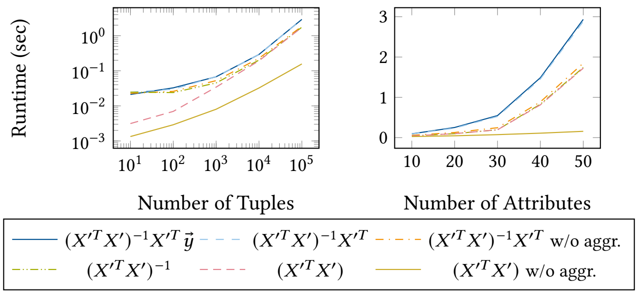

Figure 10: Runtime within Umbra broken down by operation for solving linear regression when varying the number of tuples (50 attributes) or when varying the number of attributes (10^5 input tuples).

7.2 Array Operations

This section compares the performance of ArrayQL in Umbra to the one of popular array database systems. RasDaMan[3] and MonetDB SciQL-2-NetCDF[4] ran natively on the system, a Docker container running Ubuntu 16.04 was used for SciDB[5]. We tested basic operations on the New York Taxi dataset, array operations with the SS-DB benchmark and with synthetic data.

7.2.1 New York Taxi Data. The New York taxi dataset of December 2019[6] (624 MB) provided the source for benchmarking real-world queries (see Table 3) on one-, two- and ten-dimensional arrays. Queries Q1, Q3 and Q7 benchmark projections, whereas queries Q2, Q4, Q5, Q6 and Q8 benchmark aggregation functions like summation, average and count, and queries Q9/Q10 modify the array bounds. In detail, Q1 retrieves all vendor ID attributes of the source array. Q2 sums up the total distance driven. Q3 computes the distance ratio of one ride to the total distance driven. Q4 returns the maximum duration of a trip. Q5 returns the average total amount (payment), whereas Q6 calculates the average payment per customer excluding trips with no passengers. Q7 returns all attributes of trips with four or more customers. Q8

[3] https://doc.rasdaman.org/index.html
[4] https://dev.monetdb.org/hg/MonetDB/shortlog/SciQL-2-NetCDF
[5] https://github.com/rvernica/docker-library
[6] https://nyc-tlc.s3.amazonaws.com/trip+data/yellow_tripdata_2019-12.csv

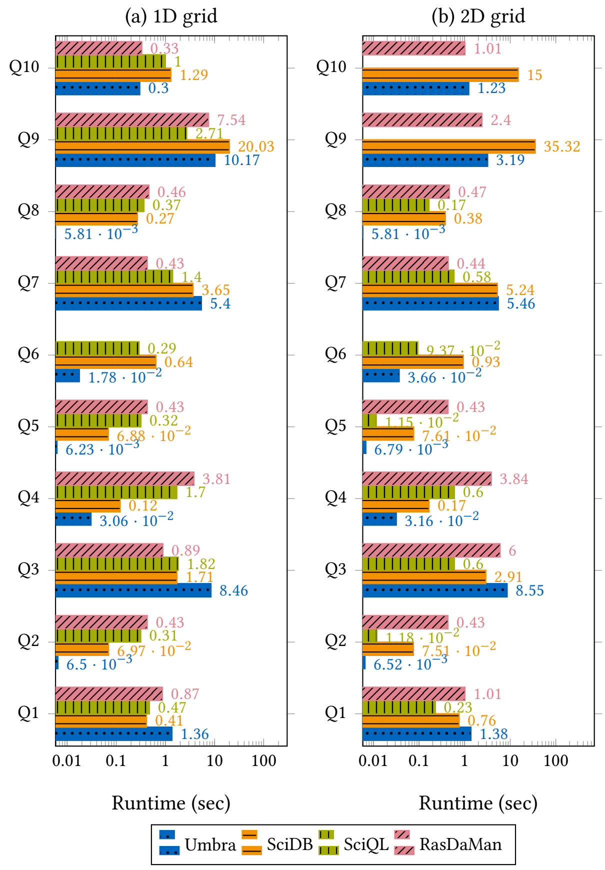

Figure 11: Runtimes of proposed queries on the New York taxi dataset with (a) one- and (b) two-dimensional indices.

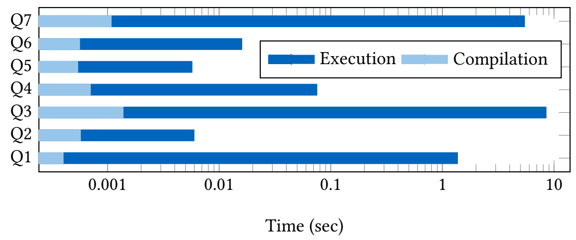

Figure 12: Impact of code-generation on selected ArrayQL queries in Umbra: Compilation time vs. runtime.

Q1	`SELECT VendorID FROM taxiData;`
Q2	`SELECT SUM(trip_distance) FROM taxiData;`
Q3	`SELECT 100.0*trip_distance/tmp.total_distance FROM taxiData, (SELECT SUM(trip_distance) as total_distance FROM taxiData) as tmp;`
Q4	`SELECT MAX((tpep_dropoff_datetime - tpep_pickup_datetime) + (end_time - start_time)) FROM taxiData;`
Q5	`SELECT AVG(total_amount) FROM taxiData;`
Q6	`SELECT AVG(total_amount/passenger_count) FROM taxiData WHERE passenger_count <> 0;`
Q7	`SELECT * FROM taxiData WHERE passenger_count>=4;`
Q8	`SELECT COUNT(*) FROM taxiData WHERE payment_type=1;`
Q9	`SELECT [0:1048574] as i, * FROM taxiData[i+1];`
Q10	`SELECT [42:42000] as i, * FROM taxiData[i];`

Table 3: ArrayQL queries on the New York taxi dataset, the corresponding AQL, RasQL and SciQL queries are omitted.

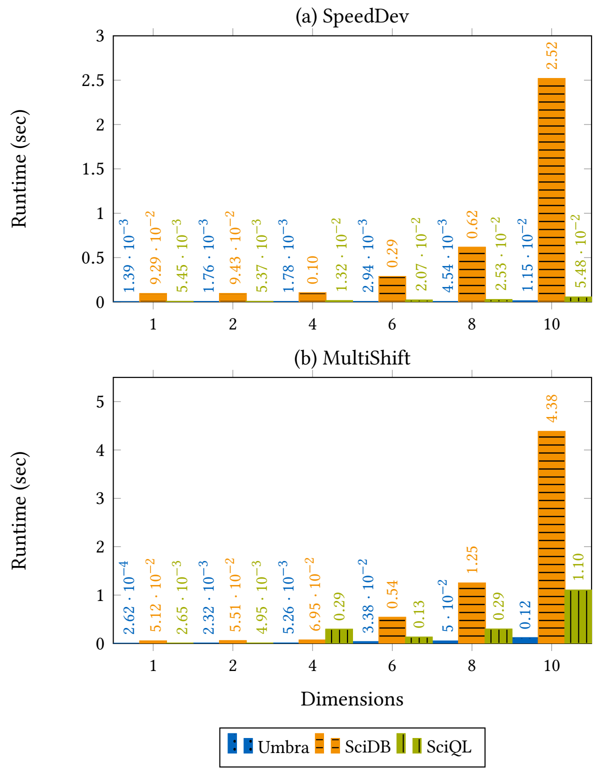

Figure 13: Impact of dimensionality on the runtime.

SpeedDev	`SELECT MAX(3600.0*(tmp3.v3-tmp1.v)) as speed FROM (SELECT pickup_day, AVG(trip_distance/total_duration) as v FROM taxiData WHERE total_duration<>0 GROUP BY pickup_day) as tmp1, (SELECT AVG(trip_distance/total_duration) as v3 FROM taxiData WHERE total_duration <> 0) as tmp3;`
MultiShift	`SELECT [pickup_day] as a, ..., * FROM taxiData[a+1, ...]`

Table 4: Multi-dimensional queries (New York taxi data).

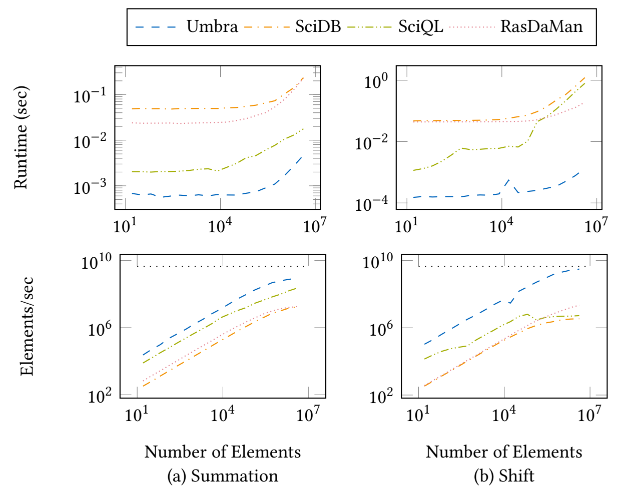

Figure 14: Runtime and throughput of an aggregation (summation) or shifting the indices of a two-dimensional matrix depending on the number of elements.

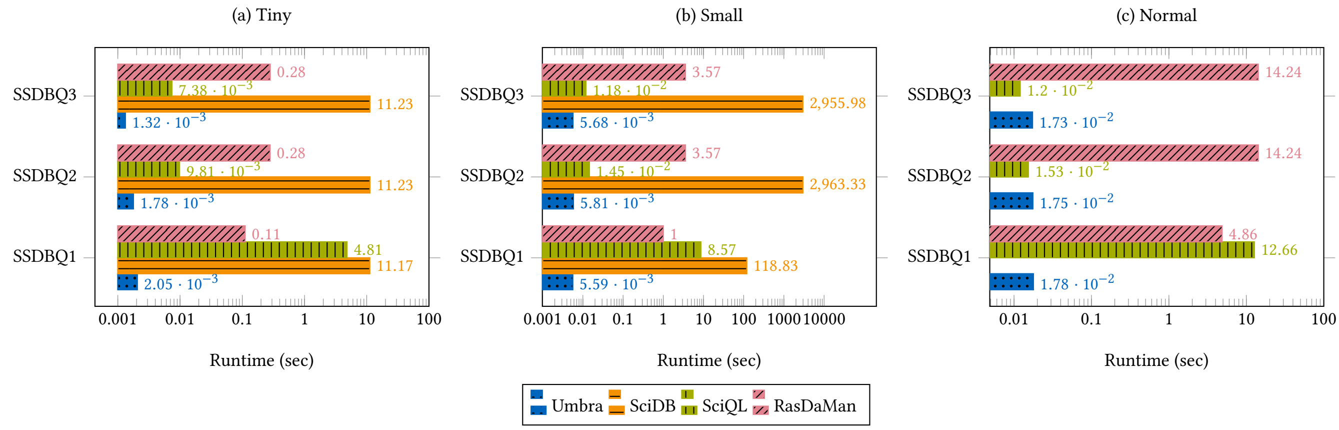

Figure 15: SS-DB benchmark with datasets of size (a) tiny, 58 MB, (b) small, 844 MB, and (c) normal, 3.4 GB.

SSDBQ1	ArrayQL:	SELECT AVG(a) FROM ssDB[0:19]
	AQL:	SELECT AVG(a) FROM subarray(very_small,0,0,0,19,1599,1599)
	SciQL:	SELECT AVG(a) FROM ssDB[0:19]
	RasQL:	SELECT AVG_CELLS(r.a * (r.z<=19)) FROM ssDB as r
SSDBQ2	ArrayQL:	SELECT AVG(a) FROM (SELECT [z], [x] as s, [y] as t, * FROM ssDB[0:19, s+4, t+4] WHERE s%2 = 0 AND t%2 = 0) as tmp GROUP BY z
	AQL:	SELECT AVG(a) FROM reshape(subarray(ssDB, 0,0,0,19,999,999), <a:int32, b:int32, c:int32, d:int32, e:int32, f:int32,g:int32,h:int32,i:int32,j:int32, k:int32> [z=0:19,1000000,0,x=4:1003,1000000,0,y=4:1003,1000000,0]) WHERE x%2=0 and y%2=0 GROUP BY z
	SciQL:	SELECT AVG(a) FROM (SELECT [z], [x+4] as s, [y+4] as t, * FROM ssDB[0:19] WHERE x%2 = 0 AND y%2 = 0) as tmp GROUP BY z
	RasQL:	SELECT ADD_CELLS(shift(r.a * (r.z<=19 and mod(r.x, 2) = 0 and mod(r.y, 2) = 0), [4,4])) / COUNT_CELLS(shift((r.z<=19 and mod(r.x,2) = 0 and mod(r.y,2) = 0) , [4,4])) FROM ssDB as r
SSDBQ3	ArrayQL:	SELECT AVG(a) FROM (SELECT [z], [x] as s, [y] as t, * FROM ssDB[0:19, s+4, t+4] WHERE s%4 = 0 AND t%4 = 0) as tmp GROUP BY z
	AQL:	SELECT AVG(a) FROM reshape(subarray(ssDB, 0,0,0,19,999,999), <a:int32, b:int32, c:int32, d:int32, e:int32, f:int32,g:int32,h:int32, i:int32,j:int32, k:int32> [z=0:19,1000000,0 ,x=4:1003,1000000,0, y=4:1003,1000000,0]) WHERE x%4=0 and y%4=0 GROUP BY z
	SciQL:	SELECT AVG(a) FROM (SELECT [z], [x+4] as s, [y+4] as t, * FROM ssDB[0:19] WHERE x%4 = 0 AND y%4 = 0) as tmp GROUP BY z
	RasQL:	SELECT ADD_CELLS(shift(r.a * (r.z<=19 and mod(r.x, 4) = 0 and mod(r.y, 4) = 0), [4,4])) / COUNT_CELLS(shift((r.z<=19 and mod(r.x,4) = 0 and mod(r.y,4) = 0) , [4,4])) FROM ssDB as r

Table 5: Performed queries on the SS-DB dataset.

retrieves the number of how often a payment method was used. Q9 reboxes the array indices (slice and shift). Q10 slices the array to return a subarray.

Figure 11 shows the runtime of the presented queries on the New York taxi dataset stored as a one- or two-dimensional grid. To be comparable to the array database systems, which store the data as a dense grid, we added a synthetic key to the data in Umbra. ArrayQL within Umbra performs well on computing aggregates (Q2, Q4, Q5, Q6 and Q8) and slicing the array (Q10). The runtime includes the time for printing to `/dev/null`, which influenced the runtime for queries returning multiple array entries (Q1, Q3, Q7, Q9) as the index was attached to every value. The compilation time of ArrayQL queries was negligible (see Figure 12). Furthermore, the performance of Umbra was not heavily influenced by additional predicates (Q8) or computations (Q4).

The architecture of RasDaMan ensures efficient execution of operations that change the dimensions and was the fastest system to retrieve specific data (Q7). SciDB's performance was mostly superior to the one of RasDaMan (Q1, Q2, Q4, Q5), but the *reshape* operator slowed down the performance on array transformations (Q9, Q10). In summary, the current Umbra-ArrayQL prototype ensures high performance on subarray accesses through its index structure and separately stored dimension indices. Furthermore, aggregations and computations based on the data itself can be processed effectively.

To measure the impact of involved dimensions on aggregations (see Table 4), we also stored the taxi data as a ten-dimensional array. Query *SpeedDev* calculates the maximum deviation of the average speed per day compared to the average speed of the whole dataset, query *MultiShift* shifts all array's dimensions. Figure 13 shows the runtime in dependency on the number of dimensions. For both queries, the runtime on all tested systems scaled linearly with an increasing number of dimensions. While Umbra and MonetDB took a similar time to run the query *SpeedDev*, SciDB could not compute it as fast as the others. For the query *MultiShift*, MonetDB SciQL treats high-dimensional arrays efficiently. Umbra outperforms both competitors in compute time (the time for printing the indices not considered). SciDB was slower for all arrays although we adapted the query to avoid the *reshape* operator.

7.2.2 Random Data. Figure 14 shows the runtime and throughput on two-dimensional arrays with synthetic data and an increasing number of elements. For both tests, summation and shifting the indices, the runtime scales linearly with the number of elements. Umbra is the fastest system when performing aggregates, but shifting slows down the performance as all indices have to be changed. The upper constant lines in the lower diagrams display the maximum throughput of $4.5 \cdot 10^9$ elements per second (based on a measured[7] memory bandwidth of 36 GB/s divided through 8 B, the size of a double precision floating point number). This shows that ArrayQL in Umbra best approaches the maximum possible performance, with only a factor of ten in between.

7.2.3 SS-DB Data. SS-DB is a benchmark for scientific workloads that simulates astronomical data for array-oriented processing. A data generator[8] synthesises three-dimensional data—one dimension identifies the tile, two dimensions determine a cell—with eleven attributes each. The queries SSDBQ1/2/3 (see Table 5), adapted from a paper comparing SciQL and SciDB [37], compute the average of one attribute for 20 tiles. SSDBQ2 and SSDBQ3 do the same but consider fewer cells (50 % and 25 % of the original

[7] https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html
[8] https://www.xldb.org/science-benchmark/

size). All queries were tested on an image of size tiny (58 MB), small (844 MB) and normal (3.4 GB).

Figure 15 presents the measured runtimes. For SSDBQ1, Umbra may use its index structure on the array's dimensions to compute the result faster than the others on all datasets. Furthermore, Umbra is not as heavily affected by the increase of the data size as the other systems. SSDBQ2 and SSDBQ3 combine array operations like shifting with specific attribute selection and aggregation functions. Although SSDBQ2 considers more values than SSDBQ3, this does not affect the runtime. As seen in Section 7.2.1, MonetDB can process shift-operations more easily than Umbra, whereas Umbra performs better on aggregations. Summarised, both systems show similar performance for these queries. The tested SciDB version took the longest to adjust the index for the smaller subarray.

8 CONCLUSION

In this paper, we have integrated ArrayQL into a code-generating database system as another query interface and addressable inside SQL as user-defined functions. As this standardised array query language has not yet been integrated into a productive system, we completed its grammar specification and extended Umbra's query engine to accept ArrayQL statements. For that reason, we defined a relational array model and translated ArrayQL operators to relational algebra. We demonstrated the suitability of an array query language for geo-temporal data by the SS-DB benchmark and by the New York taxi data as a real-world example. Moreover this language can be used for linear algebra to compute machine learning tasks. For basic matrix operations, ArrayQL statements performed better than state-of-the-art linear algebra extensions for database systems, whereas materialising table-functions as needed for inversion slowed down the runtime. For geo-temporal tasks, ArrayQL outperformed traditional array database systems, when performing aggregations or filtering data by a predicate.

REFERENCES

[1] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964.
[2] Gustavo Alonso et al. 2019. doppioDB 1.0: Machine Learning inside a Relational Engine. *IEEE Data Eng. Bull.* 42, 2 (2019), 19–31.
[3] Peter Baumann et al. 1998. The Multidimensional Database System RasDaMan. In *SIGMOD Conference.* ACM Press, 575–577.
[4] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.
[5] Robert Brijder et al. 2018. On the Expressive Power of Query Languages for Matrices. In *ICDT (LIPIcs, Vol. 98).* 10:1–10:17.
[6] Cheng Chen et al. 2021. Optimizing An In-memory Database System For AI-powered On-line Decision Augmentation. *PVLDB* 14, 5 (2021), 799–812.
[7] Philippe Cudré-Mauroux et al. 2009. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB* 2, 2 (2009), 1534–1537.
[8] Philippe Cudre-Mauroux et al. 2010. SS-DB: A standard science dbms benchmark. *Under submission* (2010).
[9] Angjela Davitkova et al. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT.* 407–410.
[10] Oksana Dolmatova et al. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *SIGMOD Conference.* ACM, 2573–2587.
[11] Jennie Duggan et al. 2015. Skew-Aware Join Optimization for Array Databases. In *SIGMOD Conference.* ACM, 123–135.
[12] Ahmed Elgohary et al. 2018. Compressed linear algebra for large-scale machine learning. *VLDB J.* 27, 5 (2018), 719–744.
[13] Franz Färber et al. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.
[14] Zekai J. Gao et al. 2017. The BUDS Language for Distributed Bayesian Machine Learning. In *SIGMOD Conference.* ACM, 961–976.
[15] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD Conference.* ACM Press, 102–111.
[16] Anja Gruenheid, Edward Omiecinski, and Leo Mark. 2011. Query optimization using column statistics in hive. In *IDEAS.* ACM, 97–105.
[17] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB@VLDB.*
[18] Joseph M. Hellerstein et al. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.
[19] Axel Hertzschuch et al. 2021. Simplicity Done Right for Join Ordering. In *CIDR.*
[20] Nina Hubig et al. 2017. HyPerInsight: Data Exploration Deep Inside HyPer. In *CIKM.* ACM, 2467–2470.
[21] Fuad T. Jamour et al. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *EuroSys.* ACM, 27:1–27:15.
[22] Matthias Jasny et al. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD Conference.* ACM, 159–173.
[23] Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. 2021. Umbra as a Time Machine. In *BTW (LNI, Vol. P-311).* GI, 123–132.
[24] Alfons Kemper et al. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE.* IEEE, 195–206.
[25] David Kernert et al. 2014. SLACID - sparse linear algebra in a column-oriented in-memory database system. In *SSDBM.* ACM, 11:1–11:12.
[26] Andreas Kunft et al. 2016. Bridging the gap: towards optimization across linear and relational algebra. In *BeyondMR@SIGMOD.* ACM, 1.
[27] Andreas Kunft et al. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.
[28] Viktor Leis et al. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR.*
[29] Xupeng Li et al. 2017. MLog: Towards Declarative In-Database Machine Learning. *PVLDB* 10, 12 (2017), 1933–1936.
[30] Kian-Tat Lim et al. 2012. ArrayQL syntax. In *XLDB.* http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf
[31] Raoni Lourenço, Juliana Freire, and Dennis E. Shasha. 2019. Debugging Machine Learning Pipelines. In *DEEM@SIGMOD.* ACM, 3:1–3:10.
[32] Shangyu Luo et al. 2017. Scalable Linear Algebra on a Relational Database System. In *ICDE.* IEEE, 523–534.
[33] David Maier et al. 2012. ArrayQL algebra: version 3. In *XLDB.* http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf
[34] Ingo Müller et al. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference.* ACM, 115–130.
[35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.
[36] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR.*
[37] Holger Pirk, Ying Zhang, Stefan Manegold, and Martin Kersten. 2013. An evaluation of ad-hoc queries on arrays in MonetDB.
[38] Maximilian Schleich et al. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD Conference.* ACM, 1642–1659.
[39] Josef Schmeißer et al. 2021. B^2-Tree: Cache-Friendly String Indexing within B-Trees. In *BTW (LNI, Vol. P-311).* GI, 39–58.
[40] Maximilian E. Schüle et al. 2017. Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. *PVLDB* 10, 12 (2017), 1921–1924.
[41] Maximilian E. Schüle et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW (LNI, Vol. P-289).* GI, 247–266.
[42] Maximilian E. Schüle et al. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT.* 562–565.
[43] Maximilian E. Schüle et al. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD.* ACM, 7:1–7:4.
[44] Maximilian E. Schüle et al. 2019. The Power of SQL Lambda Functions. In *EDBT.* 534–537.
[45] Maximilian E. Schüle et al. 2020. ARTful Skyline Computation for In-Memory Database Systems. In *ADBIS (CCIS, Vol. 1259).* Springer, 3–12.
[46] Maximilian E. Schüle et al. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM.* ACM, 6:1–6:12.
[47] Maximilian E. Schüle et al. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM.* ACM, 25–36.
[48] Maximilian E. Schüle et al. 2021. TardisDB: Extending SQL to Support Versioning. In *SIGMOD Conference.* ACM, 2775–2778.
[49] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2021. ArrayQL for Linear Algebra within Umbra. In *SSDBM.* ACM, 193–196.
[50] Amir Shaikhha et al. 2019. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Prog. Lang.* 3, ICFP (2019), 97:1–97:30.
[51] Johanna Sommer et al. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD Conference.* ACM, 1607–1623.
[52] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. 2012. Distributed Matrix Completion. In *ICDM.* IEEE, 655–664.
[53] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *PVLDB* 13, 11 (2020), 1919–1932.
[54] Steven Whang and Jae-Gil Lee. 2020. Data Collection and Quality Challenges for Deep Learning. *PVLDB* 13, 12 (2020), 3429–3432.
[55] Lucas Woltmann et al. 2021. Aggregate-based Training Phase for ML-based Cardinality Estimation. In *BTW (LNI, Vol. P-311).* GI, 135–154.
[56] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. 2014. Scalable matrix inversion using MapReduce. In *HPDC.* ACM, 177–190.
[57] Eleni Tzirita Zacharatou et al. 2021. The Case for Distance-Bounded Spatial Approximations. In *CIDR.*
[58] Steffen Zeuch et al. 2016. Non-Invasive Progressive Optimization for In-Memory Databases. *PVLDB* 9, 14 (2016), 1659–1670.
[59] Ying Zhang, Martin L. Kersten, and Stefan Manegold. 2013. SciQL: array data processing inside an RDBMS. In *SIGMOD Conference.* ACM, 1049–1052.