ArrayQL for Linear Algebra within Umbra

Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann
Technical University of Munich
Germany
{m.schuele,tobias.goetz}@tum.de,{kemper,neumann}@in.tum.de

ABSTRACT

Array database systems offer a declarative language for array-based access on multidimensional data. 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. In our evaluation, ArrayQL within Umbra computes matrix operations faster than state of the art database extensions.

ACM Reference Format:
Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2021. ArrayQL for Linear Algebra within Umbra. In *33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021), July 6–7, 2021, Tampa, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3468791.3468838

1 INTRODUCTION

Array database systems are developed for geo-temporal data and therefore specialised for multidimensional discrete data (MDD) [1]. In contrast to relational database systems, array database systems are designed for index-based array access [3, 6, 14, 19] and excel in computing aggregations on numerical data. Popular array database systems are RasDaMan [2], MonetDB SciQL [22] and SciDB [4, 10]. As each one is shipped with its own query language, ArrayQL [11] is an attempt to standardise them as presented at XLDB 2012. Although the corresponding algebra [12] has been published, it is not fully covered by the corresponding draft of a grammar specification [11] 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 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.

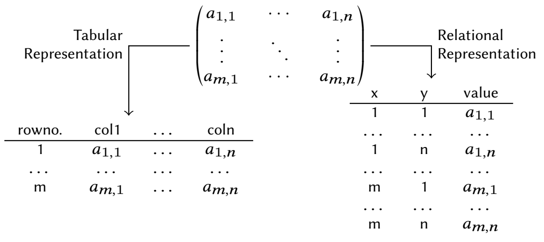
Tabular Representation
Relational Representation
$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$
rowno. | col1 | ... | coln
1 | $a_{1,1}$ | ... | $a_{1,n}$
... | ... | ... | ...
m | $a_{m,1}$ | ... | $a_{m,n}$
x | y | value
1 | 1 | $a_{1,1}$
... | ... | ...
1 | n | $a_{1,n}$
... | ... | ...
m | 1 | $a_{m,1}$
... | ... | ...
m | n | $a_{m,n}$

Figure 1: Tabular representation (left) with the attributes as columns and a relational representation (right) with the array as coordinate list.

Another use case for array-oriented data processing arises by the need of matrix operations for data mining and machine learning. The corresponding data is often stored and collected inside relational database systems [15, 21], 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 [8, 16, 20], which essentially depend on data stored in tensors and their transformations [17, 18], 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 [9, 13]. 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 provides a translation of the afore-mentioned ArrayQL operators into relational algebra and a corresponding grammar. This study's specific contributions are:

- 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.

This study comprises the following sections: Section 2 presents the architecture when integrating ArrayQL within the beyond main-memory database system Umbra as the target. Section 3 introduces the ArrayQL algebra and its translation into relational algebra. Section 4 evaluates the proposed extension using micro-benchmarks for basic matrix algebra.

2 IN-DATABASE INTEGRATION

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 2). 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 2: Array creation.

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 1). 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 1: ArrayQL as part of a user-defined function returns either an SQL table or an SQL array.

3 ARRAYQL ALGEBRA

ArrayQL offers an algebra [12] 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.

3.1 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 2: Rename operator.

3.2 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 3: Function application: addition.

3.3 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 4: Explicit and implicit filter operator.

3.4 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 $i'_1, \ldots, i'_n \in \mathbb{I}$:

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

Operator	Input	Output	Validity Map	Relational Algebra
apply	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}, f \in (\mathbb{R} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}$	$d_a = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\pi_{i_1,\dots i_n,f(v)}(a)$
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	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}$	$\mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}$	$d_a \subseteq d_{out} = \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\dots 0_{\lvert a.i_1\rvert,\dots,\lvert a.i_n\rvert} \cdots$
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	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}, i_1^l, i_1^u, \dots, i_n^l, i_n^u \in \mathbb{I}$	$\mathbb{R}^{i_1^u-i_1^l\times\cdots\times i_n^u-i_n^l}$	$d_a \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert, d_{out} \subseteq i_1^u - i_1^l \times \cdots \times i_n^u - i_n^l$	$\sigma_{i_1^l \le i_1 \le i_1^u \wedge\cdots\wedge i_n^l \le i_n \le i_n^u}(a)$
reduce	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}, f \in (\mathbb{R}^{\lvert i_n\rvert} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_{n-1}\rvert}$	$d_a \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert, d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_{n-1}\rvert$	$\gamma_{i_1,\dots,i_n,f(v)}(a)$
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	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}, i'_1, \dots, i'_n \in \mathbb{I}$	$\mathbb{R}^{\lvert i_1\rvert\times\cdots\times\lvert i_n\rvert}$	$d_a = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\pi_{i_1+i'_m,\dots,i_n+i'_n,v}(a)$

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. $i_{1\dots n}$ represents the attribute for the dimension in relational form, $\lvert i_{1\dots n}\rvert$ denotes the size of a dimension. We assume arrays having a single attribute $v \in \mathbb{R}$ only.

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

Listing 5: 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 $i_1^l, i_1^u, \dots, i_n^l, i_n^u \in \mathbb{I}$:

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

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

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

Listing 6: Rebox operator.

3.5 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:

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

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

Listing 7: The keyword `FILLED` enables the fill operator.

3.6 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.

3.6.1 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:

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

```sql
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 8: Combine operator.

3.6.2 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).

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

Listing 9: Inner dimension Join.

3.7 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:

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

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

Listing 10: Reduce operator for aggregation: summation

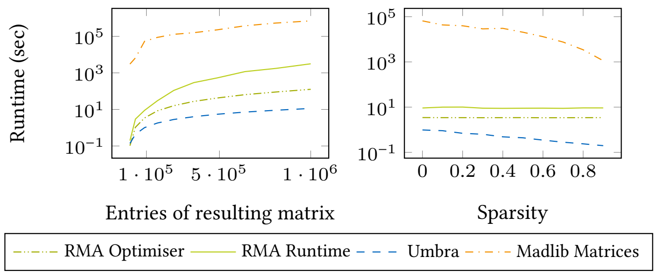

Figure 4: Evaluation of gram matrix computation: varying the number of elements in a dense array and the sparsity on a resulting matrix with 90000 **entries.**

4 EVALUATION

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 62 GB of main-memory.

Competitors: To benchmark linear algebra, we pick RMA [5] as MonetDB's extension for linear algebra and MADlib (1.17.0 release) [7] as an extension on top of PostgreSQL version 12.2.

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.

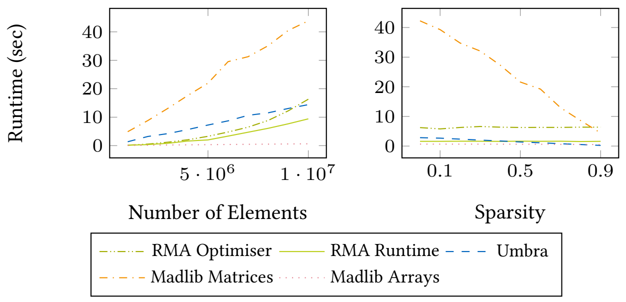

Figure 3: Evaluation of matrix addition: varying the number of elements in a dense array or the sparsity on an array with 10^6 **elements.**

Figure 3 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.

Gram matrix computation ($X \cdot X^T$, see Figure 4) 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.

5 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. 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.

REFERENCES

[1] Peter Baumann. 1993. Database Support for Multidimensional Discrete Data. In *SSD (LNCS, Vol. 692)*. Springer, 191–206.
[2] Peter Baumann, Andreas Dehmel, Paula Furtado, et al. 1998. The Multidimensional Database System RasDaMan. In *SIGMOD*. ACM Press, 575–577.
[3] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, et al. 2011. SciHadoop: array-based query processing in Hadoop. In *SC*. ACM, 66:1–66:11.
[4] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, et al. 2009. A Demonstration of SciDB: A Science-Oriented DBMS. *VLDB* 2, 2 (2009), 1534–1537.
[5] Oksana Dolmatova, Nikolaus Augsten, et al. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *SIGMOD*. ACM, 2573–2587.
[6] Tingjian Ge and Stanley B. Zdonik. 2010. A*-tree: A Structure for Storage and Modeling of Uncertain Multidimensional Arrays. *VLDB* 3, 1 (2010), 964–974.
[7] Joseph M. Hellerstein, Christopher Ré, et al. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.
[8] Nina Hubig, Linnea Passing, et al. 2017. HyPerInsight. In *CIKM*. ACM, 2467–2470.
[9] Lukas Karnowski et al. 2021. Umbra as a Time Machine. In *BTW (LNI)*. GI.
[10] Sangchul Kim et al. 2016. Selective Scan for Filter Operator of SciDB. In *SSDBM*.
[11] Kian-Tat Lim, David Maier, J. Becla, et al. 2012. ArrayQL syntax. In *XLDB*. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf
[12] David Maier, Peter Baumann, et al. 2012. ArrayQL algebra: version 3. In *XLDB*. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf
[13] Thomas Neumann and Michael J. Freitag. 2020. Umbra. In *CIDR*.
[14] Josef Schmeißer et al. 2021. B^2-Tree. In *BTW (LNI)*. GI.
[15] Maximilian E. Schüle et al. 2017. Monopedia. *VLDB* 10, 12 (2017), 1921–1924.
[16] Maximilian E. Schüle et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for MMDBS. In *BTW (LNI)*. GI.
[17] Maximilian E. Schüle et al. 2019. ML2SQL. In *EDBT*. 562–565.
[18] Maximilian E. Schüle et al. 2019. MLearn. In *DEEM@SIGMOD*. ACM, 7:1–7:4.
[19] Maximilian E. Schüle et al. 2020. ARTful Skyline Computation for In-Memory Database Systems. In *ADBIS (CCIS, Vol. 1259)*. Springer, 3–12.
[20] Maximilian E. Schüle et al. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM*. ACM.
[21] Maximilian E. Schüle et al. 2021. TardisDB. In *SIGMOD*. ACM.
[22] Ying Zhang, Martin L. Kersten, and Stefan Manegold. 2013. SciQL: array data processing inside an RDBMS. In *SIGMOD*. ACM, 1049–1052.