Recursive SQL for Data Mining

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

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

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

ABSTRACT

To implement algorithms within database systems beyond the design of SQL as a data query language, library functions or external tools were used that require the extraction of data first. To eliminate the need of data extraction out of database systems, we argue that SQL-92 plus recursive tables is capable of expressing user-defined algorithms. To underline this claim, we transform selected algorithms out of graph mining, clustering and association rule analysis into recursive common table expressions (CTEs). We compare their performance to the one of user-defined functions and external tools. Our evaluation shows a competitive performance when using recursive CTEs to library functions either when using a disk-based database systems or a modern in-memory engine.

ACM Reference Format:
Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. 2022. Recursive SQL for Data Mining. In *34th International Conference on Scientific and Statistical Database Management (SSDBM 2022), July 6–8, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3538712.3538746

1 INTRODUCTION

The performance increase of database servers through modern hardware allows database systems to be used for more than pure data management tasks [2, 6, 9, 15, 19, 38, 40, 41]. Database systems provide with SQL a declarative language to specify what to do rather than caring about optimisation details. The platform independence and reusability of SQL increases the incentive to execute complex algorithms already in the database system [3, 7, 8, 17]. SQL provides common table expressions (CTEs) that allow structuring and modularising an SQL query and which are part of the holistic query optimisation [6]. Furthermore, recursive CTEs compute the transitive closure, which allow SQL-92 to be Turing-complete.

In order for database servers to take over more application logic, database users should be able to develop algorithms independently of systems developers. In a previous publication [34], we argued that a domain-specific language with procedural constructs and embedded SQL (*HyPerScript*) is needed to eliminate the need for developing an operator in each case. So we identified building blocks to implement selected data mining algorithms as user-defined functions (UDFs) [5, 20, 39, 42] in SQL.

SSDBM 2022, July 6–8, 2022, Copenhagen, Denmark
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9667-7/22/07…$15.00
https://doi.org/10.1145/3538712.3538746

In this study, we argue that transferring procedural constructs into SQL-92 is possible. We start by converting each iteration within a procedural loop into a recursion [12, 13, 16] to provide SQL-92 only implementations for clustering (DBSCAN, k-Means), graph mining (PageRank) and association rule analysis (Apriori). We compare their performance in PostgreSQL and HyPer [24, 25], an in-memory database system [21, 22, 27, 32, 35], to their UDF counterparts and dedicated table-operators: MADlib [10] in PostgreSQL and data mining operators in HyPer [11, 18, 31, 37].

This study's contributions are data mining operators written in SQL-92 only and a comprehensive evaluation to compare their performance to existing library functions and procedural counterparts as part of UDFs. This paper is structured as follows: Section 2 describes the implementation of data mining algorithms in SQL. The evaluation in Section 3 measures the performance in dependency on the input size and the number of available threads. Section 4 concludes by an outlook on further applications for recursive CTEs.

2 SQL FOR DATA ANALYSIS

This section presents a set of data analysis algorithms written in recursive SQL: algorithms for association rule analysis (Apriori), clustering (k-Means and DBSCAN) and graph metrics (PageRank).

2.1 PageRank

PageRank [4] is a graph mining algorithm designed to determine the importance of web pages. Each web page is called a node $n \in N$; a link directing to another web page is called an edge $(s, d) \in N \times N$. Initially, each node receives the same PageRank value pr_0 (Equation 1). In each iteration, each node s distributes its own value $pr_i(s)$ equally to all outgoing edges $(s, d) \in E$. The new PageRank value $pr_{i+1}(n)$ of a node n is the sum of the values of all incoming edges $(s, n) \in E$, possibly damped by a factor α (Equation 2):

$$pr_0(n) := \frac{1}{|N|}, \tag{1}$$

$$pr_{i+1}(n) := \alpha \cdot \sum_{(s,n)\in E} \frac{pr_i(s)}{|\{d \mid (s,d) \in E\}|} + \frac{1-\alpha}{|N|}. \tag{2}$$

```sql
with recursive pagerank (iter,node,pr) as (
 select 0, e.dst, 1::float/(select count(distinct dst) from
      prscript.edges)
 from prscript.edges e group by e.dst
union all
 select iter+1,dst,0.1*((1::float/(select count(distinct dst) from
       prscript.edges)))+0.9*sum(b)
 from (
  select iter, e.dst, p.pr/(select count (*) from prscript.edges x
        where x.src=e.src) as b
  from prscript.edges e, pagerank p
  where e.src=p.Node and iter < 100 ) i
 group by dst, iter
) select * from pagerank where iter=100;
```

Listing 1: PageRank in SQL with a recursive table pagerank and $\alpha = 0.9$.

Both equations can be described in SQL with aggregations, whereas for the iteration we need either loops of a scripting language like HyPerScript or a recursive table (see Listing 1). We expect one relation containing the edges. The base case (line 2/3) computes the initial PageRank value pr_0. The recursive step first divides each node's PageRank value by the number of outgoing edges (line 7) and assigns this fraction to the destination node `dst`. It then sums up the fractions for each destination node (line 5-10).

2.2 Clustering

Clustering is an important area of data analysis that groups similar tuples. We consider k-Means and DBSCAN [23], as both algorithms are integrated into HyPer as operators.

k-Means assigns n-dimensional points $x \in P \subset \mathrm{R}^n$ to k clusters C with k points forming the initial centres $C_0 \subset P, |C_0| = k$. A point belongs to the closest located cluster $c \in C$ based on a metric like Euclidean distance ($||c - x||_2$, Equation 3 returns all points of a cluster). In each iteration, the new centre is computed as the average of all points (Equation 4):

$$cluster(c) = \{x | x \in P : \nexists d \in C : d \neq c \wedge ||d - x||_2 < ||c - x||_2\}, \tag{3}$$

$$c_{i+1} = \sum_{x \in cluster(c_i)} \frac{x}{|cluster(c_i)|}. \tag{4}$$

Using a recursive table (see Listing 2), k tuples are initially selected as centres (line 2), whose coordinates get updated in each iteration (line 4-8). The implementation for computing the centres of k-Means is based on a window function that calculates a ranking of the closest centres per point (line 5/6). The mean values of the assigned points form the new centres (line 4).

```
1  with recursive clusters (iter, cid, x, y) as (
2    (select 0,id, x, y from kmeansscript.points limit 5)
3  union all
4    select iter+1,cid, avg(px), avg(py) from (
5      select iter, cid, p.x as px, p.y as py, rank() over (partition
           by p.id
6        order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc, (c.x*c
           .x+c.y*c.y) asc)
7      from kmeansscript.points p, clusters c) x
8    where x.rank=1 and iter<100 group by cid, iter
9  )
10 select * from clusters where iter=100;
```

Listing 2: k-Means in SQL with a recursive table `clusters`: *k*=5, 100 iterations.

DBSCAN clusters points depending on a parameter ϵ, that describes the maximal distance between two points, and the minimal number of points per cluster $minPoints > 1$, declared as noise otherwise. For every point within a cluster $x \in G \subset P \subset \mathrm{R}^n$, another point $y \in G$ exists, whose distance to x is less than ϵ:

$$\forall x \in G : \exists y \in G : x \neq y \wedge ||x - y||_2 < \epsilon. \tag{5}$$

When the database system supports aggregate functions within recursive tables, the clusters can be expanded recursively (see Listing 3): First, each point forms its own cluster (line 2). Then, clusters that are less than ϵ away are merged (line 4-7).

For comparison, we use the operators k-Means and DBSCAN implemented in HyPer. As a special feature, both operators are written directly in LLVM code, are compiled directly into the query and avoid expensive function calls. Both operators are pipeline breakers, but the centre calculation for k-Means does not require materialisation of the data points as it copies the underlying operator tree per iteration.

```
1 with recursive dbscan(iter,id,x,y,clusterid,noise) as (
2   select 0,id,x,y,id,true from dbscanscript.points
3 union all
4   select iter+1,p.id,p.x,p.y, min(c.clusterid), count(*) < 3
5   from dbscanscript.points p, dbscan c
6   where iter<10 and (p.x-c.x)^2+(p.y-c.y)^2<1.5^2
7   group by iter,p.id,p.x,p.y
8 ) select * from dbscan where iter=10;
```

Listing 3: DBSCAN in SQL with a recursive table `dbscan`: 10 **iterations,** $\epsilon = 1.5$, $minPoints = 3$.

2.3 Association Rule Analysis

The Apriori algorithm [1], introduced in 1993, is the best-known representative in the field of association rule analysis. It is based on shopping cart data stored as tuples out of transaction number (*tid*) and item (sales: $\{[tid, item]\}$). First, it selects frequent item sets that occur with a minimum relative frequency, *support*, of at least s_0 in all shopping carts, which are used to create association rules. In each iteration, the item sets grow by one element, starting with the one-element set. Here, the number of iterations and item sets to be checked is limited by the *apriori principle*. The principle states that an item set whose subsets do not occur frequently cannot be a frequent item set.

The recursive implementation (see Listing 4) is based on an array representation for frequent item sets, which are iteratively extended with recursive SQL. Here, arrays are used as sets and expanded by one element in each iteration. Then the support of each item set is counted. For this purpose, each item set is compared with each shopping cart using the set operator `tuple <@ shopping cart` (line 13).

```
1  with recursive transactions (tid, bucket) as (
2  --one array per shopping cart
3   select tid, array_agg(item) from aprioriscript.sales group by tid
4  -- frequent item sets of size 1
5  ),sales_supp as (select item from aprioriscript.sales group by
       item having count(*)>=10
6  ),frequentitemsets as ( -- frequent item sets with support >= 10
7    -- with one element
8   (select distinct array[p.item]::int[] as items from sales_supp p)
9   union all ( -- extend item sets recursively by one element
10    select distinct array_append(t.items,p.item::int)::int[]
11    from frequentitemsets t, sales_supp p
12    where 10 <= ( -- count support
13       select count(*) from transactions t2
14       where array_append(t.items,p.item::int)::int[] <@ ( t2.
            bucket )
15    ) and t.items[(select count(*) from unnest(t.items))]<p.item
16 ))
17 select * from frequentitemsets;
```

Listing 4: Determining the frequent item sets for the Apriori algorithm: a recursively growing relation calculates the frequent item sets, starting with the one-element item sets (each item as an array with one element).

The operator implemented in *HyPer* is based on storing the elements in a prefix tree that grows with each iteration. Special features of the implementation in *HyPer* are the parallelism per iteration step and the handling of duplicates. Thus, the association rules consider the support of identical elements within a shopping cart.

3 EVALUATION

The evaluation compares the performance of procedures created with *HyPerScript* to table operators of HyPer and MADlib, recursive tables in HyPer, PostgreSQL and Umbra, and *PL/pgSQL* procedures (PostgreSQL 12.6 with MADlib 1.17.0 extension). All experiments were measured on a Ubuntu 20.04 LTS machine with six Intel Core i7-3930K CPUs running at 3.20 GHz and 64 GB DDR4 RAM.

For the Apriori algorithm, 100 different items and 1000 shopping carts were synthetically generated. The number of items per shopping cart varied between 0 and 10. For clustering, we generated 10^6 points whose x- and as y-coordinates were equally distributed in the interval $[0, 10^6]$. The PageRank value was computed for 10^5 nodes with the same number of edges. All experiments were repeated three times and the median was taken for the measurements.

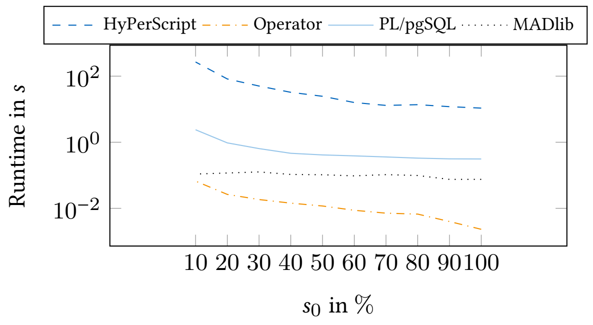
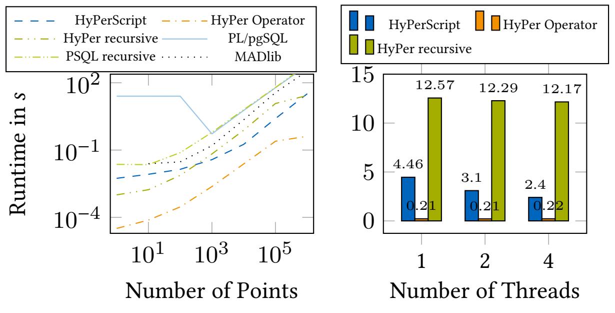


Figure 1: Runtime for association rule analysis with Apriori with twelve threads, depending on the minimum support s_0 as a parameter of Apriori: The larger s_0, the lower the number of frequent item sets and thus the lower the runtime.

For the Apriori algorithm, we varied the minimum support (the larger, the less frequent item sets exist, see Figure 1). With increasing minimum support, the runtime decreases as less frequent item sets exist. Although the HyPer operator performs the best, the implementation in *HyPerScript* is slower than its counterpart in *PL/pgSQL*. This is caused by an implementation of the array set operator within HyPer that unnests the array internally.

The runtimes of the clustering algorithms grow linearly with the input size (see Figure 2a, 3). Although the integrated operators perform the best, the k-Means implementation within *HyPerScript* and using a recursive table (in HyPer, as Umbra's support for window functions was in development at that time) show comparable performance, both outperform the computations in PostgreSQL. The recursive computation of DBSCAN in Umbra [14, 29, 30] (as the other database systems do not support `min` as aggregate function inside a recursive table) was as fast as the implemented operator. The k-Means algorithm in *HyPerScript* computes 30 % faster with each additional core (see Figure 2b), as the underlying database system executes the SQL queries in parallel, whereas the k-Means operator is explicitly parallelised. Neither DBSCAN as an operator nor as a stored procedure support scaling. The SQL queries used in



(a) k-Means: Input size. **(b) k-Means: Scalability.**

Figure 2: Runtime for k-means (five centres, 100 iterations): (a) Runtime depending on the input size with constant twelve threads and (b) depending on the available threads for 10^5 points.

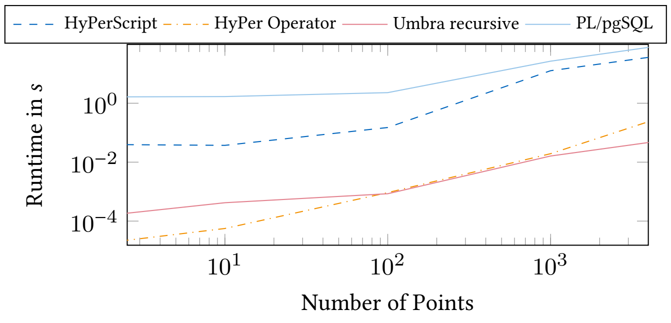
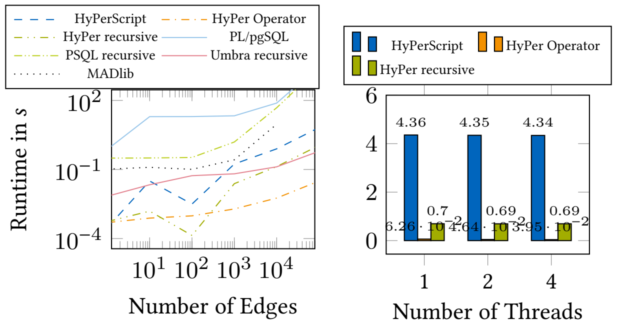


Figure 3: Runtime for DBSCAN with $\epsilon = 20$, `minPts` $= 2$ and 100 iterations: Runtime depending on the input size with constant twelve threads.



(a) PageRank: Number of edges. **(b) PageRank: Scalability.**

Figure 4: Runtime for 100 iterations for calculating the PageRank value with (a) increasing number of edges (twelve threads) or (b) threads (10^5 edges).

the *HyPerScript* procedure scale poorly because only one tuple is initialised as a new cluster per (non-parallelised) iteration.

All implementations of PageRank in HyPer outperform their counterparts in PostgreSQL (see Figure 4). The implementations within a scripting language perform slightly worse than the corresponding query using a recursive table in PostgreSQL and HyPer respectively. With few edges, the additional overhead of the integrated operator in HyPer becomes apparent, since it creates a dictionary for the nodes and stores edges in a sparse matrix as Compressed-Sparse-Row (CSR). With an increasing number of edges, the additional effort for the dictionary and the CSR data structure is amortised, so that the operator calculates the PageRank value faster than the script function.

4 CONCLUSION

This study showed how to express data mining algorithms in SQL-92 only by relying on recursive CTEs. For this reason, we implemented four algorithms, namely k-Means, DBSCAN, Apriori and PageRank in SQL and compared their performance to library functions and UDFs. The evaluation revealed that the recursive implementation performs worse than the operators in HyPer but due to the performance of an in-memory database system similar to MADlib's library functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of data analysis algorithms. Aggregate functions in database systems allow the implementation of further algorithms, for example, in machine learning to average the gradient [26, 26, 28, 28, 33, 36]. Reducing the number of computations per iteration step, for example the ranking for k-Means, and the number of used array expression (Apriori) would accelerate the performance further.

REFERENCES

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. In *SIGMOD Conference*. ACM Press, 207–216.

[2] Andrej Andrejev, Kjell Orsborn, and Tore Risch. 2020. Strategies for array data retrieval from a relational back-end based on access patterns. *Computing* (2020).

[3] Róbert Beck, László Dobos, Tamás Budavári, Alexander S. Szalay, and István Csabai. 2017. Photo-z-SQL: Integrated, flexible photometric redshift computation in a database. *Astron. Comput.* 19 (2017), 34–44.

[4] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117.

[5] Bin Dong, Patrick Kilian, Xiaocan Li, Fan Guo, Suren Byna, and Kesheng Wu. 2019. Terabyte-scale Particle Data Analysis: An ArrayUDF Case Study. In *SSDBM*. ACM, 202–205.

[6] Mehrad Eslami, Yicheng Tu, Hadi Charkhgard, Zichen Xu, and Jiacheng Liu. 2019. PsiDB: A Framework for Batched Query Processing and Optimization. In *IEEE BigData*. IEEE, 6046–6048.

[7] Abir Farouzi, Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. 2020. PandaSQL: Parallel Randomized Triangle Enumeration with SQL Queries. In *CIKM*. ACM, 3377–3380.

[8] Marios Fragkoulis, Diomidis Spinellis, and Panos Louridas. 2015. An interactive SQL relational interface for querying main-memory data structures. *Computing* 97, 12 (2015), 1141–1164.

[9] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB@VLDB*.

[10] Joseph M. Hellerstein et al. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.

[11] Nina C. Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. 2017. HyPerInsight: Data Exploration Deep Inside HyPer. In *CIKM*. ACM, 2467–2470.

[12] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *SIGMOD Conference*. ACM, 681–697.

[13] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS. *Proc. VLDB Endow.* 12, 7 (2019), 822–835.

[14] 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.

[15] Sangchul Kim and Bongki Moon. 2018. Federated database system for scientific data. In *SSDBM*. ACM, 33:1–33:4.

[16] Muideen Lawal, Pierre Genevès, and Nabil Layaïda. 2020. A Cost Estimation Technique for Recursive Relational Algebra. In *CIKM*. ACM, 3297–3300.

[17] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. 2017. A SQL-Middleware Unifying Why and Why-Not Provenance for First-Order Queries. In *ICDE*. IEEE Computer Society, 485–496.

[18] Linnea Passing et al. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*. OpenProceedings.org, 84–95.

[19] Magdalena Pröbstl et al. 2021. One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA. In *ADMS@VLDB*. 17–26.

[20] Astrid Rheinländer, Martin Beckmann, Anja Kunkel, Arvid Heise, Thomas Stoltmann, and Ulf Leser. 2014. Versatile optimization of UDF-heavy data flows with sofa. In *SIGMOD Conference*. ACM, 685–688.

[21] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2021. B^2-Tree: Cache-Friendly String Indexing within B-Trees. In *BTW (LNI, Vol. P-311)*. GI, 39–58.

[22] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2022. B^2-Tree: Page-Based String Indexing in Concurrent Environments. *Datenbank-Spektrum* 22, 1 (2022), 11–22.

[23] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21.

[24] Maximilian E. Schüle et al. 2017. Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. *Proc. VLDB Endow.* 10, 12 (2017), 1921–1924.

[25] 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.

[26] Maximilian E. Schüle et al. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT*. OpenProceedings.org, 562–565.

[27] Maximilian E. Schüle et al. 2020. ARTful Skyline Computation for In-Memory Database Systems. In *ADBIS (Communications in Computer and Information Science, Vol. 1259)*. Springer, 3–12.

[28] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*. ACM, 7:1–7:4.

[29] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2021. ArrayQL for Linear Algebra within Umbra. In *SSDBM*. ACM, 193–196.

[30] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2022. ArrayQL Integration into Code-Generating Database Systems. In *EDBT*. OpenProceedings.org, 1–12.

[31] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM, 6:1–6:12.

[32] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *SSDBM*. ACM, 169–180.

[33] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *SSDBM*. ACM, 25–36.

[34] Maximilian E. Schüle, Linnea Passing, Alfons Kemper, and Thomas Neumann. 2019. Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme. In *BTW (LNI, Vol. P-289)*. GI, 107–126.

[35] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. 2021. TardisDB: Extending SQL to Support Versioning. In *SIGMOD Conference*. ACM, 2775–2778.

[36] Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, and Thomas Neumann. 2022. LLVM Code Optimisation for Automatic Differentiation. In *DEEM@SIGMOD*. ACM.

[37] Maximilian E. Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. The Power of SQL Lambda Functions. In *EDBT*. OpenProceedings.org, 534–537.

[38] Tarique Siddiqui, Surajit Chaudhuri, and Vivek R. Narasayya. 2021. COMPARE: Accelerating Groupwise Comparison in Relational Databases for Data Analytics. *Proc. VLDB Endow.* 14, 11 (2021), 2419–2431.

[39] Kurt Stockinger, Nils Bundi, Jonas Heitz, and Wolfgang Breymann. 2019. Scalable architecture for Big Data financial analytics: user-defined functions vs. SQL. *J. Big Data* 6 (2019), 46.

[40] Sebastián Villarroya and Peter Baumann. 2020. On the Integration of Machine Learning and Array Databases. In *ICDE*. IEEE, 1786–1789.

[41] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. 2015. Lenses: An On-Demand Approach to ETL. *Proc. VLDB Endow.* 8, 12 (2015), 1578–1589.

[42] Chao Zhang and Farouk Toumani. 2020. Sharing Computations for User-Defined Aggregate Functions. In *EDBT*. 241–252.