Rethinking Relational Operators as Hardware-Accelerated Matrix Operations

Jannis Karampetsos, Mareike Schmidt, Martin Poppinga, Annett Ungethüm
Department of Informatics
Universität Hamburg
Hamburg, Germany
{jannis.karampetsos@studium., mareike.schmidt-3@, martin.poppinga@, annett.ungethuem@}uni-hamburg.de

***Abstract*—The Single Instruction Multiple Data (SIMD) paradigm has become a popular part of the optimization of query processing in state-of-the-art column-stores. Recent additions to the SIMD instruction sets offer matrix operations, thereby going beyond the vector operations offered by previous instruction sets. A main feature which is available in all of these instruction sets is the matrix multiplication. Although algebra of matrices is not commonly used in analytical query processing of relational data, there is a set of well-defined relational operators relying heavily on matrix multiplication. This definition uses relations represented as associative arrays which can serve as a common representation for multiple formats, e.g., graphs and (key, value)-pairs. In this work, we realize a selection of these operator definitions using Intel's most recent instruction sets for vector and matrix processing (AVX512 and AMX). For this purpose, we identify the code sections where AMX resp. AVX512 is useful, provide a short outline of our implementation, and create test datasets with different properties. Then, we elaborate on the contribution of each code section to the execution time of complete operators, and examine the benefits of hardware-accelerated matrix multiplication for our use-case.**

***Index Terms*—Query Processing, SIMD, Intel AMX, Associative Arrays**

I. Introduction

The Single Instruction Multiple Data (SIMD) paradigm has become a standard technique in the optimization of relational query processing, especially in column-stores [1]–[3]. SIMD processes multiple data elements stored in one SIMD register, e.g., multiple integer values, at the same time using one instruction. This way, data parallelism within a single thread is realized. The success of SIMD instruction sets led to their further development. Not only were new instructions added, but the register size has also grown. For instance, the SSE and Neon instruction sets by Intel resp. ARM offered 128 bit registers. The newer instruction set AVX512 by Intel offers 512 bit and SVE by Arm up to 2048 bit.

The most recent addition to the SIMD instruction sets, make the shift from vectors to matrices. Intel released the Advanced Matrix Extensions (AMX) [4] in 2023 while ARM introduced the Scalable Matrix Extension (SME) with the newest version SME2 [5] being available since the 4th quarter of 2025. However, the available function range of these extensions is quite small. For instance, none of the instruction sets offers row-wise or column-wise comparisons. The only feature, apart from load and store instructions, that is available in all of these instruction sets is matrix multiplication. However, this is not an intuitive choice for analytical operators that process individual columns, i.e., vectors.

Fortunately, there is a set of definitions of relational operators which heavily relies on matrix multiplications and was published in the context of HPC [6]. These operators are defined on relations represented as associative arrays, a flexible structure which can also be used to store other data formats, such as graphs. This makes it predestined as a common representation. A system which is able to translate between associative arrays and relations already exists [7], [8], but it does not implement a relational execution engine. Instead, it implements connectors to use existing database systems as storage engines and simple filters. In this work, we present an initial implementation of a subset of the relational operators defined on associative arrays and optimize them using the AMX instruction set wherever applicable. We also use the AVX512 instruction set where AMX does not provide a suitable feature.

In detail, our contributions are the following:

- In Section II, we give a short introduction into the representation of relational data as associative arrays, the corresponding definition of selected relational operators using algebra of matrices, and the AMX instruction set.
- Then, we examine the applicability of AMX for the different relational operators and propose an implementation for the promising operators in Section III.
- We evaluate our solution in Section IV.
- Finally, we reflect on related work and possible future research directions, and conclude our work in Sections V and VI.

II. Preliminaries

Associative arrays may serve as a common representation between different (No)SQL formats [8]. However, the only current implementation mainly focuses on the abstraction layer while the connected database systems are mainly used as storage engines, e.g. D4M [8]. A user must define their analytics in terms of associative array algebra instead of relational operators. Further, the well-known GraphBLAS API-specification[1] has various implementations and was spawned

[1]https://graphblas.org/, accessed: 26/01/2026

Students

	S_Course	S_Name
D001	Computer Science	Jane Doe
D002	Biology	John Doe
J001	Biology	Jan Janssen

Exams

	E_Course	E_Name
E0a1	Biology	John Doe
E0a2	Computer Science	Max Mustermann

Fig. 1: Two examples for associative arrays: *Students* and *Exams*. *Students* has the row keys $\{D001, D002, J001\}$ and the column keys $\{S\_Course, S\_Exam\}$. *Exams* has the row keys $\{E0a1, E0a2\}$ and the column keys $\{E\_Course, E\_Name\}$.

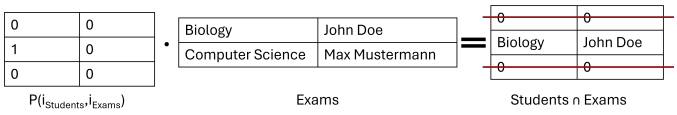

Fig. 2: The intersection operator using associative array algebra. P is the equivalence permutation matrix of *Students* and *Exams*.

by the work on associative arrays [6]. Although GraphBLAS is comprehensive, it focuses on graph algorithms rather than relational operators, for which no dedicated implementation exists. We expect the main challenge of the implementation of relational operators to be the lack of performance of associative array algebra (AAA) compared to existing database operators, which have been optimized over decades. However, the ability to fuse not only the data representation but also the execution engine for different formats remains an attractive goal. It can eliminate costly data transformations and reduce execution time in complex workloads involving heterogeneous data stored as associative arrays. Therefore, it appears appropriate to optimize the relational operators in AAA to a degree where they offer acceptable performance. One strategy for optimizing operators is the use of SIMD instruction sets which are offered by all recent x86 CPUs. One of the most recent of these instruction sets is Intel's AMX. Thus, the goal of this work is to examine the potential performance benefits of the AMX instruction set for relational operators defined in associative array algebra.

A. The Relational Model represented as Associative Arrays and Algebra of Matrices

Associative arrays, as presented by Gadepally and Kepner et al., are a flexible structure which can be used to define graphs, relations, key-value-pairs, and other NoSQL formats [6], [7]. They can be regarded as an extended form of matrix. The primary difference from a matrix is that the row and column indices of an associative array are not sequential integers but may be of any type for which a strict and total order exists, such as strings. This also means that an associative array is as large as the value range of the indices allows. However, only rows and columns which have entries are considered to be a part of the dataset. The second difference is that neither empty rows nor empty columns are stored. Finally, there is no inherent constraint on the type of the values, unless such a constraint is imposed for mapping to another representation. This way, an associative array can be used to represent an adjacency matrix with properties and labels as well as a relation.

Relations: For the representation of a relation, an associative array acts as a table where the column indices are the names of the attributes, while the row indices can either be retrieved by a counter or defined completely freely as long as the chosen indices have a unique total order. Since empty columns and rows are not stored in associative arrays, but can exist in a relational database, the dimensions of an associative array change as soon as a value is inserted into a previously empty column or row. In contrast to this, the logical dimension of the associative array does not change as it includes all possible values of the indices. Two examples for associative arrays are shown in Figure 1, which demonstrates that the mapping between relations and associative arrays is mostly intuitive. For the sake of simplicity and readability, we refer to relations in their associative array representation as relation for the remainder of this paper.

Operators: In associative array algebra, relational operators are expressed based on typical matrix operations as known from linear algebra. Only matrix multiplication, element-wise addition, and element-wise multiplication are used. A key component of most operators is the permutation matrix P. This permutation matrix serves as a selection tool for rows and is computed differently depending on the operator it is used in. Except for *project* and *rename*, each basic operator includes at least the following two steps: First, P is computed, then it is multiplied with a relation represented as associative array. Hence, P acts as a filter mask.

For set operations, e.g. intersection, P is defined as an equivalency permutation matrix, i.e., it represents the equality of non-empty rows in a relation A with another relation B and can be defined in multiple ways. One definition uses the Kronecker-Delta:

$$\delta(i,j) = \begin{cases} 0 & if \quad i \neq j \\ 1 & if \quad i = j \end{cases}$$

Using the Kronecker-Delta, P can be defined as follows:

$$P(i_A, i_B) = \delta(A(i_A, :), B(i_B, :))$$

Using this equivalence permutation matrix, the intersection is defined as $A \cap B = PB$. Figure 2 illustrates the intersection between the two example arrays introduced above: $Students \cap Exams$. First, P is computed using the Kronecker-Delta, then it is multiplied with $Exams$. The result contains all rows in $Exams$ which have a matching row in $Students$. The zeroed rows in the result equal empty rows and are not considered a part of the result set. The set difference operator uses the result of the intersection and removes it from the left relation (see Table 1). In the remainder of this paper, we will refer to this last step of the difference as *subtraction*. A challenge when using associative arrays is the distinction between 0 as an explicit value and 0 representing an empty value. Whenever

Operator	Relational	SQL	Associative Array
Intersection	$A \cap B$	`SELECT * FROM A INTERSECT SELECT * FROM B`	PB
Set Difference	$A \setminus B = A - (A \cap B)$	`SELECT * FROM A EXCEPT SELECT * FROM B`	$A \oplus -PB$
Select	$\sigma_{s(J)}(R)$	`SELECT * FROM A WHERE [s(J)]`	PA where $P = \mathbb{I}(s(A(:, J))$
Project	$\pi_J(A)$	`SELECT J FROM A`	$A(:, J) = A\mathbb{I}(J)$

Tab. 1: Selected relational operators with their SQL equivalents and their associative array algebra equivalents as defined by Kepner et al. [6], where $-$ denotes to the additive inverse: $v \oplus -v = 0$. J is a subset of the columns in a relation and $\mathbb{I}$ is the identity array. Note that there are usually multiple ways to define the same operator in associative arrays algebra.

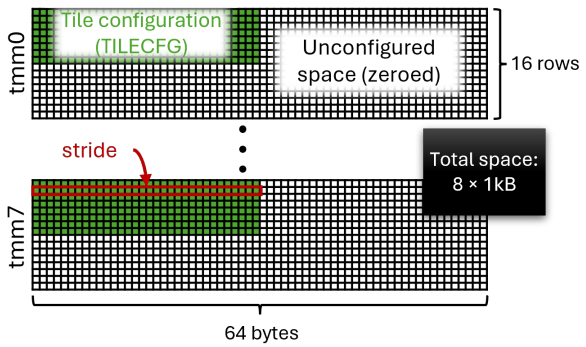

Fig. 3: There are 8 *tile* registers, *tmm0* to *tmm7*. Each of them can contain up to 16 rows with a length of 64 Byte. The space used in these registers must be defined in a *tile* configuration which is applied to all *tiles*. Unused space is zeroed. Loads and stores are strided accesses.

this distinction is necessary, it must be explicitly annotated which 0 is meant in each case, e.g. by using different data types or a secondary structure storing this information.

Table 1 shows a selection of relational operators as described by Kepner et al. [6], their SQL equivalent, and an equivalent in associative array algebra.

For instance, assume a selection $\sigma_{s(J)}(R)$ where J is a subset of columns in R. For defining P, the selection predicate is evaluated in the respective columns J of a relation. The identity matrix of the non-zero rows and columns of this result is the permutation matrix which is then multiplied with the relation R. The result contains all rows of R which match the selection predicate $s(J)$.

B. From Vectors to Tiles: A SIMD-Instruction Set for Matrix Operations

Vectorization has become a common optimization technique because all recent x86 CPUs offer at least one SIMD instruction set. Until Intel released the first CPUs equipped with the AMX instruction set [4] in 2023, SIMD registers were one-dimensional vector registers. By that time, the most recent SIMD instruction sets held up to 512 bit (Intel AVX512) resp. 2048 bit (ARM SVE). The range of functions of these instruction sets grew with each iteration, such that AVX512 even had multiple subsets. In contrast to this development, the AMX instruction set diverts from a growing vector size and function range in favor of a two-dimensional register called the *tile* register and a small but extensible range of functions.

The tile register: Figure 3 provides an overview of the tiles offered by AMX. There are 8 *tile* registers per CPU, denoted as tmm0..tmm7. Each of them is 64 byte wide, the same width as an AVX512 register, and 16 rows high. Thus, one tile can hold up to 1 kB of data. The extent to which this space can be programmed must be defined by choosing from a *palette* of configuration options. Currently, this *palette* is rather small with 2 options: 0 (initialized, tile is zeroed) and 1 (each tile is fully usable). Additionally, a developer must specify the dimensions they will use in a *tile configuration*. This can be smaller than the maximum 1 kB per tile. The configuration is stored in a dedicated 64 bit register called TILECFG. All load and store operations from and to the tiles are strided accesses. The stride size does not have to match the *tile configuration* but can also be smaller[2]. This is useful when a matrix smaller than the *tile configuration* is processed.

Functions: Similar to previous SIMD instruction sets, data must be loaded explicitly into the registers and written back to main memory. For this purpose, there are instructions for loading and storing data conventionally and without using the cache. There is also an instruction to zero a *tile*. Like the *tiles*, the *tile configuration* must also be loaded into the TILECFG register using a dedicated instruction. Once a valid *tile configuration* is loaded and there is data in the tiles, an accelerator can perform operations on them. AMX is designed to be extensible, allowing additional accelerators to be added in future CPUs [4]. However, currently available CPUs offer only one accelerator: TMUL. TMUL consists of a grid of fused multiply-add units enabling fast matrix multiplication of the contents of two tiles. The result is stored in a third tile.

III. AMX-Accelerated Operators

As mentioned in Section II, the crucial parts for most relational operators in AAA are the construction of the permutation matrix P and the multiplication of P with a relation. In this Section, we describe our approach of an implementation using the AMX instruction set. Since current AMX implementations only offer the TMUL accelerator for multiplication, only operators containing a multiplication can benefit from using the instruction set. Thus, we decided to use the intersection and the set difference as examples in this work to examine whether AMX is beneficial for the operators' performance. The intersection is a basic relational operator which requires one multiplication of P with the right relation.

[2]https://www.intel.com/content/www/us/en/developer/articles/code-sample/advanced-matrix-extensions-intrinsics-functions.html, accessed 18/01/2026

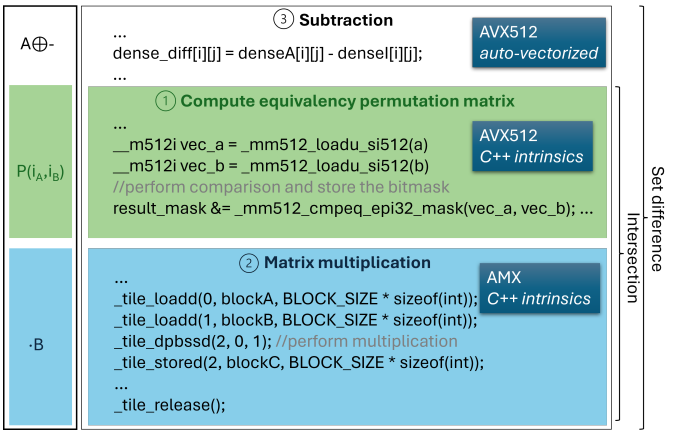

Fig. 4: The most important sections of our implementation of the intersection and set difference operators. Not shown are the surrounding loops, the definitions of auxiliary variables, the function call of the intersection, and the tile configuration. Depending on the operation performed, we chose between the two SIMD instruction sets AVX512 and AMX and additionally relied on auto-vectorization where this was safe to do.

The set difference is a derived operator using the additive inverse of the intersection result $-PB$ (see Table 1).

Implementation: Our implementation of the intersection and set difference requires three operations:

① A comparison resulting in a 2-dimensional bitmask (the equivalence permutation matrix P)
② A matrix multiplication
③ A subtraction of rows

The first and the second operations form the intersection. The set difference additionally requires the third operation. For the comparison in step ①, there is no AMX accelerator. Therefore, we use AVX512 and work on vector registers comparing 512 bit per instruction. This is done using the trivial approach which iterates over two nested loops, one for each relation. Since we do not require our input data to be sorted, this step cannot take advantage of common faster alternatives, e.g. a merge intersection. For step ②, we use the AMX accelerator for matrix multiplication. Step ③ is a simple nested loop that either subtracts (zeroes) an entire row or performs no subtraction, leaving the row unchanged. The result is materialized. In this case, where vectorization is trivial, we rely on the auto-vectorization of our compiler. Figure 4 provides an overview of the three steps and shows the most crucial parts of the source code. In the cases where we do not rely on auto-vectorization, the C++ intrinsics are used instead of writing assembly.

As shown in the code, data must explicitly be loaded into the *vector* (__m512i) and *tile* registers. While the *vectors* are parameterized, the *tiles* are enumerated and can be used by providing the according number. Additionally, unlike Intel's older SIMD instruction set SSE, AVX512 and AMX do not consider any non-numeric data types. Hence, we assume our data is already dictionary-encoded, such that intrinsics working on integers, i.e., *_mm512_cmpeq_epi32_mask* and *_tile_dpbssd*, can be applied.

The intrinsic *_mm512_cmpeq_epi32_mask* translates to the AVX instruction set and is used in step ①. It compares 32-bit integers stored in two 512-bit vector registers, which contain the rows, for equality. The result is stored in a bitmask. If this bitmask is fully populated with 1s, the corresponding rows are equal. For larger rows, this step is repeated.

The intrinsic *_tile_dpbssd* translates to the AMX instruction set and is used in step ②. It computes the dot product of bytes in tiles. The result is stored as 32-bit integers. The tiles are addressed using their index, e.g., in Fig.4 the dot product of tile 0 and 1 is computed and the result is stored in tile 2.

Limitations of AMX: Since the size of the tiles is limited and main memory accesses must be triggered explicitly using a corresponding intrinsic, it becomes necessary to handle the multiplication of larger matrices. We achieve this using the classical tiled matrix multiplication implemented by three nested loops. Another limitation of AMX is that there is no instruction to delete individual rows or columns. As a result, rows which are zeroed and therefore not a part of the result set, cannot be pruned before storing the result.

IV. Evaluation

For this evaluation, all implementations are done with C++. We compiled them with the oneAPI DPC++/C++ Compiler 2025.3.1 using the flags -O3 and -march=sapphirerapids. Then, the experiments were conducted on an c7i.metal-24xl node in the Amazon EC2 cloud. The node is equipped with an Intel Xeon Platinum 8480C and allows to run a non-virtualized environment without sharing its resources. Since the goal of SIMD is to increase single-thread performance, all experiments are executed single-threaded and entirely in-memory. In the remainder of this section, we will compare our implementation to another implementation relying on auto-vectorization for the matrix multiplication instead of explicitly using AMX. The creation of the equivalency permutation matrix is done using AVX512 in both cases. Our two relations have the same size in all test cases, resulting in a quadratic equivalency permutation matrix.

A. Operator breakdown

Figure 5 shows the breakdown of the *set difference* operator into its three parts described in Section III. In this example, our input sets are identical. Figure 5 a) shows the execution time without the use of AMX while 5 b) shows the execution time when using AMX. In both cases, the execution time increases as expected as the size of the permutation matrix grows. The execution time of the auto-vectorized subtraction ③ is negligible in all experiments, while the first two steps define the performance of the operator. However, our two implementations differ in the contribution of these two steps to the overall execution time. Without AMX, matrix multiplication accounts for nearly 80% of the operator's total execution time, and this proportion increases with input size. In contrast, in our AMX implementation the fraction never exceeds roughly

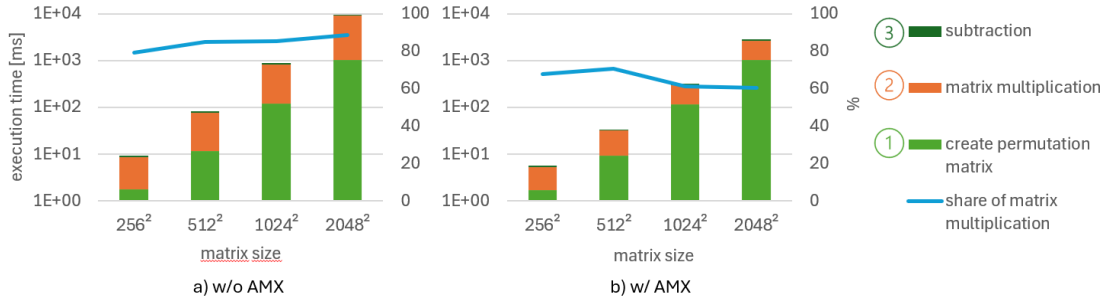

Fig. 5: A breakdown of the *set difference* operator for different dataset sizes. While the time required for the subtraction, i.e. step (3), is negligible, steps (1) and (2) define the performance. **a)** Without the use of AMX, the share of the matrix multiplication increases with the size of the dataset. **b)** Using AMX, the share of the matrix multiplication is generally lower than without, and it decreases while the dataset grows.

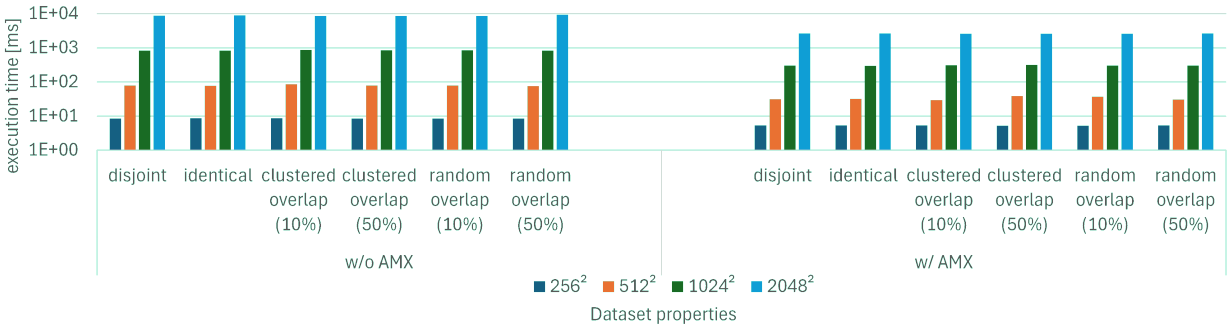

Fig. 6: The execution time of the *set difference* operator for differently overlapping datasets and different sizes of the datasets. The left side shows the execution time without the use of AMX. The right side shows the execution time with the use of AMX. Note that the vertical axis is in log scale.

70%, and it decreases as the matrix size grows. Most of the remaining execution time is used for the creation of the permutation matrix. Furthermore, the overall execution time is lower when using AMX, which was expected.

B. *Influence of data set properties*

In our previous example, the two datasets were identical. In our first experiment, we examine the influence of the dataset properties, i.e. how much they overlap and how this overlap is clustered or scattered. Figure 6 shows the execution time of the intersection operator for different datasets. Additionally to the size, their intersection also varies. We conducted the experiment not only on the identical sets, but also with completely disjoint datasets and datasets where only a fraction is overlapping. For the overlap, we created four different versions. In the first two versions, the overlap is clustered, i.e., one large continuous part of each dataset belongs to the intersection. In the other version, the overlapping rows are randomly scattered across the input sets. The right side of the graph shows the execution times using AMX, the left side shows them without using AMX. As expected, our AMX approach performs better for every dataset. However, the lack of a difference in the execution time for different intersection properties is less intuitive. We assume that the reason for this behavior is that zero rows are still stored in

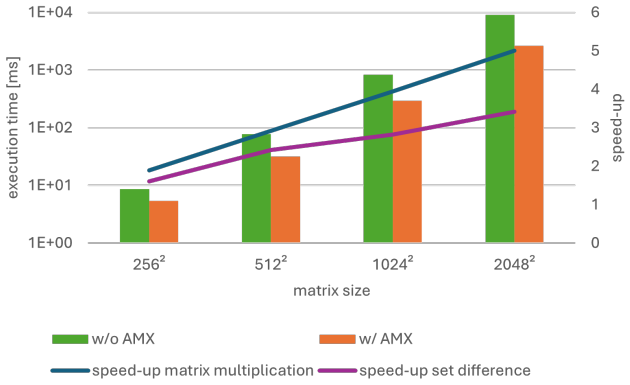

Fig. 7: The execution time for the approach with AMX and without AMX. Additionally, the speed-up while using AMX is shown for the whole operator and the matrix multiplication only.

both implementations, i.e., the permutation matrix P from step (1) and the result of step (2) always return a matrix of the same size which differs only in the sparsity and distribution of its population. However, these results suggest that the observations presented in Figure 5 extend to intersections of non-identical datasets.

C. Influence of AMX

Figure 7 shows the execution time of the *set difference* operator for both our AMX and non-AMX implementations. In addition, the speed-up when using AMX is pictured for the entire operator and for the matrix multiplication (step (2)) only. While the speed-up of the multiplication grows proportionally with the data set, this does not hold for the complete operator. Since the contribution of the matrix multiplication to the overall execution time decreases, the speed-up caused by the use of AMX instructions also decreases. In all our experiments, AMX led to a speed-up between 1.6x and 3.4x for the whole operator compared with the implementation that uses only AVX512 and auto-vectorization. Step (2) of our operator even shows a speed-up of up to 5x. This speed-up can be observed for different data set properties when comparing the diagrams on the left side of Fig. 6(without AMX) with the diagrams on the right side (with AMX).

V. Related Work

Previous work using SIMD instruction sets to optimize relational operators focuses on vector processing as this is supported by all recent x86/x64 CPUs. Individual operators have been optimized numerous times, e.g., radix sort [9] and set intersection [10]. Additionally, specialized vector processors with an additional scalar processing unit are available and have been tested for their applicability in relational operators [11]. Besides the optimization of individual operators, there is the processing engine *VIP* which is built bottom up from reusable sub-operators and heavily relies on AVX512 [3].

There are also approaches using the tensor computation frameworks, which became popular with the rise of AI, to increase the performance of relational queries, e.g., *TQP* [12]. However, these approaches rely on modern GPUs and a reliably fast interconnect to the host system to outperform the CPU performance. In contrast, our approach only requires a modern CPU with the AMX instruction set. Historically, the intel CPU instruction sets have also been implemented in the x86/x64 CPUs of other brands after they have proven to be successful. Thus, we expect our approach to be more conveniently adaptable for real-world systems in the future. Moreover, while similar on first sight, our used data model is not relational.

Other work goes into the direction of generalization, e.g., a common API for different SIMD instruction sets [13] or the treatment of other architectures such as FPGAs or GPUs as vector processors [14], [15]. However, the AMX instruction set works on matrices instead of vectors. To the best of our knowledge, it has primarily been used in the context of LLMs where fast matrix multiplication is a key factor for performance [16], [17]. Our approach considers the relational operators defined in associative array algebra, which uses matrix multiplications as a core component. This way, we achieve two goals: 1) Processing relations stored as associative arrays without converting them back into relations, and 2) Using the AMX instruction set to increase the performance of set operators.

VI. Conclusion and Future Work

The most recent Intel instruction set, AMX, offers native CPU-instructions for matrices. Currently, only matrix multiplications are available. We could show that this feature increases the operator performance for set operations when directly processing relations as associative arrays. Further, AMX is meant to be extensible. Especially useful for our work would be future accelerators for element-wise comparisons, transposition, a direct data transfer between AMX and AVX512-registers, and the deletion of individual rows and columns. A transpose accelerator was previously announced for the models released in 2026 but then canceled. Thus, it is not certain if existing limitations might relax in the future.

Future Research Directions: Currently, we use a naive approach for computing the equivalency permutation matrix P. Its optimization is a necessary next step, which should be straight-forward considering the numerous existing optimizations for set intersections. However, this might be outdated as soon an appropriate accelerator for element-wise or row-wise comparisons is available. A look into the most recent version of ARMs Scalable Matrix Extension (SME2) is also promising as it adds new features like vector extraction and transposition. CPUs equipped with SME2 have been released recently.

In order to create a fully running execution engine, the implementation of the remaining operators is also on our agenda. This first attempt using set operations provided valuable insights to achieve this goal.

Beyond this, a more complex challenge is query optimization using associative array algebra. While building P is computationally expensive, it is a sparsely populated matrix which can be compressed. This allows to keep the matrix as a part of the databases meta data and reuse it in other operators or even in other queries which use the same associative arrays. Moreover, when the base data changes, only a fraction of P must be changed accordingly.

References

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[2] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Morphstore: Analytical query engine with a holistic compression-enabled processing model," *Proceedings of the VLDB Endowment*, vol. 13, no. 11.

[3] O. Polychroniou and K. A. Ross, "Vip: A simd vectorized analytical query engine," *The VLDB Journal*, vol. 29, no. 6, pp. 1243–1261, 2020.

[4] *Intel® Architecture Instruction Set Extensions and Future Features, Programming reference*, March 2025. Document number 319433-057.

[5] *Arm C1-Scalable Matrix Extension 2 Technical Reference Manual*, 2026. https://developer.arm.com/documentation/107831/0103, accessed: 26.01.2026.

[6] J. Kepner, V. Gadepally, D. Hutchison, H. Jananthan, T. Mattson, S. Samsi, and A. Reuther, "Associative array model of sql, nosql, and newsql databases," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2016.

[7] V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, *et al.*, "D4m: Bringing associative arrays to database engines," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2015.

[8] L. Milechin, V. Gadepally, S. Samsi, J. Kepner, A. Chen, and D. Hutchison, "D4m 3.0: Extended database and language capabilities," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2017.

[9] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 755–766, 2014.

[10] J. Zhang, Y. Lu, D. G. Spampinato, and F. Franchetti, "Fesia: A fast and simd-efficient set intersection approach on modern cpus," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1465–1476, IEEE, 2020.

[11] A. Ungethüm, L. Schmidt, J. Pietrzyk, D. Habich, and W. Lehner, "Mastering the nec vector engine accelerator for analytical query processing," in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pp. 60–65, 2021.

[12] D. He, S. C. Nakandala, D. Banda, R. Sen, K. Saur, K. Park, C. Curino, J. Camacho-Rodríguez, K. Karanasos, and M. Interlandi, "Query processing on tensor computation runtimes," in *Proc. VLDB Endow.*, 2022.

[13] A. Ungethüm, J. Pietrzyk, P. Damme, A. Krause, D. Habich, W. Lehner, and E. Focht, "Hardware-oblivious SIMD parallelism for in-memory column-stores," in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, 2020.

[14] J. Pietrzyk, A. Krause, C. Faerber, D. Habich, and W. Lehner, "Program your (custom) SIMD instruction set on FPGA in C++," in *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*, 2024.

[15] J. Fett, A. Ungethüm, D. Habich, and W. Lehner, "The case for simdified analytical query processing on gpus," in *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China* (D. Porobic and S. Blanas, eds.), pp. 14:1–14:5, ACM, 2021.

[16] H. Kim, G. Ye, N. Wang, A. Yazdanbakhsh, and N. S. Kim, "Exploiting intel advanced matrix extensions (amx) for large language model inference," *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 117–120, 2024.

[17] M. Kim, H. Ji, J. Kang, H. Lee, D. Kim, and N. S. Kim, "Cabana: Cluster-aware query batching for accelerating billion-scale anns with intel amx," *IEEE Computer Architecture Letters*, 2025.