Accelerating machine learning queries with linear algebra query processing

Sun, Wenbo; Katsifodimos, Asterios; Hai, Rihan

DOI
10.1007/s10619-024-07451-7

Publication date
2025

Document Version
Final published version

Published in
Distributed and Parallel Databases

Distributed and Parallel Databases (2025) 43:8
https://doi.org/10.1007/s10619-024-07451-7

Accelerating machine learning queries with linear algebra query processing

Wenbo Sun[1] · **Asterios Katsifodimos**[1] · **Rihan Hai**[1]

Accepted: 3 December 2024

Abstract

The rapid growth of large-scale machine learning (ML) models has led numerous commercial companies to utilize ML models for generating predictive results to help business decision-making. As two primary components in traditional predictive pipelines, data processing, and model predictions often operate in separate execution environments, leading to redundant engineering and computations. Additionally, the diverging mathematical foundations of data processing and machine learning hinder cross-optimizations by combining these two components, thereby overlooking potential opportunities to expedite predictive pipelines. In this paper, we propose an operator fusion method based on GPU-accelerated linear algebraic evaluation of relational queries. Our method leverages linear algebra computation properties to merge operators in machine learning predictions and data processing, significantly accelerating predictive pipelines by up to 317x. We perform a complexity analysis to deliver quantitative insights into the advantages of operator fusion, considering various data and model dimensions. Furthermore, we extensively evaluate linear algebra query processing and operator fusion utilizing the widely-used Star Schema and TPC-DI benchmarks. Through comprehensive evaluations, we demonstrate the effectiveness and potential of our approach in improving the efficiency of data processing and machine learning workloads on modern hardware.

Keywords Database · Query optimization · Machine learning · Operator fusion

✉ Wenbo Sun
w.sun-2@tudelft.nl

Asterios Katsifodimos
a.katsifodimos@tudelft.nl

Rihan Hai
r.hai@tudelft.nl

[1] Faculty of EEMCS, TU Delft, 2628ZE Delft, The Netherlands

Published online: 13 January 2025

1 Introduction

In recent years we are witnessing unprecedented growth in large-scale ML applications fueled by rapid advancements in computational capabilities, sophisticated models, and the increasing availability of vast amounts of data. Enterprises are now utilizing predictive results to assist in business decision-making and product design for customers. For instance, banks employ ML models for credit scoring and fraud detection, while online retailers use customers' historical behavior to provide real-time recommendations. In this thriving context, predictive ML applications call for efficient computation to meet the growing demands for real-time ML predictions and the substantial data processing workload required by ML models.

Pitfalls of separating data processing and ML predictions Plenty of research efforts and commercial products have provided various solutions to accelerate data processing [3, 12] and ML predictions [4, 23] using modern hardware like Graphics Processing Units (GPU). Thanks to massive parallelism and hardware architectures designed for linear algebra (LA) computations, the throughput of data processing and model predictive pipelines has significantly improved. However, the mixture of relational operators in data processing pipelines and Linear Algebraic operators in ML models introduces diverse data structures and software stacks. Specifically, data processing typically involves tasks such as data transformation and aggregation, which are traditionally solved with relational query engines. In contrast, model prediction workloads involve vast linear algebraic operations. The distinct mathematical foundations of data processing and model predictions often result in using separate software tools, libraries, and hardware configurations, which can hinder overall performance and efficiency. *This separation can increase complexity, higher development and maintenance costs, and potential performance bottlenecks.*

Mathematical gap of RA and LA. The different mathematical foundations present challenges for cross-optimizations when merging relational and linear algebra. Relational operators primarily process input data as sets of tuples, while LA computations operate on ordered scalars, vectors, and matrices. The data transformation and I/O cost between these two algebra systems result in significant overhead. Additionally, the diverging algebra systems imply different logical optimization strategies. Specifically, relational algebra (RA), a specification of first-order logic, can utilize logical reduction to decrease computational complexity. In contrast, LA operators can often take advantage of the numerical information of input matrices to reduce the size of intermediate results and overall complexity. *In short, the foundational differences between LA and RA obstruct further optimization by combining these two systems at both the logical and implementation levels.*

RA operators on top of LA with GPU acceleration A unified data representation and operators are desirable for ML practitioners. One promising new approach to address the challenges associated with the inconsistencies between LA and RA is to *process relational data queries using linear algebra* operations,

such as matrix-matrix and matrix–vector multiplication. We term this method as Linear Algebra Queries (LAQ). By reformulating RA operations as LA operations, this approach can help bridge the gap between data processing and ML domains. Matrix multiplication is a linear algebra operation that can be efficiently parallelized and optimized using modern hardware, such as GPUs, which are designed to handle large-scale LA computations. By translating relational data queries into matrix multiplication operations, this approach can take advantage of the inherent parallelism and computational power of GPUs, leading to significant improvements in efficiency and scalability for both data processing and ML predictions.

Some operators (e.g., join, aggregation) have already been implemented and evaluated in recent studies [1, 6, 13, 14, 27]. However, these studies do not compare their performance with full-fledged GPU databases, nor do they incorporate their methods into predictive pipelines involving machine learning predictions. As a result, *the potential performance gains and practical implications of their methods in the end-to-end data processing and ML predictive pipelines remain unclear.*

In this work, we propose a new approach to optimize performance of ML prediction following relational queries. The new approach leverages the unified representation in LAQ and fundamental properties of LA computations. The contributions of this work can be summarized as follows:

- We integrate batch model predictions into LAQ through operator fusion. By leveraging the computation properties of LA (i.e., associativity), we push down linear operators in ML models to dimension tables in a star schema [16] and merge operators in LAQ and models before prediction. Our operator fusion method achieves up to 317x speedups when evaluated on synthetic star schemas, as shown in Fig. 1, compared to the separate execution of queries and predictions.
- We present a complexity analysis for operator fusion in the context of star schema queries followed by model predictions. This analysis provides quantitative insights into the benefits that can be gained from operator fusion, given specific data and model sizes.

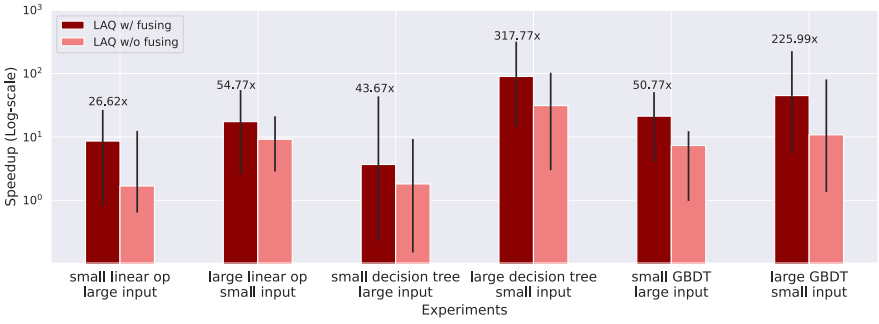


Fig. 1 Speedups of our operator fusion method in four experimental predictive pipelines. The baseline is cuDF without operator fusion. The maximum attainable speedup is 317.77×

- We thoroughly evaluate LAQ and operator fusion using the widely-adopted SSB benchmark [21] and TPC-DI [24] benchmark. We report the performance comparison with cuDF [26] and HeavyDB [12], two popular GPU-accelerated data processing systems. Our evaluation helps demonstrate the potential of LAQ and operator fusion in handling traditional data query workloads and its effectiveness in the context of end-to-end predictive pipelines.

This paper is organized as follows: Sect. 2 introduces primary operators in LAQ. These building blocks are based on existing works [10, 13]. Following that, Sect. 3 presents three examples, namely *simple linear operators*, *decision trees*, and *gradient boosting decision trees*, to demonstrate the usefulness of our approach in predictive pipelines, which integrates linear operators in ML models into LAQ based on computation properties of LA. In Sect. 4, we first evaluate the performance of LAQ using the SSB dataset, and then we test the efficiency of our operator fusion method with synthetic datasets and TPC-DI benchmark. In the final section, we provide insights into our research findings through experiments and discuss potential research directions derived from this study.

2 Preliminaries: linear algebra based query processing

This section introduces how relational queries can be evaluated through linear algebra. As the preliminaries to our operator fusion method in Sect. 3, we implement the LAQ based on existing solutions. In particular, Sect. 2.1 and 2.2 elaborates projection and selection operator proposed in our earlier work [10]. Sections 2.3–2.5 introduces MM-Join, group-by aggregation and sorting operators in TCUDB [13] and TQP [11].

For clarity, we refer to the input tables of an RA operator (e.g., projection, join) as *source tables* and the query results after executing the relational algebras as *target tables*. Before we perform LAQ, all input tables are transformed into matrices for subsequent LA operators. Important notations are listed in Table 1.

2.1 Projection

We can effectively address the projection operator using matrix multiplication. Projection entails extracting multiple columns from the source table and obtaining the target table. We compute projection through matrix multiplication by defining a *column mapping matrix* $M \in \{0,1\}^{c \times k}$ [10], where c is the number of columns of the source table and k denotes the number of projected columns. As a preparation step, we add ID numbers to columns in the source and target tables. We define M as follows:

$$M[i,j] = \begin{cases} 1, & \textit{if jth column is the ith column after selection} \\ 0, & \textit{otherwise} \end{cases}$$

Table 1 Important notations in Sects. 2 and 3

Notations	Description
c	#Columns of a table
r	#Rows of a table
i	#Rows of the target table after joining
k	#Columns of the target table after joining
p	#Features of a decision tree
l	#Output shape of models
t	#Trees in GBDT models
$\mathbf{v}$	Feature predicates of a decision tree
$\mathbf{h}$	Values of leaves in a decision tree
M	Schema mapping matrix
I	Row mapping matrix
L	A simple linear operator
F	Feature mapping matrix of decision tree
H	Paths to leaves in a decision tree
T	Target table after joining
R, S, B, C, D	Tables
MAT_R, MAT_S	Mapping matrix between keys and common domain

Within one matrix M, for each projected column, its location in the target table after projection is represented by the vertical index i, while its original location in the source table is denoted by the horizontal index j. Non-zero values within the matrix M indicate column correspondences between the source and target tables. As the source table has been converted to a matrix, column projection can be evaluated by multiplying the source table matrix and M. Figure 2 shows an example of the projection process: given source table $Patient(weight, height, age)$, the projection operator $\pi_{weight,age}(Patient)$ is transformed to the matrix multiplication of source table matrix and column mapping matrix M. M indicates that the columns with indexes of 0 and 2 of a source table are mapped to columns 0 and 1 of the target table.

2.2 Selection

The selection operator produces a subset of tuples based on specific conditions, essentially filtering rows of the source table according to these conditions. We use a binary vector with the same length as the number of rows in the source table matrix to implement the function of selection. For each row, the corresponding entry in the selection vector will be 1 if the row satisfies the selection condition and 0 otherwise. Multiplying the source table matrix with the selection vector (or its transpose, depending on the orientation of the matrices) effectively filters out rows that do not meet the selection criteria. The resulting matrix will contain only the rows that satisfy the selection condition.

Improvement and implementation Matrix–vector multiplication is a costly floating-point operation. Therefore, in our implementation, we use a vectorized predicate

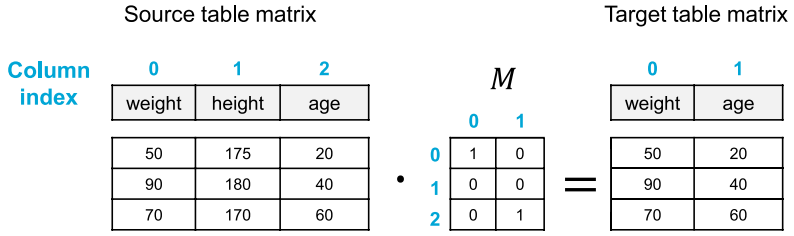


Fig. 2 An example of projection as matrix multiplication

`AND` to compute the indices that is in the target table. When LAQ deals with multiple filtering conditions, it first computes the `AND` among the filtering vectors and applies the final result to the source table, reducing the amount of memory movement when generating the target table. In our implementation, we use the out-of-the-box `mask_select` operator provided by CuPy [20].

2.3 MM-Join

Algorithm 1 Matrix Multiplication Join

1: key_domain $\leftarrow$ union(key_R, key_S) ▷ CUDA reduce
2: key_len $\leftarrow$ len(key_domain)
3: key_dict $\leftarrow$ dict(zip(key_domain, range(0, key_len))
4: $\text{rows}_R \leftarrow \text{range}(0, r_R)$
5: $\text{rows}_S \leftarrow \text{range}(0, r_S)$
6: $\text{columns}_R \leftarrow 0$
7: $\text{values}_R \leftarrow 1$
8: $\text{columns}_S \leftarrow 0$
9: $\text{values}_S \leftarrow 1$
10: **for** $i = 0$ to r_R **do** ▷ CUDA parallelization
11: $\text{columns}_R[i] = \text{key\_dict}[key_R[i]]$
12: **end for**
13: **for** $i = 0$ to r_S **do** ▷ CUDA parallelization
14: $\text{columns}_S[i] = \text{key\_dict}[key_S[i]]$
15: **end for**
16: $\text{MAT}_R \leftarrow \text{cuda\_construct\_CSR}(\text{rows}_R, \text{columns}_R, \text{values}_R)$
17: $\text{MAT}_S \leftarrow \text{cuda\_construct\_CSR}(\text{columns}_S, \text{rows}_S, \text{values}_S)$
18: $I \leftarrow \text{cuda\_sparse\_multiplication}(\text{MAT}_R, \text{MAT}_S).\text{to\_COO}()$
return I

The Matrix Multiplication Join (MM-Join) method takes advantage of matrix multiplication to evaluate join operations, which can be particularly beneficial when working with large datasets or when using hardware optimized for matrix multiplication, such as GPUs. This section introduces the MM-Join implementation in TCUDB [13]. Apart from the implementation details, we discuss the computational

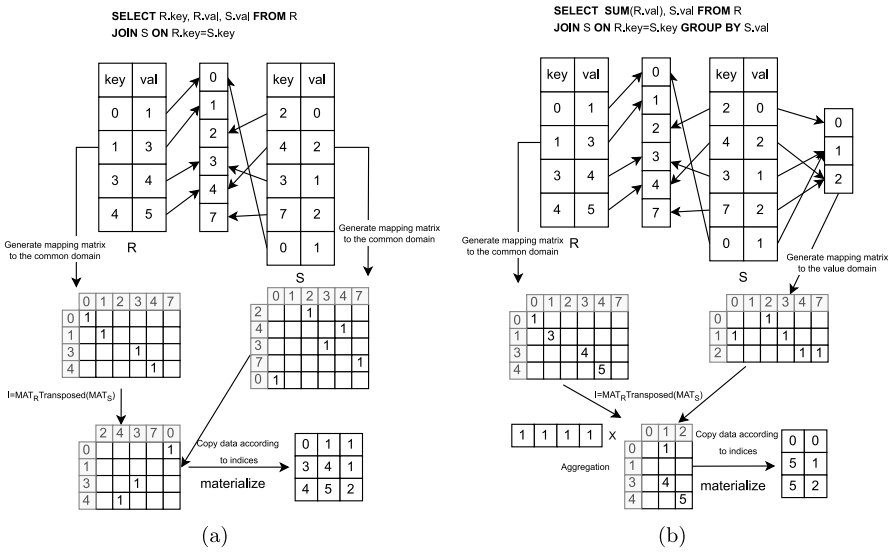


Fig. 3 Illustration of **a** equi-join and **b** group-by-sum with matrix multiplication

complexity of the MM-Join and hash joins. To ensure portability and compatibility with machine learning workloads, we implement this algorithm using CuPy.

2.3.1 2-Way join

The core principle of MM-Join involves mapping keys in two source tables to a common key domain, which is the union of keys in source tables, and representing these mappings as matrices, MAT_S and MAT_R. Subsequently, the key mapping between the two tables can be determined through the multiplication of MAT_S and MAT_R. We illustrate the process of MM-Join with the pseudo-code in Algorithm 1, which has four steps. We explain Algorithm 1 with the running example in Fig. 3.

(1) Suppose R and S are two tables to be joined, we first calculate the union of keys in R and S to construct the common domain (Lines 1–3), resulting in $\{0, 1, 2, 3, 4, 7\}$.
(2) Then we fill non-zero values and positions in sparse matrix format[1] to generate MAT_R and MAT_S (Lines 4–17), which are sparse matrices storing relationships between keys and the common domain. The column indexes of the matrices are identical to the keys' positions in the common domain, and the row indexes are the row numbers of keys in original relations.

[1] We implement the sparse matrices in SciPy CSR: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html.

(3) We execute sparse matrix multiplication over MAT_R and transposed matrix MAT_S (Line 18).
(4) The result I is a row matching matrix,[2] defined as follows.

$$I[i,j] = \begin{cases} 1, & \textit{if ith row of } R \textit{ matches the jth row of } S \\ 0, & \textit{otherwise} \end{cases}$$

The row-column pairs with non-zero values are matched rows in R and S.

The high computational complexity and memory consumption have hindered the application of MM-Joins in CPU-based databases. In Algorithm 1, transforming relations to matrices requires extra time and memory space based on the number of tuples and distinct keys, which is infeasible for relations with a large number of rows.

Approach analysis The domain generation and retrieving process required by constructing sparse matrices involves a set union and two binary search in a sorted array, leading to a computational complexity as $O(n^2 \log n)$. Moreover, even though the CSR format can reduce memory usage, the computational complexity of sparse matrix multiplication (spMM) can not be further reduced: the best-known complexity of spMM is $O(n^2)$ [29],[3] which is higher than $O((|R| + |S|) * log(|R|))$ of a radix hash join algorithm [2], where $|R|$ and $|S|$ represent the cardinalities of the two tables participating in the join.

Nevertheless, MM-Joins present an optimization opportunity that allows for the integration of linear operators in ML models with join processing. Conventionally, the results of relational queries need to be materialized before being utilized in model predictions. However, due to the LA representation of relational join processing, we can leverage LA optimization techniques, such as multiplication re-ordering, to reduce computational complexity and memory usage associated with redundant materialization. This integration can potentially improve the overall efficiency of combining relational operations with models.

2.3.2 Multi-way join

In principle, multi-way joins can be naturally extended from 2-way joins through iterative evaluation following a given order. However, this naive implementation involves the materialization of intermediate tables, overlooking potential optimization opportunities hidden in the selectivity of join operators. In contrast, we can skip the materialization and use the matrix I to evaluate subsequent joins. For instance, let's assume a join order of Q, R, and S. The matching rows of Q and R are stored in matrix I_{QR}. The rows that fail to match Q will not appear in the final result. Therefore, we can directly use the matching row IDs of R to join with

[2] Implemented in SciPy COO format: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html.
[3] The complexity of spMM depends on matrix shapes and sparsity. Here we use an approximate value to show the complexity gap between spMM and hash join.

S and generate the matching matrix I_{RS}. This approach can enhance the performance between R and S due to the potential low selectivity of $Q \bowtie R$.

2.3.3 Materialization

Now we use the row matching matrix I to preserve the matching row IDs of intermediate join results. We could use the IDs to generate a binary vector and treat the materialization as a selection using the method described in Sect. 2.2.

However, this non-LA operation hinders further optimizations by integrating ML models with joins. Alternatively, a materialized table can be viewed as a result of the projection of transposed source tables. In this regard, we need to construct the mapping matrix M using the result matrix I from MM-Join. Consequently, we require two row sparse mapping matrices for relations R and S, as follows:

$$I_*[i,j] = \begin{cases} 1, & \textit{if jth row of R is the ith row of materialized table} \\ 0, & \textit{otherwise} \end{cases}$$

The COO format of I has three attributes: row indexes, column indexes, and the number of non-zero values (nnz). The nnz is precisely the number of rows in the materialized table, which implies that the row IDs of the materialized table can be represented as a vector $m = \langle 0, 1, 2, \ldots, nnz - 1 \rangle$. Consequently, we can construct two row mapping matrices I_R and I_S by aligning row indexes and column indexes with m, respectively.

2.4 Group-by aggregation

In certain analytical queries, we may need to perform aggregation based on specific values after a join operation. Employing a similar technique used for join processing, we can also represent group-by operations concerning a single attribute through matrix multiplication. However, we cannot directly compute multi-aggregation following multi-way joins because it lacks the capability to express value interaction among attributes. In this section, we explain the single-column aggregation in TCUDB [13] and the multi-column aggregation inspired by TQP [11].

2.4.1 Aggregation by a single column

Figure 3b demonstrates how to evaluate single attribute aggregation using LA. The fundamental pattern is similar to the MM-Join, but two sparse matrices require some adjustments. First, MAT_R is no longer a binary matrix; the value column of R to be aggregated is filled into the sparse matrix MAT_R.

As for table S, we begin by finding unique values as groups. MAT_S is filled with values of 1, according to relationships between groups and the key domain. In the example presented in Fig. 3b, values 0, 1, 2 are found as groups. Then we find relationships between keys of S and the groups, which are $\{2\} \rightarrow 0$, $\{3,0\} \rightarrow 1$, $\{4,7\} \rightarrow 2$. After filling 1 s according to the relationships, relationships between keys in R to the group can be evaluated by multiplying MAT_R and ${MAT_S}^T$. Finally, to perform summation of values in R, we introduce a reduction vector filled with values of 1, enabling the materialization of the result.

2.4.2 Aggregation by multiple columns

Aggregation by multiple columns cannot be directly integrated with MM-Join in the same way as single-column aggregation. As shown in Fig. 3b, we require a matrix representing relationships between the key domain and groups. For single-column aggregation, groups can be evaluated using a numerical unique function. However, for multi-column aggregation, we must first join tables involved in groups and then apply the unique function to tuples, which is not consistent with other numerical operators.

To complete the queries for evaluation in our experiments, we adopt an alternative solution proposed in [11], where unique tuples are identified using a sort-unique procedure.

2.5 Sorting

Sorting can not be directly represented with LA operators, but we can integrate sorting into MM-Join if the sorting is performed on keys to be joined. The column indices in matrices MAT_S and MAT_R correspond to the positions of keys in the key domain. As a result, by sorting the key domain, we can obtain MAT_R and MAT_S with sorted keys. This approach allows us to seamlessly integrate the sorting operation into the MM-Join process.

Summary This section discusses existing methods for evaluating relational operators using LAQ and identifies their limitations. Some operators, such as selection, projection, equi-join, and single-column aggregation, can be equivalently represented by linear algebraic computations. However, multi-column aggregation and sorting cannot be transformed into linear algebra operations. To address these limitations, we implement alternative GPU-compatible methods for these two operators, enabling LAQ to evaluate a wider range of relational queries. This allows us to explore the theoretical unification between data processing and downstream ML model predictions on GPUs.

3 Operator fusion

On the basis of LAQ, in this section, we propose an operator fusion method to merge operators in ML model predictions and LAQ for the speedup of predictive pipelines. Specifically, given the fact that operators in LAQ and ML predictions are uniformly represented as linear algebraic computations, we can leverage the computation properties of LA, such as associativity of matrix multiplication, to reduce computational complexity or the size of intermediate results. Moreover, by utilizing the distributive property of matrix multiplication, ML operators can be pushed down to source tables and stored as matrices, subsequently decreasing the computational complexity of real-time predictions.

In this section, based on the LAQ operators introduced in Sect. 2, we demonstrate how does operator fusion perform with three ML models: simple linear operators (Sect. 3.2), decision trees (Sect. 3.3) and gradient boosting decision trees (Sect. 4).

3.1 Scenario description

Given a data warehouse containing a star schema with a central fact table A and dimension tables B, C, and D, we consider the following scenario. Fact table A stores transactional data, while dimension tables B, C, and D contain contextual attributes associated with the facts in table A. A star join operation is applied to join the fact table A with dimension tables B, C, and D, leveraging their respective foreign key-primary key relationships. The resulting dataset S from this star join operation integrates facts and dimension attributes. Subsequent ML operators take dataset S as input to produce matrices for further applications.

The query above contains two relational operators: schema projection and join, which can be expressed in SQL as follows:

```
SELECT B.values, C.values, D.values FROM A
JOIN B ON A.B_key = B.key
JOIN C ON A.C_key = C.key
JOIN D ON A.D_key = D.key;
```

Based on the LAQ rewriting introduced in Sect. 2.1 and 2.3, we can generate indicator matrices through MM-Join and schema mapping matrices with projection for tables B, C, and D. Thus, the query can be expressed using matrix multiplications between the indicator matrix, source table, and schema mapping matrix. Specifically, by converting source tables to matrices, we can express the relational query with LAQ as follows:

$$T = I_1 B M_1 + I_2 C M_2 + I_3 D M_3 \tag{1}$$

In the rest of this section, we will continue using this example to explain how operator fusion integrates database queries and ML inference.

Operator fusion According to the design principles of star schema data warehouses discussed in [16], fact tables tend to exhibit higher update frequencies and larger data volumes compared to dimension tables. Consequently, fusing downstream ML operators with relatively static dimension tables allows for pre-fusing of partial results, thereby reducing the cost of predictions. We term this approach *operator fusion*. In the following sections, we leverage three models to demonstrate how operator fusion can accelerate predictive pipelines.

3.2 Fusing simple linear operators

Linear operators, essentially represented as matrices, are not only fundamental building blocks in linear algebra but also serve as the cornerstone for a wide array of general linear models. These models are crucial across diverse domains, from simple data transformations to complex machine learning algorithms. The ability to efficiently fuse linear operators with data processing queries significantly enhances computational performance, particularly in high-dimensional data scenarios.

Suppose the result of a star join is $T \in \mathbb{R}^{i \times k}$. According to Eq. (1), the query can be evaluated as $T = I_1 B M_1 + I_2 C M_2 + I_3 D M_3$, where $I_1 \in \{0,1\}^{i \times r_1}$, $I_2 \in \{0,1\}^{i \times r_2}$, $I_3 \in \{0,1\}^{i \times r_3}$, and $M_* \in \{0,1\}^{c \times k}$. c and k denote the number of columns in dimension tables and selected features for linear operators respectively.

The result S is then multiplied by a linear operator $L \in \mathbb{R}^{k \times l}$, resulting in predictions as $\mathbb{R}^{i \times l}$. For simplification, in the following analysis, we equally separate features into each dimension table, which means $c = \frac{k}{3}$. According to the s associativity of matrix multiplication, we can fuse L to dimension tables using:

$$\begin{aligned} \text{predictions} =& TL \\ =& (I_1 B M_1 + I_2 C M_2 + I_3 D M_3) L \\ =& I_1 \underline{B M_1 L} + I_2 \underline{C M_2 L} + I_3 \underline{D M_3 L} \end{aligned} \tag{2}$$

We follow the common assumption that dimension tables are less frequently updated than the fact table, BM_1L, CM_2L and DM_3L can be treated as constants in a period. Therefore, we can *pre-fuse* them and only apply row matching matrix I_* when materialization.

3.2.1 Complexity analysis

We now perform a complexity analysis for operator fusion in a predictive pipeline and compare it with non-fused methods. As both fused and non-fused methods share the same domain generation step, we will omit the complexity of domain generation in the following analysis for comparison purposes. Additionally, matrix additions have lower complexity order than matrix multiplications. Therefore, in the complexity analysis for this section and Sect. 3.3, *we omit the complexity of matrix additions*.

Given the aforementioned dimensions of matrices, the computational complexity without operator fusion is:

$$\begin{aligned}\mathcal{C}_{non\_fusion} &= \mathcal{C}_{mmjoin} + \mathcal{C}_{op} \\ &= ck \sum_j r_j + ik \sum_j r_j + ikl \\ &= \left(ik + \frac{k^2}{3} \right) \sum_j r_j + ikl\end{aligned}$$

The non-fusion method performs joining and multiplying linear operators separately. The joining step with MM-Join, as elaborated in Sect. 2.3.1 involves two steps. First, each source table generates the indicator matrices through a sparse multiplication between the mappings from the source key to the common domain, which takes $ck \sum_j r_j$. Then, the source tables need to multiply the indicator matrices, which costs $k \sum_j r_j$. Finally, after obtaining the target table, the complexity of multiplying the target and linear algebra is ikl.

An alternative way is pushing L down to dimension tables, we will get three pre-fused partial values of the final result, $BM_1L \in \mathbb{R}^{r_1 \times l}$, $CM_2L \in \mathbb{R}^{r_2 \times l}$ and $DM_3L \in \mathbb{R}^{r_3 \times l}$. Because the dimension tables and schema mappings are less frequently updated, we can store these pre-computed values and regard them as cache, saving runtime in query execution. As such, the linear operator can be directly applied to these partial results. Then we have the computation complexity as:

$$\mathcal{C}_{fusion} = il \sum_j r_j,$$

which is the summation of the complexity of three matrix multiplications between cached matrices and indicator matrices.

Now, we compare two complexity values:

$$\begin{aligned}\frac{\mathcal{C}_{non\_fusion}}{\mathcal{C}_{fusion}} &= \frac{\left(ik + \frac{k^2}{3} \right) \sum_j r_j + ikl}{il \sum_j r_j} \\ &= \frac{k}{l} + \frac{k^2}{3il} + \frac{k}{\sum_j r_j}\end{aligned} \tag{3}$$

Upon analyzing the information above, it becomes evident that the speedup of operator fusion is correlated with the shape of the linear operator and the cardinality of dimension tables. In practical predictive tasks, the total number of rows ($\sum r_j$) is usually much larger than the number of columns. Therefore, we can safely ignore the terms $\frac{k^2}{3il}$ and $\frac{k}{\sum_j r_j}$. In particular, $\frac{k}{l}$ can be considered as the filtering effect of the linear operator. For instance, a linear regression model can be viewed as a linear operator with an output shape of 1. By pre-fusing the linear regression model with dimension tables, the partial values to be composed after a join operation are vectors instead of matrices. Consequently, the execution time of the join-prediction

operation can be significantly reduced. In Sect. 5.3, we will examine the speedups of the fusion method concerning various input settings.

3.3 Fusing decision trees

Tree models, such as decision trees, are popular among data scientists due to their interpretability [25]. In this section, we elaborate on our operator fusion method with more complex decision tree models and explore optimization opportunities using a matrix representation of decision tree predictions. For clarity, our method is built on the linear algebraic representation of decision trees proposed by Hummingbird [19] in Sect. 3.3.1.

3.3.1 Matrix representation of decision trees

Hummingbird [19] introduces a method to represent decision tree models using linear algebra operators. The key idea is to transform the tree structure into a set of linear algebraic operations and vectorized predicates, which can then be efficiently executed on hardware optimized for such computations, like GPUs.

Figure 4 illustrates the structure of a matrix-represented decision tree and the computing process for inference. Matrix F is used to map the schema in the target table to features in the decision tree based on the given feature order. Vector v represents the decision values of each feature. Matrix H and vector h collectively represent the structure and decision path of the decision tree. Specifically, in matrix H, the column indices represent the leaf indices, and the values in each column correspond to the path to reach that leaf. The row indices indicate the encoded nodes.

Suppose we have a batch of vectors $S \in \mathbb{R}^{i \times k}$. To represent a tree, we need two binary matrices, $F \in \{0,1\}^{k \times p}$ and $H \in \{-1,0,1\}^{f \times l}$, as well as two vectors, $\mathbf{f} \in \mathbb{R}^p$ and $\mathbf{h} \in \mathbb{R}^l$. As Fig. 4 shows, inference over the matrix representation mainly has 4 steps. The final result is a binary encoding for the prediction label, which can be subsequently retrieved through a lookup table.

- *Step 1.* The binary matrix F is an orthogonal matrix that maps input vectors to features. Some columns may not be in the selected features; thus, the matrix serves as a feature selection operator. As the initial linear operator of the decision tree, its orthogonality enables operator fusion because the result of TF is a linear combination of the original columns in the input. In practice, F can be

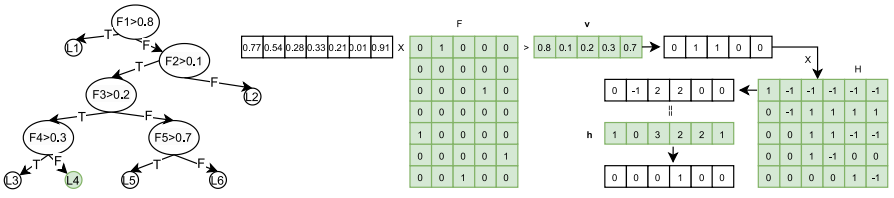


Fig. 4 Prediction with a decision tree with linear algebraic representation adapted from [19]

integrated into the column mapping matrix M_* (described in Sect. 2.1), but we retain it in the rest of the implementation for completeness.

- *Step 2.* Vector **v** represents the values of nodes in the decision tree. The order of this vector follows a pre-assigned rank. The output of operator F undergoes a predicate '$> \mathbf{v}$', producing a binary vector. In practice, we often apply predictions to a batch of vectors; as such, the output of this step turns out to be a matrix. Notably, since the output of the last step is a linear combination of the original input, each column can be independently compared to the corresponding values in **v**. This means that the predicate can be fused with dimension tables as well.
- *Step 3.* Matrix H signifies the structure of decision trees. Each column represents the path of a leaf node. Values in the column indicate the choices of nodes on this path, where 1 means True, and -1 means False. For instance, the path to L2 contains two nodes, F1 and F2, both of which choose the False side. Consequently, the values of the corresponding nodes are -1. Notably, H is a reduction operator. Fusing H with dimension tables is applicable but only produces a local sum.
- *Step 4.* Vector **h** is the column sum of positive values in matrix H. Before comparing with **h**, the pre-fused matrices must be added to obtain a complete vector. The result vector is compared with **h**, and a one-hot encoding denoting prediction labels will be produced. We can compute dot product between the output vector and the vector storing leaf values to get the final prediction.

3.3.2 Fusing with dimension tables

As we discussed in last section, the prediction results of the decision tree over T can be represented by:

$$predictions = \overbrace{(\overbrace{\overbrace{TF}^{\text{step 1}} > \mathbf{v}}^{\text{step 2}})H == \mathbf{h}}^{\text{step 3}}\Big\}\text{step 4}$$

In the scenario introduced in Sect. 3.1, we can replace the target table T with Eq. (1), yielding the following:

$$predictions = (((I_1BM_1 + I_2CM_2 + I_3DM_3)F > \mathbf{v})H == \mathbf{h} \tag{4}$$

Subsequently, we explore the strategy for pushing the subsequent operators down to the individual dimension tables to enable the separate evaluation of partial results. Initially, the matrix F is identified as a linear operator, a concept previously addressed in Sect. 3.2. Consequently, Eq. (4) can be reformulated as:

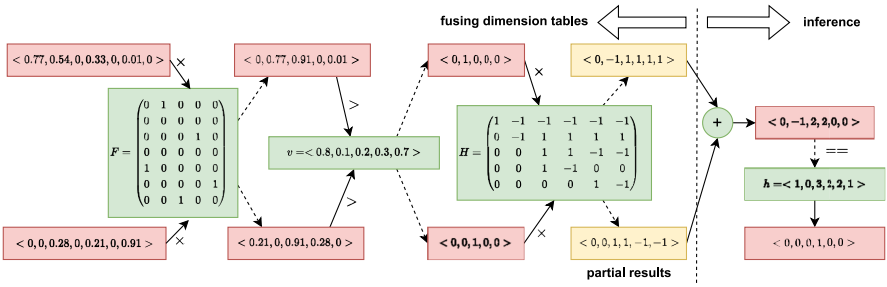


Fig. 5 Following the example in Fig. 4, this figure showcases fusing dimension tables and inference using partial results. First 3 steps can be computed locally. Step 4 is computed when a query comes in

$$predictions = ((I_1BM_1F + I_2CM_2F + I_3DM_3F) > \mathbf{v})H == \mathbf{h} \tag{5}$$

The predicate operator '$> \mathbf{v}$' can also be pushed down to the dimension tables. Let's examine the binary orthogonal matrix F first. It acts as a mask, extracting vector elements based on the positions of the non-zero entries in each column of F. For instance, as depicted in Fig. 5, the outcome of the multiplication between F and the vector $< 0.77, 0.54, 0, 0.33, 0, 0.01, 0 >$ is $< 0, 0.77, 0.91, 0, 0.01 >$. Thus, the predicate '$>$' for each column is independent of the values in other columns, indicating that the predicate can be evaluated partially within each dimension table. Accordingly, Eq. (5) can be rewritten as:

$$predictions = (I_1BM_1F > \mathbf{v} + I_2CM_2F > \mathbf{v} + I_3DM_3F > \mathbf{v})H == \mathbf{h} \tag{6}$$

Continuing with this approach, the matrix H, also a linear operator, can be pushed down similarly to matrix F, leading to the following:

$$predictions = ((I_1BM_1F > \mathbf{v})H + (I_2CM_2F > \mathbf{v})H + (I_3DM_3F > \mathbf{v})H) == \mathbf{h} \tag{7}$$

The predicate '$== \mathbf{h}$', however, cannot be further pushed down. Unlike F, matrix H is not orthogonal, which implies that the outcome of '$== \mathbf{h}$' is contingent upon the entire set of values within the input vector.

In conclusion, we have identified all operators within a decision tree evaluation that are amenable to being pushed down to dimension tables. For clarity, we summarize the partial results as T_i and define the fused decision tree over the dimension tables as follows:

$$predictions = (I_1T_1 + I_2T_2 + I_3T_3) == \mathbf{h} \tag{8}$$

Figure 5 illustrates how the first three steps are pushed down to the dimension tables. In this figure, we split the input vector from Fig. 4 into two parts to represent vectors in two dimension tables. The first three steps can be computed locally, and the partial results T_1, T_2, T_3 can be cached. Step 4 must be executed as inference because it depends on the summation of the partial results.

3.3.3 Complexity analysis

Similar to the complexity analysis for the linear operator, we first present the complexity of non-fusion method:

$$\begin{aligned}\mathcal{C}_{non\_fusion} &= \mathcal{C}_{mmjoin} + \mathcal{C}_F + \mathcal{C}_{\mathbf{v}} + \mathcal{C}_H + \mathcal{C}_{\mathbf{h}} \\ &= \left(\frac{k^2}{3} + ik\right)\sum_j r_j + ikp + ip + ipl + il\end{aligned}$$

The non-fusion method needs to materialize the data for inference via MM-Join. The materialized data is then applied to a sequence of computations with the operators in a tree. Each item in the equation represents the computational complexity of matrix-matrix multiplications and matrix–vector multiplications.

With operator fusion presented in Eq. (8), we have three pre-fused matrices whose dimensions are $R^{r_i \times l}$. As such, the remaining part involves a matrix summation for aggregating the partial results and a matrix–vector predicate to retrieve the output leaf. The complexity of remaining operations of decision tree's result can be expressed by:

$$\mathcal{C}_{fusion} = il\sum_j r_j + il$$

Then we compare the complexities through $\frac{\mathcal{C}_{no-fusion}}{\mathcal{C}_{fusion}}$, which is:

$$\frac{\left(\frac{k^2}{3} + ik\right)\sum_j r_j + ikp + ip + ipl}{(il+1)\sum_j r_j} \tag{9}$$

r_j represents the number of rows in a dimension table, while p denotes the number of features. For simplicity, we assume the number of features (p) equals to length of input (k). Additionally, considering $il \gg 1$, we remove the constant term. Then, we have:

$$\begin{aligned}\frac{\mathcal{C}_{non\_fusion}}{\mathcal{C}_{fusion}} \approx& \frac{\frac{k^2}{3}}{il} + \frac{ik}{il} + \frac{ik}{il\sum_j r_j} + \frac{ikl}{il\sum_j r_j} + \frac{1}{\sum_j r_j} \\ =& \frac{k}{l} + \frac{k^2}{3il} + \frac{k^2}{l\sum_j r_j} + \frac{k}{\sum_j r_j} + \frac{k}{l\sum_j r_j} + \frac{1}{\sum_j r_j}\end{aligned} \tag{10}$$

Due to the involvement of more linear operators in decision trees, additional tail terms appear in Eq. (10). When r_j is smaller than the number of features, we can expect that our approach facilitates a certain speedup through operator fusion. In contrast, when the size of dimension tables is significantly larger than k, the speedup is correlated with the first term $\frac{k}{l}$. Similar to the discussion in Sect. 3.3, the filtering effect of decision trees determines the benefit of operator fusion. If a tree has only a small number of leaves, a significant speedup can be expected. We will further substantiate this analysis with experimental results in Sect. 5.3.

Our analysis above assumes that all matrices involved in the computation are dense matrices. In our implementation, matrices I, M, and F are stored in the CSR format and computed using a sparse matrix multiplication kernel. This approach has lower computational complexity compared to naive dense matrix multiplication, further enhancing the efficiency of the overall computation process.

4 Fusing gradient boosting decision trees

Building on the foundation of fusing linear operators and decision trees, we extend our exploration to the more complex and practically significant model of Gradient Boosting Decision Trees (GBDT) [7]. Recognizing the intricacies and critical role of GBDT in real-world predictive pipelines, this section delves not only into the application of our fusion method to GBDT but also introduces our novel tensor representation. This representation expands upon the matrix approach used for decision trees, offering enhanced computational efficiency. By focusing on GBDT, we aim to present a more comprehensive understanding of fusion techniques in sophisticated modeling scenarios.

4.1 Matrix representation of GBDT predictions

In a GBDT model, each decision tree plays a role in shaping the final prediction through an additive process. This contribution is modeled by extending the matrix and vector operations that are typically used for a single decision tree. In a GBDT model comprising t trees, we create a series of structures introduced in Sect. 3.3.1. The prediction process involves applying each tree's linear and predicate operators sequentially, accumulating the results. This can be represented as:

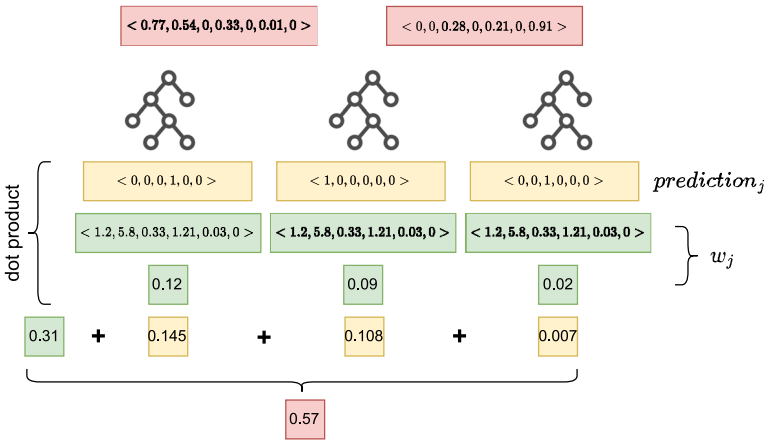


Fig. 6 GBDT models make predictions by summing the numerical results from each constituent decision tree

$$\text{GBDT Predictions} = \text{Initial Prediction} + \eta \sum_{j=1}^{t} w_j \cdot prediction_j \tag{11}$$

In this context, η represents the learning rate, and the initial prediction is typically the average value of the target variable. The one-hot encoded vector prediction_j, as defined in Eq. (8), indicates the predicted value for tree j.

Fig. 6 demonstrates the prediction process of GBDT models, which relies on aggregating the outcomes from each individual decision tree. Essentially, the final prediction of a GBDT model is computed as a weighted sum of the numerical results contributed by each tree. The vector w_j associated with tree j is derived from the dot product of the leaf values and the corresponding weight of tree j. Subsequently, the dot product of w_j and prediction_j yields the numerical outcome for tree j. The computational complexity of this dot product is directly proportional to the number of leaves in the tree. Considering that the number of leaves is generally much smaller than $\sum_j r_j$, for simplicity, we omit this operation from the subsequent complexity analysis.

4.2 Fusing with dimension tables

The fusion of GBDT with dimension tables follows a similar approach to decision trees, with the added complexity of handling the sequential and additive nature of GBDT. Specifically, Eq. (11) illustrates that the final outcome of the GBDT model is derived from a weighted summation of the results contributed by each constituent tree. The summation operator cannot be further pushed down to dimension tables; hence, the operation fusion process for the GBDT model mirrors the operator fusion in decision trees. In other words, only the operators we discussed in Sect. 3.3.2 can be pre-computed in conjunction with dimension tables. With this in mind, utilizing Eq. (8), we can express operator fusion in GBDT as follows:

$$\text{GBDT Predictions} = \text{Initial Prediction} + \eta \sum_{j=1}^{t} w_j \cdot ((I_1 T_1^j + I_2 T_2^j + I_3 T_3^j) == \mathbf{h}_j) \tag{12}$$

In the given equation, T_i^j represents a partial result that has been fused with a dimension table in tree j.

4.3 Complexity analysis

The complexity analysis for GBDT when applying operator fusion is distinct from that of a single decision tree, primarily due to the additive nature of the trees in GBDT. Assuming the complexity of operations for each tree is comparable, the overall complexity of the GBDT model can be estimated to be about t times that of a single decision tree. Consequently, the complexity ratio between the non-fusion

and fusion methods, as initially described in Eq. (9), transforms into the following equation:

$$\frac{\left(\frac{k^2}{3} + ik\right) \sum_j r_j + t(ikp + ip + ipl)}{t(il + 1) \sum_j r_j} \tag{13}$$

Expanding and simplifying this equation, the complexity ratio for GBDT can be expressed as:

$$\begin{aligned} \frac{\mathcal{C}_{non\_fusion}}{\mathcal{C}_{fusion}} \approx & \frac{\frac{k^2}{3}}{til} + \frac{ik}{til} + \frac{tik}{til \sum_j r_j} + \frac{tikl}{til \sum_j r_j} + \frac{t}{t \sum_j r_j} \\ = & \frac{k}{tl} + \frac{k^2}{3til} + \frac{k^2}{tl \sum_j r_j} + \frac{k}{\sum_j r_j} + \frac{k}{l \sum_j r_j} + \frac{1}{\sum_j r_j} \end{aligned} \tag{14}$$

In Eq. (14), the dominant term becomes $\frac{k}{tl}$, which represents the ratio of the number of input features to the product of the number of trees and the number of leaves in each tree. Notably, $t * l$ represents the total number of leaves in a GBDT model. Similar to the ratio $\frac{k}{l}$ in decision trees, $\frac{k}{tl}$ serves as a key indicator of the potential speedup through operator fusion for GBDT models. A substantial speedup from the fusing method is attainable when the count of input features k greatly exceeds total number of leaves l.

4.4 Implementation of fusing GBDT

Instead of representing each tree in a GBDT model as a set of matrices and vectors and calculating their results iteratively, our fused GBDT implementation employs high-dimensional tensor representation. Given that a GBDT model consists of multiple trees, each associated with its matrices F, and H, as well as vectors $\mathbf{v}$, and $\mathbf{h}$, we extend this structure into a three-dimensional tensor space. This tensor-based approach facilitates more efficient computation, particularly when utilizing the cuTensorNet[4] library for GPU-accelerated hardware, thereby enhancing the overall computational performance of the GBDT model.

Efficient memory access One of the primary advantages of using tensor representation in GBDT models is the consolidation of data for each tree within the same physical memory space. This allocation allows for coalesced memory access, which is significantly more efficient than accessing separate memory blocks. In traditional tree representations, data are often scattered across different memory locations, leading to increased memory access times and reduced computational efficiency. However, with tensor representation, data retrieval becomes streamlined, as contiguous memory blocks can be accessed in a single, unified operation. This optimization

[4] https://docs.nvidia.com/cuda/cuquantum/latest/cutensornet/index.html.

reduces the overhead associated with memory access, thereby enhancing the overall performance of fused GBDT.

Efficient computation with tensor algebra Another significant benefit arises from the high parallelism inherent in GPU architectures and the capabilities of specialized vendor libraries for tensor computation. The cuTensorNet library, specifically designed for GPU-accelerated environments, offers a 'tensor contraction' function that plays a vital role in this context. This function enables the library to allocate resources strategically and optimize computations for a set of tensor algebra operations. For instance, in a GBDT model with an input data T and a tensor $\begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$, the conventional approach involves sequentially computing TF_1 and TF_2. However, cuTensorNet can potentially process these operations in parallel if the workload is manageable, offering speedups beyond theoretical expectations. Such speedups depend on specific workloads and data structures. If the workload for each tree exceeds the resource capacity, the speed degrades that of the iterative approach. This variability underscores the need for practical modeling to better anticipate the actual computational cost.

In summary, the implementation of GBDT using high-dimensional tensor algebra, with the aid of cuTensorNet library, represents a significant advancement in the efficient computation of GBDT models. This approach not only simplifies the computational process but also opens up possibilities for substantial performance improvements, particularly in GPU-accelerated environments.

4.5 Summary

In this section, we proposed an operator fusion method to accelerate predictive pipelines, building on the preliminaries introduced in Sect. 2. Within the context of data warehouses, we presented two predictive pipelines that demonstrate how operator fusion accelerates them by pre-fused partial results. Additionally, we compare the theoretical complexity of fusion and non-fusion methods and identify a preliminary decision boundary for determining when to apply operator fusion for speedup. In Sect. 5.3, we will present the speedup of operator fusion in predictive pipelines.

It is important to note that three examples in this section assume dimension tables are updated less frequently than the fact table, which is a common design principle in traditional data warehouses. However, many data architectures (e.g., data mesh [18], data fabric [9]) proposed in recent years have gradually deviated from this principle. Further investigation is needed to determine the applicability of the operator fusion method in these scenarios.

5 Experimental evaluation

In this section, we evaluate the performance of LAQ and examine the performance enhancement achieved through operator fusion. In Sect. 5.2, we use the SSB [21] dataset to compare the performance of LAQ with two GPU-accelerated relational

Table 2 Summary for query groups in SSB

Queries	Group 1	Group 2	Group3	Group 4
ID of subqueries	11, 12, 13	21, 22, 23	31, 32, 33	41, 42, 43
# Joins	1	3	3	4
Aggregations	Sum	Group-by sum	Group-by sum	Group-by sum
Sorting	No	Yes	Yes	Yes

Table 3 Types and cardinalities of SSB tables

Tables	Type	Cardinality
Lineorder	Fact	$sf * 6,000,000$
Part	Dim	$200,000 * floor(1 + \log_2 sf)$
Supplier	Dim	$sf * 2000$
Customer	Dim	$sf * 30,000$
Date	Dim	$7 * 365$

sf is a parameter controlling data sizes

query processing engines. Before presenting the experimental results, we first provide an overview of the experimental setup, encompassing the implementations under evaluation, dataset characteristics, and hardware.

5.1 Experiment settings

Implementation. In this paper, we implement GPU-accelerated LAQ using CuPy [20]. The implementation involves two-way join, multi-way join, and bi-group aggregation, all of which are computed using CSR format with CuSparse, which is a CUDA library for sparse matrix multiplication.[5]

Baselines To assess the performance of our LAQ implementation, we compare it with two other GPU-accelerated data processing libraries: HeavyDB [12] and cuDF [26]. HeavyDB, formerly known as OmniSciDB, is a commercial GPU data management system that supports a wide range of relational queries on GPUs. It features a query optimizer and cache strategy to expedite query execution. cuDF, on the other hand, is a GPU DataFrame library that provides support for commonly used relational operators, such as selection, projection, join, and aggregations. Unlike HeavyDB, cuDF is a vanilla query processor, similar to our LA query implementation, and serves as an appropriate baseline for our study.

[5] https://docs.nvidia.com/cuda/cusparse/index.html.

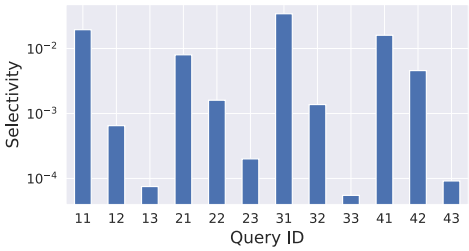


Fig. 7 Selectivity of each query in SSB

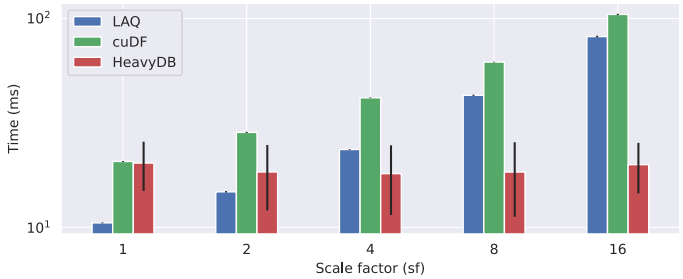


Fig. 8 Average execution time under various scale factors

Workload For the evaluation of operator fusion, we use star join queries on a synthetic dataset characterized in Table 4, and the results are subsequently utilized as input for linear operators.

Hardware All the implementations in these experiments are executed on an Nvidia A40 (48 GB) GPU, eliminating the need to account for the communication cost between host memory and device memory. Each experiment is conducted ten times, and we report the mean values and standard errors.

5.2 Performance evaluation for LAQ

In this section, we evaluate the performance of LAQ using the SSB dataset on GPUs and compare the results with two GPU-accelerated data processing engines. First, we measure the average execution time with respect to varying scale factors (*sf*), and then we examine LAQ's performance on different queries. To identify the most time-consuming operator, we provide a performance breakdown and suggest an optimization opportunity for future research.

5.2.1 SSB dataset

We use Star Schema Benchmark (SSB) [21] to investigate the performance in real-world workloads. The SSB dataset is a widely-used benchmark for evaluating the performance of data warehouse systems and database management systems. It was developed as a simplified version of the TPC-H benchmark, which is also designed

for benchmarking data warehouse systems. The SSB dataset focuses on star schema query processing and comes with a predefined set of queries that test various aspects of database performance, such as join operations, aggregations, and filtering.

Table 2 provides a summary of the workloads and query groups, while Table 3 displays the types and cardinality settings for each table in the SSB dataset. Additionally, Fig. 7 illustrates the selectivities of each query for subsequent evaluations. The parameter *sf* represents the scale factor that controls data sizes, and it will be used to denote the scale of data throughout the rest of the paper.

5.2.2 Evaluation results for LAQ

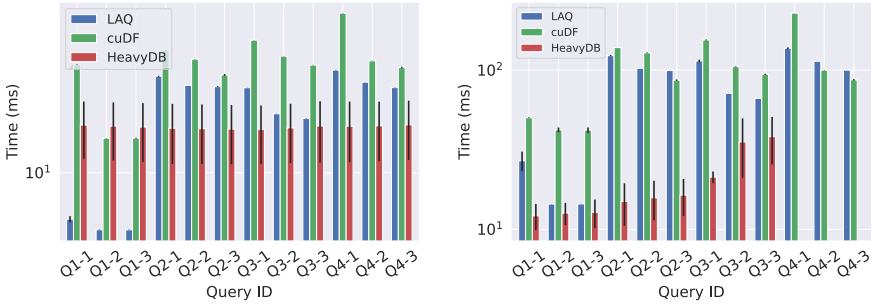
Q_1 : *When does LAQ perform better than cuDF and HeavyDB w.r.t. varying data sizes*?

Observation Fig. 8 illustrates the average execution time of all queries under different scale factors. HeavyDB presents similar average performance with different *sf* but has significant standard errors. MM-Join exhibits a significant advantage against the other two systems at small scale factor.s As the scale factor increases, the performance of LAQ turns out to be slower than HeavyDB and approaches cuDF.

Analysis HeavyDB is a well-designed data management system with dedicated caching mechanisms. When the evaluation executes repeatedly, more data are cached in global memory, which leads to superior performance at large *sf*. cuDF and LAQ are vanilla implementation join algorithms. They can not obtain advantages through caching strategies during repeated experiments. Another notable finding is that performance of LAQ degrades faster than cuDF due to the high computational complexity of the spMM kernel.

Q_2 : *How do speedups of LAQ against cuDF and HeavyDB vary w.r.t different queries*?

Observation Figs. 9a, b show execution time with respect to different queries under $sf = 4$ and $sf = 16$. We can find out that all systems perform faster in query group 1. LAQ exhibits noticeable speedups against the other two systems in query group 1 when $sf = 4$. We can also observe that LAQ becomes slower than cuDF in query 42 and 43 when $sf = 16$.



(a) Average execution time of different queries when $sf = 4$.

(b) Average execution time of different queries when $sf = 16$.

Fig. 9 Average execution time of subqueries

Figure 9a does not show the results of HeavyDB for query group 4 when sf = 16 due to out-of-memory error raised during evaluation. This error was caused by the size of intermediate results exceeding the memory capacity.

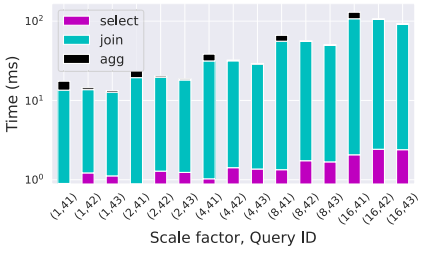
Analysis Matrix multiplication (MM), which is the most common operation in LAQ, is known to effectively exploit GPU parallelism due to the inherent nature of LA algorithms. However, this does not eliminate the computational complexity disadvantage of MM. When processing large-scale data, this increased complexity causes MM-Join to underperform compared to the partitioned hash join in cuDF.

Moreover, we observed a positive correlation between the performance of LAQ and the selectivity of the queries, as illustrated in Fig. 7. Among all results, the performance on query 33 is exceptional. Although query 33 has lower selectivity, both algorithms exhibit slower performance due to an additional join operation. Interestingly, HeavyDB does not display a correlation between performance and selectivity. Query group 3 involves more joins, resulting in increased data movement in memory. When the scale factor is small, memory bandwidth is not a bottleneck. However, when the scale factor reaches 16, the excessive memory access during joins can degrade overall speed due to becoming memory-bound.

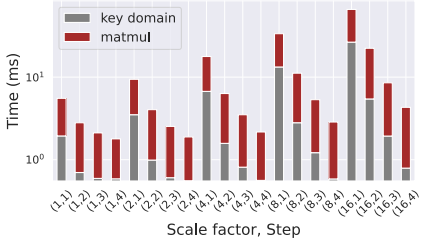
Q_3 : *Which operator in a query needs more optimization?*

Observation In this experiment, we use query group 4 as an example to present the breakdown performance of MM-Join. Figure 10a shows the execution time of different operations in a query. Apparently, join operations dominates the execution time. In Fig. 10b, we further investigate two primary operations of joins. Domain generations take a similar portion of execution time within 4 join steps in query group 4, whereas join operations' portion decreases as the selectivity decreases.

Analysis In Sect. 2.3, we have learned that the computational complexity of domain generation is $O(n^2 \log n)$, independent of selectivities. This operation becomes particularly costly when selectivity is low. Nonetheless, the domain consists of a union of tables, allowing us to cache the domain for reuse. Caching proves advantageous when key updates are infrequent. If updates to the cached domain are necessary, the complexity of searching and inserting into a sorted array is $O(n + \log n)$, which is still more efficient than rebuilding the domain from scratch.



(a) Performance breakdown for queries w.r.t different scale factors. We take query group 4 as an example.



(b) Performance breakdown for MM-Join w.r.t different scale factors. We take query group 4 as an example.

Fig. 10 Performance breakdown

As a result, we can further enhance performance by employing domain caching strategies.

5.3 Performance evaluation for operator fusion

In this section, we evaluate operator fusion examples introduced in Sects. 3.2 and 3.3, and 4, to demonstrate the performance improvement that operator fusion brings to predictive pipelines. In addition to evaluating performance with different *sf*, we also examine the impact of model shape. Specifically, we vary the shape of models with different values of k and l to validate a potential factor, $\frac{k}{l}$, that may influence the speedup of operator fusion.

5.3.1 Datasets

To thoroughly assess the performance of operator fusion across a broad range of data characteristics and within realistic predictive workloads, we employ a synthetic dataset derived from the Star Schema Benchmark (SSB) and a subset of the TPC-DI benchmark. This combination allows for a comprehensive evaluation under diverse data conditions and practical scenarios.

Synthetic dataset SSB queries are well-suited for evaluating operator fusion, as our scenario setting aligns with the design principles of SSB. Because allocating SSB dataset and models to be evaluated in the same GPU causes out-of-memory error, we generate a synthetic dataset based on down-scaled cardinalities of SSB tables for operator fusion experiments. The detailed table design is shown in Table 4.

In addition to varying settings in Table 4, we also alter the size of models to be fused in order to test the performance of operator fusion under different computing workloads. To match the input shape k of models, we adjust the number of columns accordingly. The parameters of linear operators are demonstrated in Table 5.

We defined two distinct groups of cardinality settings to evaluate performance under different scenarios: Cardinality setting 1 is characterized by a large volume of records to be predicted with a smaller model size, while cardinality setting 2 involves fewer input records but uses a larger model.

TPC-DI benchmark To assess the performance of our operator fusion method in practical settings, we perform experiments on real-world data integration using the TPC-DI benchmark [24]. This involves four datasets distributed among three

Table 4 Types and cardinalities of synthetic tables

Tables	Type	Cardinality setting 1	Cardinality setting 2
Lineorder	Fact	$sf * 600,000$	$sf * 3000$
Part	Dim	$20,000 * floor(1 + \log_2 sf)$	$2000 * floor(1 + \log_2 sf)$
Supplier	Dim	$sf * 2000$	$sf * 2000$
Date	Dim	$7 * 365$	$7 * 365$

sf is a parameter controlling data sizes

Table 5 Parameters for linear operator and decision tree

Simple linear operator				
Cardinality	sf	input length (k)	Output length (l)	
Setting 1	1, 2, 4, 8, 16	$2^{[4\ldots7]}$	$2^{[1\ldots7]}$	
Setting 2	1, 2	$2^{[8\ldots11]}$	$2^{[1\ldots k]}$	
Decision tree				
Cardinality	sf	input length (k)	#Features (p)	#Leaves (l)
Setting 1	1, 2, 4, 8, 16	$2^{[4\ldots7]}$	$2^{[4\ldots7]}$	$2^{[1\ldots6]}$
Setting 2	1, 2	$2^{[8\ldots11]}$	$2^{[3\ldots11]}$	$2^{[6\ldots11]}$

GBDT					
Cardinality	sf	Input length (k)	#Features (p)	#Trees (t)	#Leaves (l)
Setting 1	1, 2, 4	$2^{[4\ldots7]}$	$2^{[4\ldots7]}$	5, 15, 20	8, 16, 32
Setting 2	1, 2	$2^{[8\ldots11]}$	$2^{[3\ldots11]}$	25, 50, 100	8, 16, 32

Table 6 Data sizes of the realistic data integration scenario based on TPC-DI benchmark

Scale factors	Source 1	Source 2	Source 3		Target
	Trade	Customer	Stock	Reports	
3	390,979	4729	2620	98,778	528,798
5	650,413	7801	4202	198,263	949,623
7	911,674	10,865	5802	297,748	1,347,168
9	1,171,674	13,896	7401	397,022	1,738,255
11	1,430,674	16,984	9003	496,820	2,139,083

The table shows the number of rows in sources (r_j) and the target table w.r.t varying scale factors

distinct sources, each holding different segments of data. The datasets are joined using keys in the fact table, namely *Trade*. The resulting target table has 27 output features. Table 6 details the cardinalities of each table under various scale factors, providing a comprehensive view of the data landscape across different scenarios. This setup enables us to thoroughly evaluate the effectiveness of the fusion method in a realistic, multi-source data integration context.

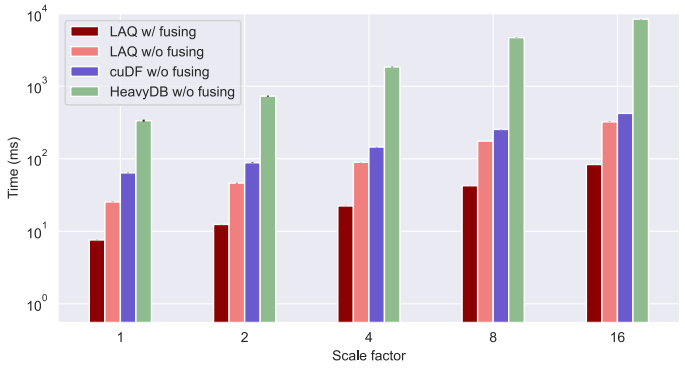
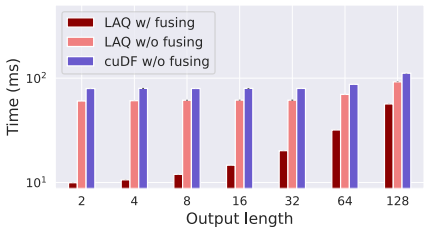


Fig. 11 Average execution time of join with and w/o fusing linear operators under different scale factors. The experimental scenario is cardinality setting 1

Fig. 12 Average execution time of predictive pipeline of simple linear operator with and w/o operator fusion when $sf = 4$. The experimental scenario is cardinality setting 1



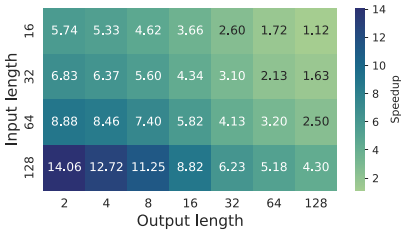


Fig. 13 Heatmap of speedup w.r.t lengths of input and output when $sf = 8$

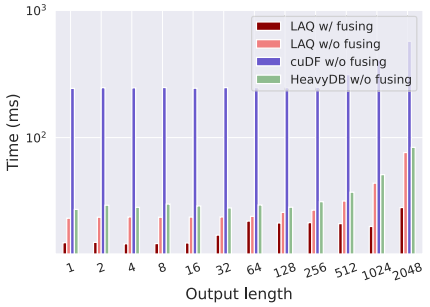


Fig. 14 Average execution time of predictive pipeline of decision tree with and w/o operator fusion when $sf = 2$. The experimental scenario is cardinality setting 2

5.3.2 Evaluation results for simple linear operators

This example exhibits a scenario where the output of join operations is fed to a linear operator producing a matrix. We separately evaluate two conditions: input with large *sf*, where the cardinality setting 1 is enabled, followed by a small linear operator, and input with small *sf* (cardinality setting 2) connected to a relatively large operator.

Q_4 : *How much speedup can fusing linear operators deliver in cardinality setting* 1?

Observation In this experiment, we compare the execution time of a star join with operator fusion to LA-after-join implementations. Figure 11 displays the average execution time under various scale factors. It is evident that HeavyDB's execution time consistently exceeds that of cuDF by a factor of more than five, on average, at scale factors 1 and 2. Furthermore, at a scale factor of 16, the performance gap widens significantly, with HeavyDB's execution time lagging behind cuDF by an order of magnitude.

In the subsequent sections of this paper, unless there are compelling reasons to do otherwise, we have chosen to exclude HeavyDB's performance results when they are markedly inferior to those of the other methods under comparison.

The fusion method outperforms the other two implementations. In Fig. 12, we hold all parameters constant except for the output shape of the linear operator. Both cuDF and the non-fusion method do not exhibit significant changes in execution time compared to the fusion method. Although the fusion method still demonstrates speedups, these speedups continue to decrease as the output shape grows larger.

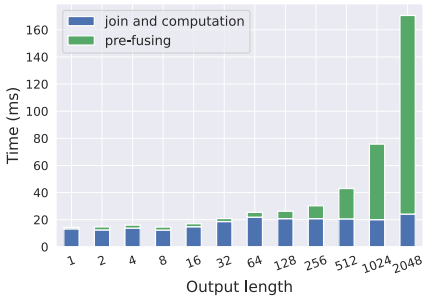


Fig. 15 The execution time that pre-fusion stage and join-computation stage take in prediction with linear operator after joining

Analysis Through Eq. (3), we understand that the speedup is negatively correlated with output shape l and positively correlated with input width k. Due to the large input size in this experiment, the lower order terms $\frac{k^2}{3il}$ and $\frac{k}{\sum_j r_j}$ in the equation are neglectable. Consequently, in Fig. 12, we observe that the speedup of the fusion method gradually decreases as l increases. Additionally, we illustrate the speedup values concerning different k and l, while maintaining $sf = 8$, in Fig. 13. The highest speedup occurs at the largest k and smallest l, whereas the lowest speedup is found along the diagonal. This result validates our analysis derived from Eq. (3).

Q_5 : *How much speedup can fusing linear operators deliver in cardinality setting* 2?

Observation In this experiment, we set the base cardinality to 1/1000 of SSB and enlarge the output shape l up to 2^{11}. Comparing Figs. 13 and 11, we can clearly observe much more significant speedups of operator fusion in small dimension tables in cardinality setting 2.

Furthermore, while HeavyDB consistently underperformed in the results of Query Q_4, it outperformed cuDF in the cardinality setting 2, as depicted in Fig. 14. Given that both HeavyDB and cuDF employ non-fusion methods, the observed performance discrepancy can be attributed to differences in query execution times.

Analysis In our test scenario, dimension table columns are initially joined together, followed by multiplying a schema mapping matrix to perform projections. This approach requires extensive column manipulation during the join operation.

In cardinality setting 2, the target table contains a substantially greater number of columns than in cardinality setting 1. cuDF, which manages tables as row-major numeric arrays, faces increased computational costs for column-wise data manipulation, particularly when joining wide tables compared to those with a smaller number of columns. On the other hand, HeavyDB, as a fully-fledged database system, segments columns into more granular fragments and chunks.[6] This design enhances performance when handling the extensive column manipulations typical of cardinality setting 2. However, this advantage comes with a trade-off: when dealing with high r_j values, the conversion of block-based data structures to arrays requires additional time, leading to degraded performance compared to

[6] https://github.com/heavyai/heavydb/blob/master/docs/source/data_model/columnar_layout.rst.

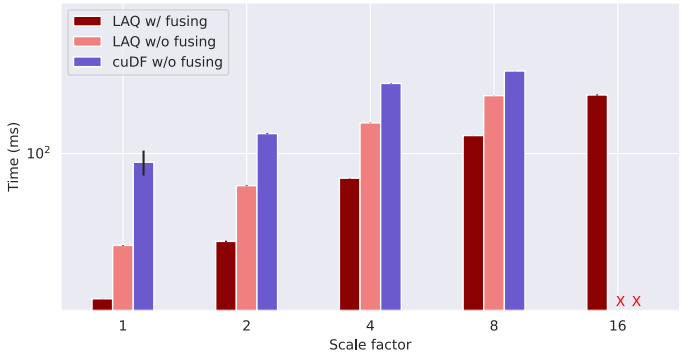


Fig. 16 Average execution time of join with and w/o fusing decision trees regarding different scale factors. The experimental scenario is cardinality setting 1

Fig. 17 Average execution time of predictive pipeline of decision tree with and w/o operator fusion when $sf = 4$. The experimental scenario is cardinality setting 1

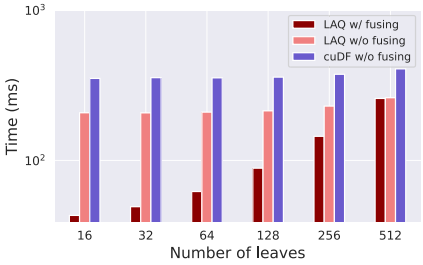
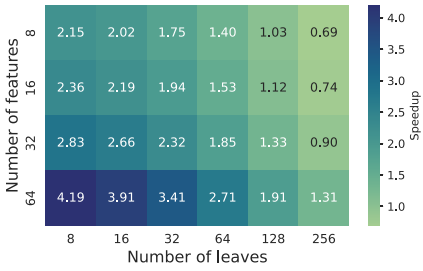


Fig. 18 Heatmap of speedup w.r.t numbers of features and leaves when $sf = 8$



the array-native cuDF. Similar trends were observed in subsequent sections when evaluating decision trees and GBDT models in cardinality setting 2.

Q_6 : *How much time does the pre − fusion phase take?*

Analysis As indicated by Eq. (3), a reduction in input cardinality corresponds to a smaller value for $\sum_j r_j$, resulting in a larger speedup. Furthermore, HeavyDB is slower than both LAQ with and without operator fusion due to data structure conversions across different runtimes between the database and ML systems. Thus, we can conclude that the fusion method is more advantageous when processing linear queries with small dimension tables.

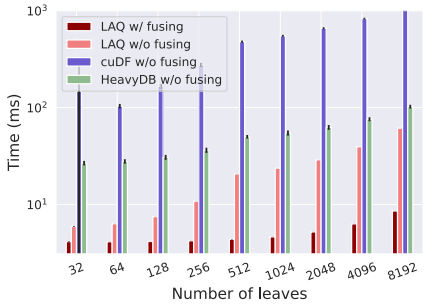


Fig. 19 Average execution time of predictive pipeline of decision tree with and w/o operator fusion when $sf = 2$. The experimental scenario is cardinality setting 2

Observation and analysis While the operator fusion method provides considerable speedup, it is crucial to consider the cost of the pre-fusion step, as shown in the underlined parts of Eq. (2). This is because dimension tables, although updated less frequently than fact tables, are not static constants. Moreover, the pre-fused tables may be larger than the original dimension tables when the output shape exceeds the number of columns, resulting in increased memory usage. Consequently, a quantitative trade-off between fusion and non-fusion methods still calls for further study in practice.

Figure 15 presents a stacked plot illustrating the relative proportion between pre-fusion and subsequent multiplication with I_*. Based on the parameter settings in Q5, we observe that when the output shape l is less than or equal to 512, the linear operation dominates the total execution time. As a result, if memory constraints are present, we can prioritize query completion without encountering out-of-memory errors, considering the diminishing speedup with larger output shapes.

5.3.3 Evaluation results for decision trees

In this experiment, we substitute the simple linear operator with a more intricate decision tree model to explore the performance advantages resulting from operator fusion. Following a similar experimental approach for simple linear operators, we separately assess the performance of two scenarios: cardinality setting 1 followed by a simple decision tree and cardinality setting 2 followed by a relatively large model.

Q_7 : *How much speedup can fusing decision trees deliver in cardinality setting* 1?

Observation and analysis Figs. 16, 17 and 18 display the results for large input scenarios. Figure 16 demonstrates that the average execution time of the fusion method is significantly faster than the other two methods across all scale factors. Notably, both cuDF and the non-fusion method fail to execute due to out-of-memory errors, while the fusion method completes a larger portion of evaluations. In Fig. 17, we vary parameter l while keeping $sf = 4$. It is evident that LAQ with fusion outperforms other methods when l is low, but its performance deteriorates as l increases. Another observation is that the performance of both *LAQ without fusing* and *cuDF without fusing* is not significantly influenced by the number of leaves. According to the complexity analysis detailed in Sect. 3.3.3, the primary complexity factors for non-fusion methods are k and r_j. In contrast, the fusion method's complexity is

also impacted by the number of leaves l. Therefore, performance degradation can be observed for the fusion method, but not for the non-fusion methods, as the number of leaves grows.

The performance degradation can be explained using Eq. (10). We focus on $\frac{k}{l}$ because the remaining terms can be disregarded with large $\sum_j r_j$. As l increases, $\frac{k}{l}$ decreases, leading to a reduced speedup compared to the non-fusion method. In Fig. 18, we examine the speedup concerning different values of k and l. The highest speedup occurs at the largest k and smallest l, which validates our complexity analysis that the speedup is correlated with $\frac{k}{l}$. A large k and small l suggest that the model functions as a data compressor, indicating that the fusion method can be advantageous when applying a narrow-down model to a large amount of data. From a hardware perspective, a pre-fusion method with a filtering effect actually reduces the size of input data, which further decreases memory usage and memory I/O in subsequent computations. Therefore, the value of $\frac{k}{l}$ can serve as a potential indicator for determining whether pre-fusion should be applied.

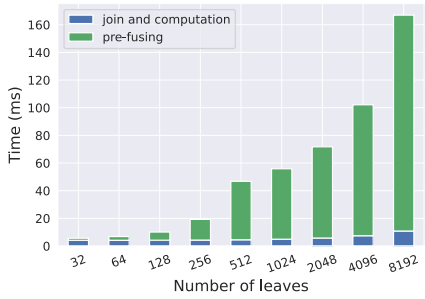


Fig. 20 The execution time that pre-fusing stage and join-computation stage take in prediction with decision tree after joining

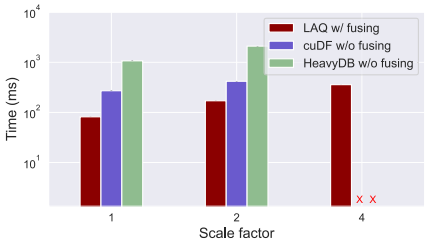


Fig. 21 Average execution time of predictive pipeline of GBDT regarding scale factors. The experimental scenario cardinality setting 1

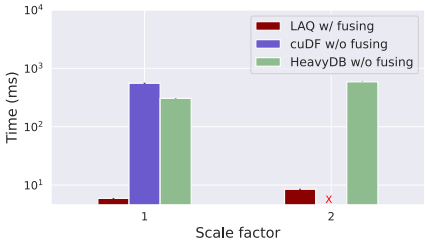


Fig. 22 Average execution time of predictive pipeline of GBDT regarding scale factors. The experimental scenario cardinality setting 2

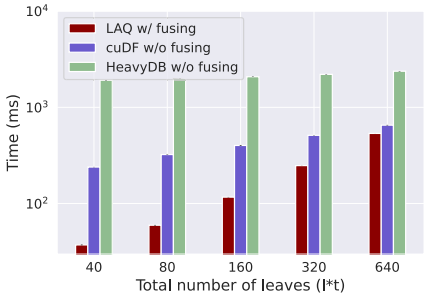


Fig. 23 Average execution time of predictive pipeline of GBDT regarding total number of leaves when sf = 2. The experimental scenario cardinality setting 1

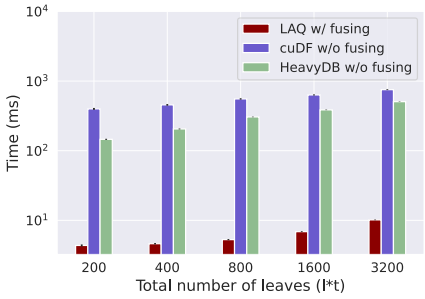


Fig. 24 Average execution time of predictive pipeline of GBDT regarding total number of leaves when sf = 1. The experimental scenario cardinality setting 2

Q_8 : *How much speedup can fusing decision trees deliver in cardinality setting* 2?

Observation and analysis In scenarios where dimension tables with small cardinality are processed using a large model, the operator fusion method exhibits a more significant speedup compared to the other three methods, as illustrated in Fig. 19. When the input scale factor is reduced to 1% of that in Q_8, the residual terms in Eq. (10) can no longer be ignored, leading to a greater speedup. However, as the model size increases, the cost of pre-operator fusion becomes more expensive relative to subsequent computations, as shown in Fig. 20. Considering that dimension tables are not entirely static but updated according to changes in the dimension tables, the actual benefits of the operator fusion method depend on the update frequency of the dimension tables.

5.3.4 Evaluation results for GBDT

Building upon the decision tree evaluation, we extend our evaluation to the more complex GBDT model. Given the extensive space complexity inherent to GBDT, our experiments focused on speedup metrics under relatively small scale factors.

Q_9 : *How much speedup can fusing GBDT deliver in different cardinality settings*?

In both cardinality settings, our operator fusion method demonstrates a marked superiority in performance compared to cuDF and HeavyDB. Notably, our method's efficiency is not only reflected in its speedup but also in its memory usage.

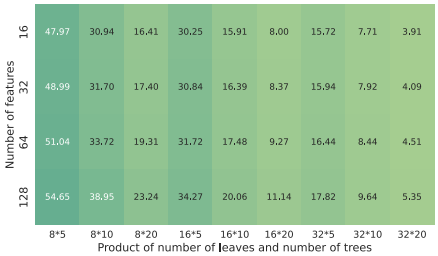


Fig. 25 Heatmap of speedups regarding input length and combinations of number of trees and leaves. The experimental scenario is cardinality setting 1 and sf=2

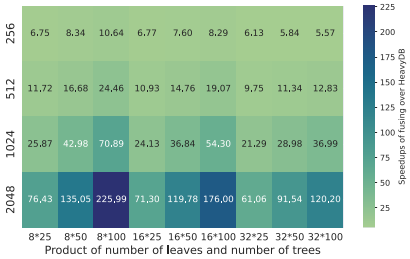


Fig. 26 Heatmap of speedups regarding input length and combinations of number of trees and leaves. The experimental scenario is cardinality setting 2 and sf=1

Figure 21 shows the execution time with cardinality setting 1. At a scale factor of 4, both cuDF and HeavyDB encountered out-of-memory errors, whereas our fusion method continued to operate effectively. This underscores our fusion method's capability in reducing memory footprint by avoiding redundancy from joins. When it comes to cardinality setting 2 (Fig. 22), speedup achieved by our fusion method becomes even more remarkable. Although the speedup presents a trend of decreasing when scale factor increases, the low memory footprint feature still make our method applicable in scenarios with limited resources.

cuDF performs worse in cardinality setting 2 compared to HeavyDB. This is due to its underlying data structure not being optimized for vast column manipulations. We had an extensive discussion of the reason that HeavyDB outperforms cuDF reversely in Q_5.

Q_{10} : *What are the dynamics of speedup in relation to varying input sizes and model scales ?*

As illustrated in Figs. 23 and 24, our findings for the GBDT models align with those observed with decision trees. Specifically, we noticed that, given a set k, the speedup generally decreases as the model complexity increases. This trend of speedup degradation is more significant in cardinality setting 1, characterized by large-scale input data and smaller models. However, it's crucial to understand that the total number of trees in a GBDT model doesn't straightforwardly correlate to the speed attained. This variation in performance is attributed to the computational efficiencies that are specific to the shapes of the tensors involved.

Q_{11} : *How does the implementation of GBDT affect speedups*?

Further insights are provided in Figs. 25 and 26, which illustrate the speedup relative to k and various combinations of tree and leaf counts. Beyond the general

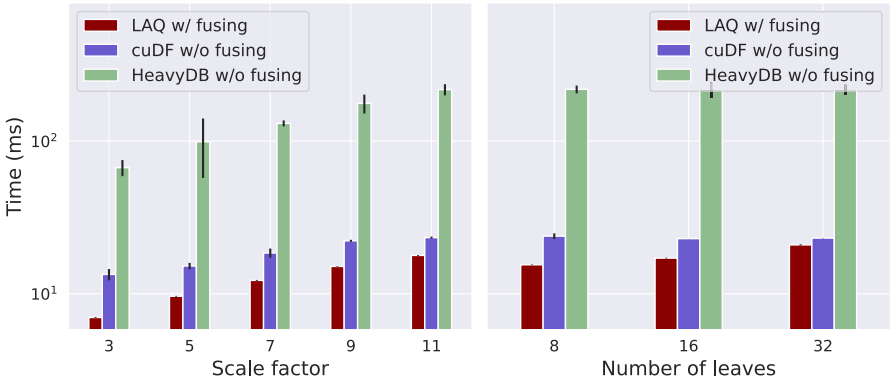


Fig. 27 Average execution time of decision trees in TPC-DI benchmark regarding scale factors and number of leaves

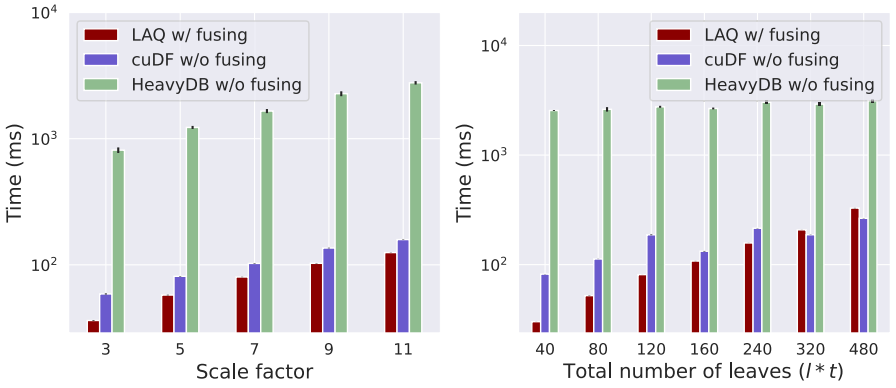


Fig. 28 Average execution time of GBDT in TPC-DI benchmark regarding scale factors and total number of leaves

trends, a particularly interesting observation arises in cardinality setting 2, especially when comparing models with an identical total number of leaves, such as configurations of 16 trees with 50 leaves each versus 8 trees with 100 leaves each. Models with a larger number of trees but fewer leaves per tree exhibit better performance.

As discussed in Sect. 4.4, this can be attributed to the potential for optimization in tensor computing when individual trees are smaller in size. Since the computational workload for a single tree doesn't fully utilize all available resources, the cuTensorNet library can efficiently parallelize multiple tensor computations within a single kernel launch. This explains why, even with the same $\frac{k}{t*l}$ ratio, a GBDT model with fewer leaves per tree and more trees overall performs faster. This phenomenon underscores the impact of tensor shape on actual computational complexity in our fused GBDT model implementation.

Summary Our analysis indicates that for a small k, larger GBDT models tend not to deliver faster results. However, it's important to consider the typical data

preprocessing steps involved in using GBDT models, such as vectorization or one-hot encoding. These steps often substantially increase the number of features, which is where our fusing method demonstrates its strengths. Consequently, this emphasizes the method's potential utility in practical scenarios, particularly in data-intensive fields where preprocessing expands feature space significantly, thereby aligning with the advantages of our fusion-based approach.

5.4 Performance evaluation on TPC-DI

In this experiment, we integrate our operator fusion approach into a real-world predictive pipeline, using data from the TPC-DI benchmark. The TPC-DI benchmark simulates real-world data integration scenarios using a realistic data model based on a retail brokerage firm. Therefore, we use TPC-DI to evaluate the performance of operator fusion in realistic data integration scenarios.

The dataset we used for evaluation involves four tables distributed across three disjoint parties, with the target table serving as the input for predictive tasks. We evaluate the performance of fusing both decision tree and GBDT models on this dataset, focusing on varying scale factors. Given our previous evaluations, which consistently showed the LAQ without fusing to be slower than our fusion approach, we have chosen to omit the results for LAQ without fusing.

Figure 27 presents the results of fusing decision trees. Consistent with patterns observed in synthetic datasets, the fusion method continuously outperforms cuDF in terms of speed. However, the speedup decreases with increasing scale factors. Notably, since k is only 27, the speedup is less significant than what we observe in synthetic data scenarios.

In the results for fusing GBDT, detailed in Fig. 28, we observe the same trend as in the fusing decision tree results. Specifically, when the total number of tree leaves reaches 480, our fusion method starts to be slower than cuDF without fusion. This is also attributed to the small k.

This experiment demonstrates the usability of our fusion method in real-world predictive pipelines, but it also highlights that the fusion method is not a one-size-fits-all solution. It is suitable particularly in predictive tasks with a large k, as discussed in Sects. 3.3 and 4, highlighting that while powerful, the operator fusion is most effective in specific types of data-intensive tasks.

6 Related work

GPU relational data processing GPU-accelerated query processing has been extensively researched in recent decades. As GPU architectural design and memory bandwidth between hosts and GPUs have advanced, several database management systems (DBMS) have incorporated GPU acceleration to optimize their query processing capabilities. Notable examples of GPU-based systems include Crystal, OmniSci (now known as HeavyDB) [12], BlazingSQL [3], and PG-Strom. These systems take advantage of the parallel processing capabilities of GPUs to perform

operations such as filtering, aggregation, and join processing at a significantly faster rate compared to traditional CPU-based systems.

However, these works do not change the nature of relational data processing. The theoretical and practical gap between relational data and linear algebraic input for machine learning still hinders potential integration and optimization opportunities.

Hummingbird [17, 19] is a system capable of compiling a wide range of traditional ML models into modern tensor-based runtimes designed specifically for deep learning models. In addition to providing a unified runtime, Hummingbird employs deep learning compilers to optimize the overall efficiency of the ML pipeline.

However, despite the benefits of tensor representation, the operators in Hummingbird do not implement joins and aggregations commonly found in data integration and training data generation processes.

Inspired by Hummingbird, TQP [11] further extends tensor programs for relational operations, including sort-merge join and hash join, enabling it to handle the full TPC-H benchmark [28]. TQP leverages a widely-used tensor computing runtime, to optimize and execute workflows containing both relational data processing and model prediction on GPUs. Following this research, TDP [8] expands capabilities to encode multi-modal data processing. Nevertheless, the physical implementation of join and aggregation operators remains in the relational style rather than LA. This diversity prevents the differentiability from being further pushed down to the source data before joins and also misses optimization opportunities brought about by LA rewriting. Our research implements joins and aggregations in linear algebra and proposes an operator fusion method leveraging this unified theoretical language, significantly accelerating predictive pipelines.

Query processing using matrix multiplication Matrix multiplication has been widely adopted in graph query processing. Earlier research [1, 6] proposed LA-based algorithms for computing an equi-join followed by a duplicate-eliminating projection, which yields smaller intermediate results and more efficient memory I/O than conventional relational operators. One recent paper [14] proposed *DIM3* to address several performance bottlenecks in [6]. *DIM3* introduces partial result caching and support for join-aggregation operations. However, this line of research focuses primarily on join operations rather than a general method of processing relational queries with linear algebra (LAQ) discussed in our research.

TCUDB [13] is the first GPU query engine that primarily uses LAQ as its query engine, which implements equi-join and single-column aggregation using LAQ. The design principle of the join-aggregation operator in TCUDB is similar to that of [1, 6], but it is embedded within a query planner that supports a wide range of SQL queries and analytic queries. In the TCUDB paper, the authors evaluate its performance with graph query workloads, but they do not provide insights into its performance into the cost of each operator in LAQ. In contrast, our work extensively evaluates LAQ on a wide range of data and reports detailed performance breakdown.

To support an integrated pipeline of data integration and ML model training, in our previous work [10], we have defined matrix-based representations for mapping columns and rows between source and target tables. With the logical representations, we identify the method to evaluate outer-join, inner-join, left-join, and union in data integration tasks using linear algebra operators. Building upon this

foundation, in the current research, we evaluate the extended LAQ using relational query benchmark datasets to assess its performance in traditional data queries and predictive pipelines.

Cross-optimization of ML and relational data processing Raven [22] and LaraDB [15] implemented cross-optimization methods for batch prediction tasks that follow relational data processing. The optimizer, built on a unified intermediate representation, enables the exchange of information between relational operators and ML models. However, in this research, the relational and linear components must execute in separate runtimes, which may involve potential data transformation and communication overhead. In contrast, our method unifies the data processing and ML model prediction in representation as well as runtime.

7 Conclusion and future research

In this paper, we present the operator fusion method to optimize the speed of predictive pipelines consisting of data processing and ML model predictions. The fundamental principle of LAQ involves fusing dimension tables with linear operators within a part of the model and caching the intermediate results. When executing queries with ML inference, the cached results serve as the input for the remaining part of the model, thereby reducing the execution time of queries with ML inference.

Furthermore, through the analysis of the complexity of operator fusion and LAQ without operator fusion, we find that the length ratio of input vector and output vector, described as $\frac{k}{l}$ as discussed in Sect. 3, may influence the speedup of our method in the context of the star schema. In our evaluation, we use SSB, TPC-DI, and a synthetic dataset to test the performance of LAQ and operator fusion. Based on the experimental results, we draw the following conclusions:

- LAQ outperforms cuDF, a standard GPU relational query processor, in most evaluations except for query group 4 when *sf* is 16. The inherent high computational complexity of domain construction and matrix multiplication dominates the execution time, causing performance degradation when data sizes increase. However, we can expect performance improvement by caching key domains.
- In experiments for predictive pipelines in Sect. 5.3, operator fusion exhibits significant speedups up to 317x compared to the traditional predictive pipeline without operator fusion. Moreover, the experiment results confirm the hypothesis that $\frac{k}{l}$ in Eqs. (3), (10) and 14 affects the speedup of operator fusion.
- The speedup of operator fusion also depends on the sizes of input matrices. Fusing large models is costly, but it can be beneficial when the update frequencies and cardinality of dimension tables are low. We need to make trade-offs between operator fusion and non-operator fusion based on update patterns and data sizes.

Limitations The performance advantage of operator fusion, as proposed in this paper, depends on size differences between dimension tables and the fact table. When the dimension table increases in size, the intermediate results from operator

fusion also expand. Consequently, the remaining portion of the model must process a substantially larger input, which may lead to memory-bound operations. Additionally, the residuals in the computational complexity become non-negligible, thereby reducing the efficacy of the LAQ method with fusion. Determining the decision boundary for this performance benefit calls for a more comprehensive and detailed performance model.

Future research Although we have preliminarily shown that fusing linear operators in ML models with LAQ is beneficial, a detailed cost estimation that can assist with automatic pipeline optimization is still missing.

Furthermore, in the context of thriving large-scale deep learning, more operator fusion rules that can optimize deep learning operators are urgently needed. The challenges of applying the operator fusion proposed in this work to deep learning models lie in the non-linear activation functions, such as sigmoid and tanh. Non-linear activations are not additive and therefore cannot be pushed down to dimension tables, as the final result of operator fusion is the summation of fused model-dimension tables. However, ReLU, which is widely used in deep learning models, can be regarded as a piecewise linear function. This characteristic has been successfully exploited for model inference and training over normalized data [5]. In future work, we will apply the operator fusion method to neural networks where ReLU is the activation function to expand the usability of our method.

Author contributions W. Sun wrote the main manuscript text. R. Hai and A. Katsifodimos helped reviewing and refining the content.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Amossen, R.R., Pagh, R.: Faster Join-Projects and Sparse Matrix Multiplications. In: ICDT 2009, pp. 121–126. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1514894.1514909
2. Balkesen, C., Teubner, J., Alonso, G., et al.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. ICDE **2013**, 362–373 (2013)
3. BlazingDB: BlazingSQL. https://github.com/BlazingDB/blazingsql (2020)
4. Chen, T., Moreau, T., Jiang, Z., et al.: TVM: an automated End-to-End optimizing compiler for deep learning. OSDI **2018**, 578–594 (2018)

5. Cheng, Z., Koudas, N., Zhang, Z., et al.: Efficient construction of nonlinear models over normalized data. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 1140–1151 (2021). https://doi.org/10.1109/ICDE51399.2021.00103
6. Deep, S., Hu, X., Koutris, P.: Fast join project query evaluation using matrix multiplication. SIGMOD **2020**, 1213–1223 (2020)
7. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Ann. Stat. **29**(5), 1189–1232 (2001). http://www.jstor.org/stable/2699986
8. Gandhi, A., Asada, Y., Fu, V., et al.: The tensor data platform: towards an ai-centric database system. In: CIDR 2023 (2023)
9. Ghiran, A.M., Buchmann, R.A.: The model-driven enterprise data fabric: a proposal based on conceptual modelling and knowledge graphs. In: Douligeris, C., Karagiannis, D., Apostolou, D. (eds.) Knowledge Science, pp. 572–583. Springer, Engineering and Management (2019)
10. Hai, R., Koutras, C., Ionescu, A., et al.: Amalur: data integration meets machine learning. In: ICDE 2023, p To appear (2023)
11. He, D., Nakandala, S.C., Banda, D., et al.: Query Processing on Tensor Computation Runtimes. vol 15, pp. 2811–2825. VLDB Endowment (2022)
12. Heavy.ai: HeavyDB. https://github.com/heavyai/heavydb (2022)
13. Hu, Y.C., Li, Y., Tseng, H.W.: Tcudb: accelerating database with tensor processors. SIGMOD **2022**, 1360–1374 (2022)
14. Huang, Z., Chen, S.: Density-optimized intersection-free mapping and matrix multiplication for join-project operations. vol 15, pp. 2244–2256. VLDB Endowment (2022)
15. Hutchison, D., Howe, B., Suciu, D.: LaraDB: a minimalist kernel for linear and relational algebra computation. In: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR'17 (2017)
16. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd edn. Wiley Publishing (2013)
17. Koutsoukos, D., Nakandala, S., Karanasos, K., et al.: Tensors: an abstraction for general data processing. Proc. VLDB Endow. **14**(10), 1797–1804 (2021)
18. Machado, I.A., Costa, C., Santos, M.Y.: Data mesh: concepts and principles of a paradigm shift in data architectures. Procedia Comput. Sci. **196**, 263–271 (2022)
19. Nakandala, S., Saur, K., Yu, G.I., et al.: A tensor compiler for unified machine learning prediction serving. In: OSDI 2020. USENIX Association (2020)
20. Okuta, R., Unno, Y., Nishino, D., et al.: CuPy: a NumPy-compatible library for NVIDIA GPU calculations. In: NIPS 2017 Workshop: LearningSys (2017)
21. O'Neil, P., O'Neil, E., Chen, X., et al.: The star schema benchmark and augmented fact table indexing. In: Performance Evaluation and Benchmarking: 1st TPC Technology Conference, TPCTC 2009, pp. 237–252. Springer (2009)
22. Park, K., Saur, K., Banda, D., et al.: End-to-end optimization of machine learning prediction queries. SIGMOD **2022**, 587–601 (2022)
23. Paszke, A., Gross, S., Massa, F., et al.: Pytorch: an imperative style, high-performance deep learning library. In: NeurIPS 2019 (2019)
24. Poess, M., Rabl, T., Jacobsen, H.A., et al.: Tpc-di: the first industry benchmark for data integration. Proc. VLDB Endow. **7**(13), 1367–1378 (2014). https://doi.org/10.14778/2733004.2733009
25. Psallidas, F., Zhu, Y., Karlas, B., et al.: Data science through the looking glass: analysis of millions of GitHub notebooks and ML.NET Pipelines. SIGMOD Rec **51**(2), 30–37. https://doi.org/10.1145/3552490.3552496 (2022)
26. Rapidsai: cuDF. https://github.com/rapidsai/cudf (2022)
27. Sun, W., Katsifodimos, A., Hai, R.: An empirical performance comparison between matrix multiplication join and hash join on GPUs. In: ICDE 2023 Workshop: HardBD & Activep (to appear) (2023)
28. Transaction Processing Performance Council TPC Benchmark H. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf (2018)
29. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms **1**(1), 2–13 (2005). https://doi.org/10.1145/1077464.1077466