Accelerating Machine Learning Queries with Linear Algebra Query Processing

Sun, Wenbo; Katsifodimos, Asterios; Hai, Rihan

DOI
10.1145/3603719.3603726

Publication date
2023

Document Version
Final published version

Published in
Scientific and Statistical Database Management - 35th International Conference, SSDBM 2023 - Proceedings

Accelerating Machine Learning Queries with Linear Algebra Query Processing

Wenbo Sun
w.sun-2@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Asterios Katsifodimos
a.katsifodimos@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Rihan Hai
r.hai@tudelft.nl
Delft University of Technology
Delft, The Netherlands

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 fusing 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 matrix multiplication query processing utilizing the widely-used Star Schema Benchmark. 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.

CCS CONCEPTS

• **Information systems-Query optimization**; • **Information systems-Join algorithms**; • **General and reference** → Performance;

KEYWORDS

database, query optimization, machine learning, operator fusion

ACM Reference Format:
Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. Accelerating Machine Learning Queries with Linear Algebra Query Processing. In *35th International Conference on Scientific and Statistical Database Management (SSDBM 2023), July 10–12, 2023, Los Angeles, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3603719.3603726

SSDBM 2023, July 10–12, 2023, Los Angeles, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0746-9/23/07.
https://doi.org/10.1145/3603719.3603726

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, 10] and ML predictions [4, 21] using modern hardware like Graphics Processing Units (GPU). Thanks to massive parallelism and LA-friendly hardware architectures, 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. A 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 these queries 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, 5, 11, 12, 24]. 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.*

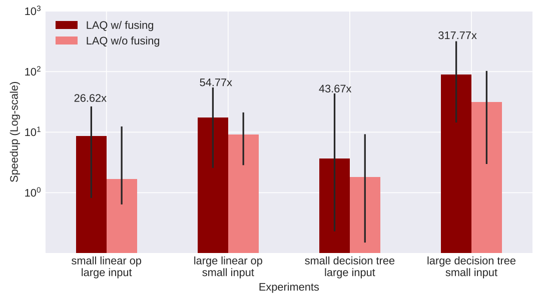


Figure 1: Speedups of our operator fusion method in four experimental predictive pipelines. The baseline is cuDF without operator fusion. The maximum attainable speedup is 317.77x.

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 [14] 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 Figure 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.
- We thoroughly evaluate LAQ using the widely-adopted SSB data processing benchmark [19] and report the performance comparison with cuDF [23] and HeavyDB [10], two popular GPU-accelerated data processing systems. Our evaluation helps demonstrate the potential of LAQ in handling traditional data query workloads and its effectiveness in the context of end-to-end predictive pipelines.

This paper is organized as follows: Section 2 introduces primary operators in LAQ. These building blocks are based on existing works [8, 11]. Following that, Section 3 presents two examples 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 Section 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 models. 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 the approach of processing relational queries with linear algebra (LAQ). As the preliminaries to our operator fusion method in Section 3, we implement the LAQ based on existing solutions. In particular, Section 2.1 and 2.2 elaborates selection and projection operator proposed in our earlier work [8]. Section 2.3 - 2.5 introduces MMJoin, group-by aggregation and sorting operators in TCUDB [11] and TQP [9].

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.

Table 1: Important notations in Section 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
$\mathbf{v}$	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

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* $M \in \{0,1\}^{c \times k}$ [8], 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:

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

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.

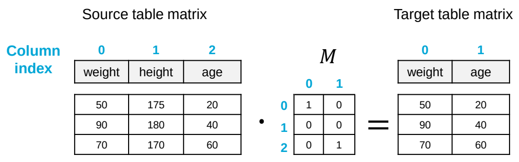


Figure 2: An example of projection as matrix multiplication.

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 achieve the function of selection with linear algebra. 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. However, this method may require an additional pass of column scan to generate the filter vector in advance, potentially making it less efficient than traditional relational selection. Thus, in our implementation, the multiplication with the filter vector is achieved using vectorized predicate 'AND' and memory copy rather than floating-point operations. In particular, if the input matrix has a row-dominated layout in memory, we select target row-pointers according to the filter vector and copy the selected rows to the target memory space. In our implementation, we use an out-of-box mask_select operator provided by CuPy [18].

2.3 MM-Join

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 [11]. Apart from the implementation details, we discuss the computational complexity of the MM-Join and hash joins. To ensure portability and compatibility with machine learning workloads, we implement this algorithm using CuPy.

Algorithm 1: Matrix Multiplication Join

```
   Input  : $R$,$S$: input relations
   Output : $I$: sparse matrix indicating maping rows in $R$ and $S$
1  key_domain=union($key_R$, $key_S$) //CUDA reduce
2  key_len=len(key_domain)
3  key_dict = dict(zip(key_domain range(0, key_len))
4  $rows_R$ = range(0, $r_R$), $rows_S$ = range(0, $r_S$)
5  $columns_R$ = 0, $values_R$ = 1
6  $columns_S$ = 0, $values_S$ = 1
7  for $i \in [0, r_R)$ do
8      $columns_R$[i] = key_dict[$key_R$[i]] //CUDA parallel
9  for $i \in [0, r_S)$ do
10     $columns_S$[i] = key_dict[$key_S$[i]] //CUDA parallel
11 $MAT_R$ = cuda_construct_CSR($rows_R$, $columns_R$, $values_R$)
12 $MAT_S$ = cuda_construct_CSR($columns_S$, $rows_S$, $values_S$)
13 $I$ = cuda_sparse_multiplication($MAT_R$, $MAT_S$).to_COO()
14 return $I$
```

2.3.1 2-way join. We illustrate the process of MM-Join with the pseudocode 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 maximum key value 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 get MAT_R and MAT_S (Lines 4-12), 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;

3) We execute sparse matrix multiplication over MAT_R and transposed MAT_S (Line 13);

4) The result I is a row matching matrix[2], defined as follows.

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

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 $O(n^2)$[3] [26], 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 the join.

Nevertheless, MM-Joins present an optimization opportunity that allows for the integration of linear operators in ML models with join processing.

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

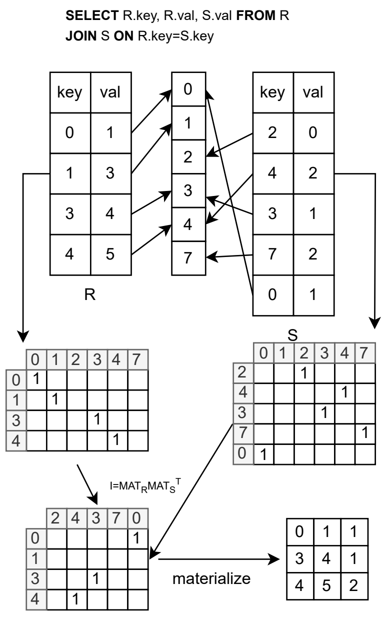


Figure 3: An illustration for evaluating equi-join with matrix multiplication.

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 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 Section 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:

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

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 $m =< 0, 1, 2, \ldots, nnz-1 >$. Consequently, we can construct two row mapping matrices I_R and I_S by aligning row indexes and column indexes with m, respectively.

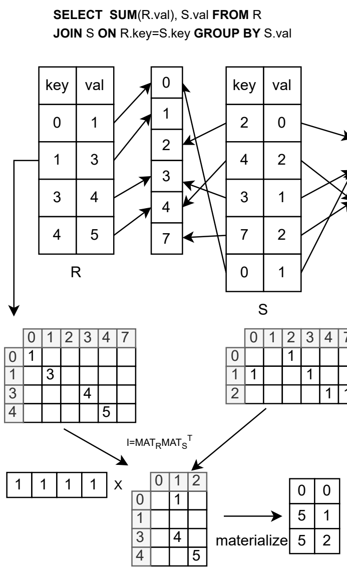

SELECT SUM(R.val), S.val FROM R
JOIN S ON R.key=S.key GROUP BY S.val
key
val
R
S
I=MAT_R MAT_S^T
X
materialize

Figure 4: An illustration for evaluating group-by-sum with matrix multiplication.

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 [11] and the multi-column aggregation inspired by TQP [9].

2.4.1 Aggregation by a single column. Figure 4 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 Figure 4, values 0, 1, 2 are found as groups. Then we find relationships between keys of S and the groups, which are $\{2\}- > 0$, $\{3,0\}- > 1$, $\{4,7\}- > 2$. After filling 1s 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 Figure 4, 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 [9], where unique tuples are identified using a sort-unique procedure.

2.5 Sorting

Sorting cannot be directly represented in LA, 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 LA operators introduced in Section 2, we analyze two operator fusion with two ML examples to illustrate the benefits: fusing linear operators (Section 3.2), and decision trees (Section 3.3).

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.

Operator Fusion. According to the design principles of star schema data warehouses discussed in [14], 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 two models to demonstrate how operator fusion can accelerate predictive pipelines.

3.2 Fusing Simple Linear Operators

Suppose the result of a star join $T \in \mathbb{R}^{i\times k}$ is computed through MM-Join according to keys in the fact table A. The evaluation can be presented as $T = I_1BM_1 + I_2CM_2 + I_3DM_3$, 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:

$$\begin{aligned}\text{predictions} &= TL \\ &= (I_1BM_1 + I_2CM_2 + I_3DM_3)L \\ &= I_1\underline{BM_1L} + I_2\underline{CM_2L} + I_3\underline{DM_3L}\end{aligned} \tag{1}$$

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 Section 3.3, *we omit the complexity of matrix additions.* Given the aforementioned dimensions of matrices, the computational complexity without operator fusion is:

$$\begin{aligned}C_{no-fusion} &= C_{mmjoin} + C_{op} \\ &= ck\sum_j r_j + ik\sum_j r_j + ikl \\ &= (ik + \frac{k^2}{3})\sum_j r_j + ikl\end{aligned}$$

If L is pushed 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 $DM_3L \in \mathbb{R}^{r_3\times l}$. The linear operator can be directly applied to these partial results. Then we have the computation complexity as:

$$C_{fusion} = il\sum_j r_j$$

Now, we compare two complexity values:

$$\begin{aligned}\frac{C_{non-fusion}}{C_{fusion}} &= \frac{(ik + \frac{k^2}{3})\sum_j r_j + ikl}{il\sum_j r_j} \\ &= \frac{k}{l} + \frac{k^2}{3il} + \frac{k}{\sum_j r_j}\end{aligned} \tag{2}$$

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 of dimension tables 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 Section 4.3, 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 [22]. 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 [17] in Section 3.3.1.

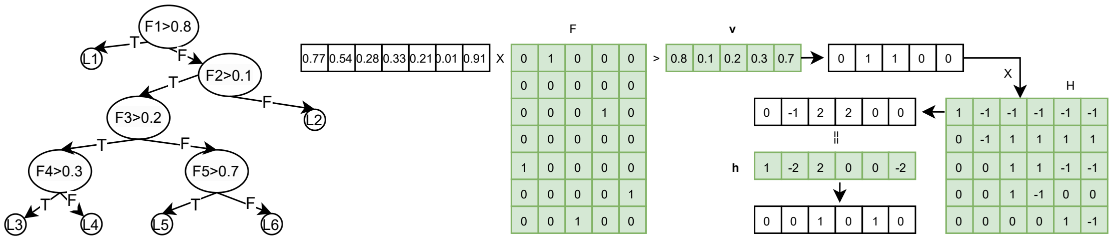


Figure 5: Prediction with a decision tree with linear algebraic representation adapted from [17].

3.3.1 *Matrix representation of Descion Trees.* Hummingbird [17] 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.

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 $\mathbf{h} \in \mathbb{R}^l$. Figure 5 illustrates how to use a sequence of linear and predicate operators to perform a prediction for decision trees. 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 integrated into the column mapping matrix M_* (described in Section 2.1), but we retain it in the rest of the implementation for completeness.

Step 2. Vector $\mathbf{v}$ 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 $\mathbf{v}$. 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 $\mathbf{h}$ is the column sum of matrix H. Before comparing with $\mathbf{h}$, the pre-fused matrices must be added to obtain a complete vector. The result vector is compared with $\mathbf{h}$, and a binary encoding denoting prediction labels will be produced.

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:

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

Because vectors in F are orthogonal, the result of TF can be interpreted as a linear combination of vectors in T. As a consequence, the predicate operator '$> \mathbf{v}$' can be partially evaluated. In contrast, the predicate operator '$== \mathbf{h}$' depends on the predecessor reduction operator H, which can not be partially evaluated. Therefore, we can push down $(TF > \mathbf{v})H$ to dimension tables and evaluate the predicate equal operator by summing up partial results. The process is expressed as follows:

$$\begin{aligned} predictions &= ((TF > \mathbf{v})H) == \mathbf{h} \\ &= (I_1((BM_1F > \mathbf{v})H) \\ &\quad + I_2((CM_2F > \mathbf{v})H) \\ &\quad + I_3((DM_3F > \mathbf{v})H)) == \mathbf{h} \\ &= (I_1T_1 + I_2T_2 + I_3T_3) == \mathbf{h} \end{aligned} \tag{3}$$

3.3.3 *Complexity Analysis.* Similar to the complexity analysis for the linear operator, we first present the complexity of non-fusion method:

$$\begin{aligned} C_{non-fusion} &= C_{mmjoin} + C_F + C_{\mathbf{v}} + C_H + C_{\mathbf{h}} \\ &= (\frac{k^2}{3} + ik)\sum_j r_j + ikp + ip + ipl + il \end{aligned}$$

With operator fusion presented in equation 3, we have three pre-fused matrices whose dimensions are $R^{r_i \times l}$. The complexity of remaining operations of decision tree's result can be expressed by:

$$C_{fusion} = il\sum_j r_j + il$$

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

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

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 $il >> 1$, we remove the constant term. Then, we have:

$$\begin{aligned} \frac{C_{non-fusion}}{C_{fusion}} &= \frac{\frac{k^2}{3}}{il} + \frac{ik}{il} + \frac{ik}{il\sum_j r_j} + \frac{ikl}{il\sum_j r_j} + \frac{1}{\sum_j r_j} \\ &= \frac{k}{l} + \frac{k^2}{3il} + \frac{k^2}{l\sum_j r_j} + \frac{k}{\sum_j r_j} + \frac{k}{l\sum_j r_j} + \frac{1}{\sum_j r_j} \end{aligned} \tag{4}$$

Due to the involvement of more linear operators in decision trees, additional tail terms appear in Equation 4. When the number of rows in dimension tables 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 Section 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 Section 4.3.

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.

Summary. In this section, we proposed an operator fusion method to accelerate predictive pipelines, building on the preliminaries introduced in section 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 Section 4.3, we will present the speedup of operator fusion in predictive pipelines.

It is important to note that two 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 [16], data fabric [7]) 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.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of LAQ and examine the performance enhancement achieved through operator fusion. In Section 4.2, we use the SSB [19] dataset to compare the performance of LAQ with two GPU-accelerated relational query processing engines. Then, based on LAQ, we assess the speedup of operator fusion through two models introduced in Section 3.2 and 3.3. Before presenting the experimental results, we first provide an overview of the experimental setup, encompassing the implementations under evaluation, dataset characteristics, and hardware.

4.1 Experiment settings

Implementation. In this paper, we implement GPU-accelerated LAQ using CuPy [18]. 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[4]. Although multi-group aggregation can be theoretically calculated through tensor multiplications, we opt to use the scatter_add operator due to the absence of a suitable vendor library for sparse tensor multiplications.

Baselines. To assess the performance of our LAQ implementation, we compare it with two other GPU-accelerated data processing libraries: HeavyDB [10] and cuDF [23]. 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.

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 (48GB) 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.

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

4.1.1 *Datasets.*

We use Star Schema Benchmark (SSB) [19] to investigate the performance in real-world workloads. On the other hand, considering the memory space required by linear operators in the operator fusion evaluation, we generate a synthetic dataset with down-scaled cardinality of SSB dataset. In the following parts, we introduce these two datasets and their data characteristics.

Star Schema Benchmark. 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 testing 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, Figure 6 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.

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. sf is a parameter controlling data sizes.

Tables	Type	Cardinality
lineorder	Fact	$sf * 6,000,000$
part	Dim	$200,000 * floor(1 + \log_2 sf)$
supplier	Dim	$sf * 2,000$
customer	Dim	$sf * 30,000$
date	Dim	$7 * 365$

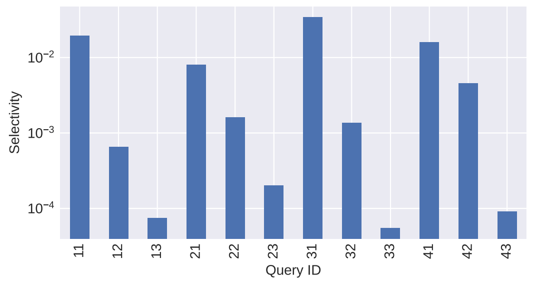


Figure 6: Selectivity of each query in SSB

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. We introduce two groups of cardinality settings to test the performance of operator fusion with varying numbers of input rows. 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

Table 4: Types and cardinalities of synthetic tables. sf is a parameter controlling data sizes.

Tables	Type	Cardinality setting 1	Cardinality setting 2
lineorder	Fact	$sf * 600,000$	$sf * 3,000$
part	Dim	$20,000 * floor(1 + \log_2 sf)$	$2,000 * floor(1 + \log_2 sf)$
supplier	Dim	$sf * 2,000$	$sf * 2,000$
date	Dim	$7 * 365$	$7 * 365$

Table 5: Parameters for linear operator and decision tree

Simple Linear Operator				
Cardinality	**sf**	**length of input (k)**	**length of output (l)**	
Setting 1	1,2,4,8,16	$2^{[4...7]}$	$2^{[1...7]}$	
Setting 2	1,2	$2^{[8...11]}$	$2^{[1...k]}$	
Decision Tree				
Cardinality	**sf**	**length of input (k)**	**# features (p)**	**# leaves (l)**
Setting 1	1,2,4,8,16	$2^{[4...7]}$	$2^{[4...7]}$	$2^{[1...6]}$
Setting 2	1,2	$2^{[8...11]}$	$2^{[3...11]}$	$2^{[6...11]}$

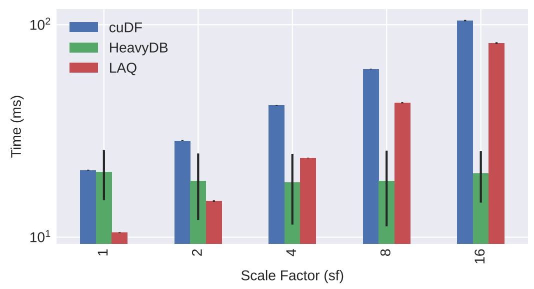


Figure 7: Average execution time under various scale factors.

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 Tables 5.

4.2 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.

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

Observation. Figure 7 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.

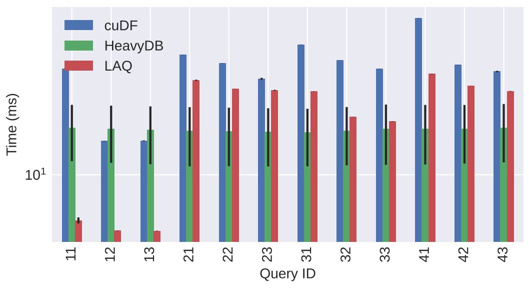


Figure 8: Average execution time of different queries when $sf = 4$.

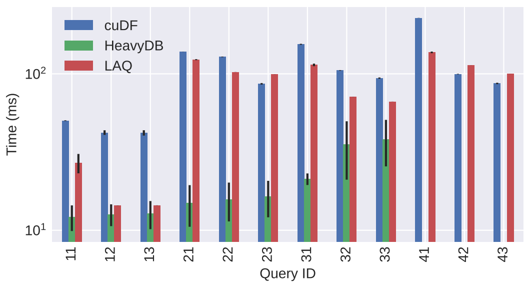


Figure 9: Average execution time of different queries when $sf = 16$.

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

Observation. Figure 8 and 9 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 $sf = 16$.

Figure 9 does not show the results of HeavyDB for query group 4 when sf=16 due to out-of-memory error raised during evaluation.

Analysis. Matrix multiplication (MM), which is the most crucial 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 Figure 6. 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 because of its cache management. In query group 3, HeavyDB shows performance degradation, which can be explained by the overhead of cache eviction due to increased intermediate results generated by joins.

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 10 shows the execution time of different operations in a query. Apparently, join operations dominates the execution time. In Figure 11, we further investigate two primary operations of joins. Domain generations take a similar portion of

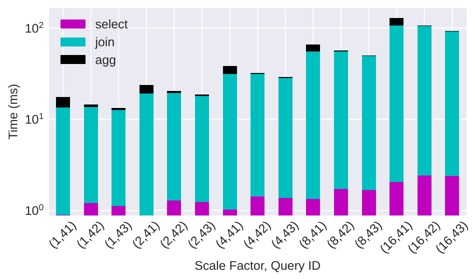


Figure 10: Performance breakdown for queries w.r.t different scale factors. We take query group 4 as an example.

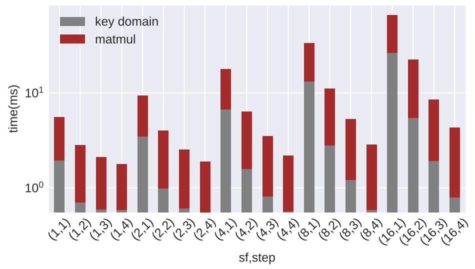


Figure 11: Performance breakdown for MM-Join w.r.t different scale factors. We take query group 4 as an example.

execution time within 4 join steps in query group 4, whereas join operations' portion decreases as the selectivity decreases.

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

4.3 Performance Evaluation for Operator Fusion

In this section, we evaluate operator fusion with linear operators in Section 3.2 and 3.3 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.

4.3.1 Simple Linear Operator. 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 operator fusion deliver in scenarios with cardinality setting 1 followed by a simple linear operator?*

Observation. In this experiment, we compare the execution time of a star join with operator fusion to LA-after-join implementations. It is important to note that HeavyDB is not included here, as it is slower by around 5 times of cuDF. Figure 12 displays the average execution time under various scale factors. The fusion method outperforms the other two implementations.

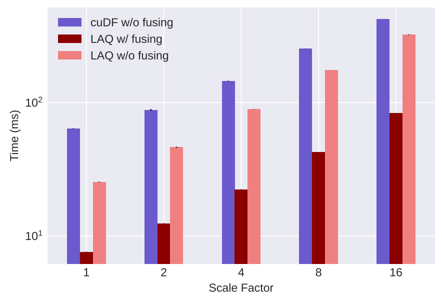


Figure 12: Average execution time of join with and w/o fusing linear operators under different scale factors.

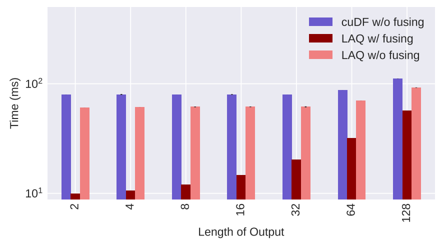
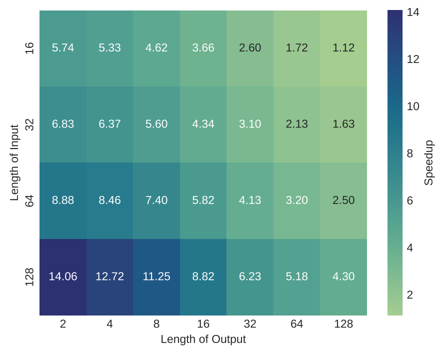


Figure 13: Average execution time of predictive pipeline of simple linear operator with and w/o operator fusion when $sf = 4$. The experimental scenario is large input with small model.



Length of Input \ Length of Output	2	4	8	16	32	64	128
16	5.74	5.33	4.62	3.66	2.60	1.72	1.12
32	6.83	6.37	5.60	4.34	3.10	2.13	1.63
64	8.88	8.46	7.40	5.82	4.13	3.20	2.50
128	14.06	12.72	11.25	8.82	6.23	5.18	4.30

Figure 14: Heatmap of speedup w.r.t lengths of input and output when $sf = 8$.

In Figure 13, 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.

Analysis. Through Equation 2, 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 Figure 13, 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 Figure 14. 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 Equation 2.

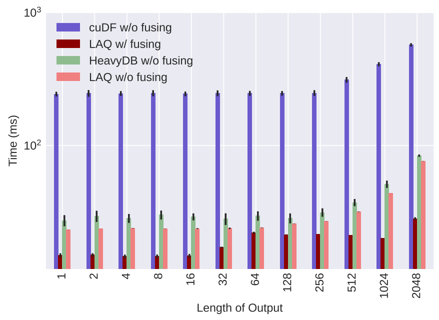


Figure 15: Average execution time of predictive pipeline of simple linear operator with and w/o operator fusion when $sf = 4$. The experimental scenario is small input with large model.

> Q_5 : *How much speedup can operator fusion deliver in scenarios with cardinality setting 2 followed by a large linear operator?*

Observation. In this experiment, we set the base cardinality to 1/1000 of SSB and enlarge the output shape l up to 2^{11}. Due to the size reduction of source tables, HeavyDB also exhibits comparable performance against others. Comparing Figure 15 and 13, we can clearly observe much more significant speedups of operator fusion in small dimension tables in cardinality setting 2.

Analysis. As indicated by Equation 2, 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.

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

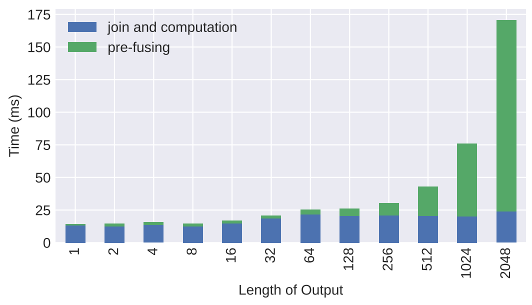


Figure 16: The execution time that pre-fusion stage and join-computation stage take in prediction with linear operator after joining.

Observation & 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 1. 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.

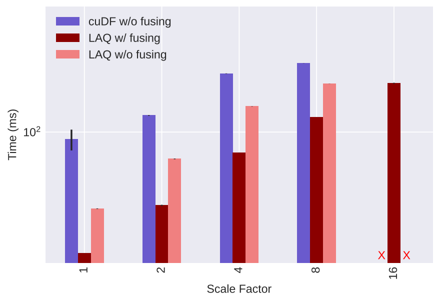


Figure 17: Average execution time of join with and w/o fusing decision trees under different scale factors.

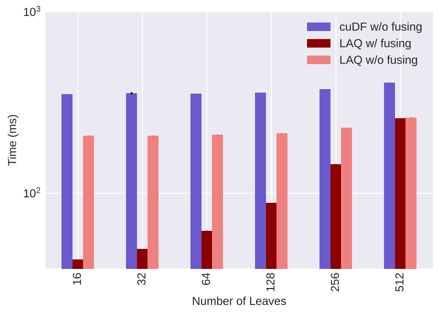


Figure 18: Average execution time of predictive pipeline of decision tree with and w/o operator fusion when $sf = 4$. The experimental scenario is large input with small model.

Figure 16 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.

4.3.2 Decision Tree. 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 operator fusion deliver in scenarios with cardinality setting 1 followed by a simple decision tree?*

Observation & Analysis. Figure 17-19 display the results for large input scenarios. Figure 17 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 Figure 18, 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.

The performance degradation can be explained using Equation 4. 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 Figure 19, 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

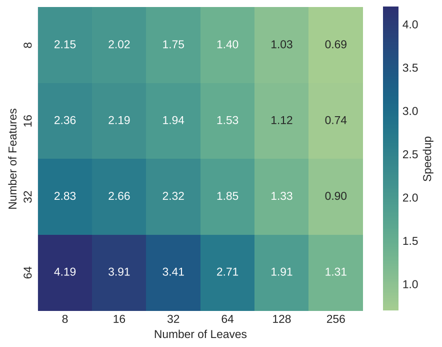


Figure 19: Heatmap of speedup w.r.t numbers of features and leaves when $sf = 8$.

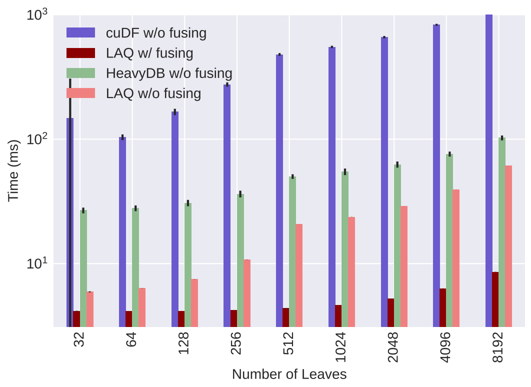


Figure 20: Average execution time of predictive pipeline of decision tree with and w/o operator fusion when $sf = 4$. The experimental scenario is small input with large model.

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.

Q_8 : How much speedup can operator fusion deliver in scenarios with cardinality setting 2 followed by a large decision tree?

Observation & 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 Figure 20. When the input scale factor is reduced to 1% of that in Q8, the residual terms in Equation 4 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 Figure 21. 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 RELATED WORK

GPU relational data processing. GPU-accelerated query processing has been extensively researched in recent decades. As GPU architectural design

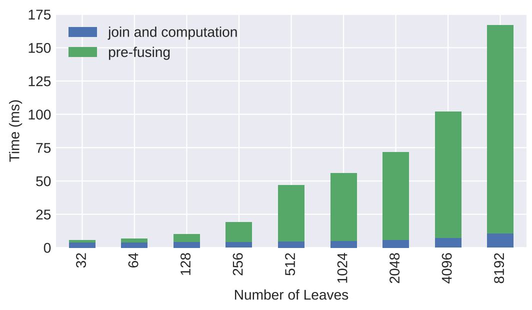


Figure 21: The execution time that pre-fusing stage and join-computation stage take in prediction with decision tree after joining.

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) [10], 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.

Query processing using matrix multiplication. Matrix multiplication has been widely adopted in graph query processing. Earlier research [1, 5] 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 [12] proposed *DIM3* to address several performance bottlenecks in [5]. *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 [11] 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] and [5], 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 [8], 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[20] and LaraDB [13] 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.

Hummingbird [15, 17] 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 [9] further extends tensor programs for relational operations, including sort-merge join and hash join, enabling it to handle the full TPC-H benchmark [25]. 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 [6] 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.

6 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. By employing LAQ to represent data query processing, our approach can merge operators in ML model predictions with data processing operators. 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 Section 3, may influence the speedup of our method in the context of the star schema. In our evaluation, we use a widely-adopted data query benchmark, SSB, 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 Section 4.3, operator fusion exhibits significant speedups up to 317x compared to the LAQ without operator fusion. Moreover, the experiment results confirm the hypothesis that $\frac{k}{l}$ in Equation 2 and 4 affect the speedup of operator fusion through.
- 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.

Future research. Based on the observations and analysis from the experiments, we identify that LAQ and optimizations in integrated data processing and ML pipelines call for further 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. Last but not least, exploring the optimization of training performance with similar linear algebraic operator fusion techniques is also a valuable research direction.

REFERENCES

[1] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT 2009* (St. Petersburg, Russia). Association for Computing Machinery, New York, NY, USA, 121–126. https://doi.org/10.1145/1514894.1514909

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE 2013*. 362–373.

[3] BlazingDB. 2020. BlazingSQL. https://github.com/BlazingDB/blazingsql.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 2018*. 578–594.

[5] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation Using Matrix Multiplication. In *SIGMOD 2020*. 1213–1223.

[6] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2023. The Tensor Data Platform: Towards an AI-centric Database System. In *CIDR 2023*.

[7] Ana-Maria Ghiran and Robert Andrei Buchmann. 2019. The Model-Driven Enterprise Data Fabric: A Proposal Based on Conceptual Modelling and Knowledge Graphs. In *Knowledge Science, Engineering and Management*, Christos Douligeris, Dimitris Karagiannis, and Dimitris Apostolou (Eds.). Springer International Publishing, 572–583.

[8] Rihan Hai, Christos Koutras, Andra Ionescu, Ziyu Li, Wenbo Sun, van Schijndel Jessie, Yan Kang, and Asterios Katsifodimos. 2023. Amalur: Data Integration Meets Machine Learning. In *ICDE 2023*. To appear.

[9] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11, 2811–2825.

[10] Heavy.ai. 2022. HeavyDB. https://github.com/heavyai/heavydb.

[11] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *SIGMOD 2022*. 1360–1374.

[12] Zichun Huang and Shimin Chen. 2022. Density-Optimized Intersection-Free Mapping and Matrix Multiplication for Join-Project Operations. *VLDB Endowment* 15, 10, 2244–2256.

[13] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. 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)*.

[14] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley Publishing.

[15] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An Abstraction for General Data Processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804.

[16] Inês Araújo Machado, Carlos Costa, and Maribel Yasmina Santos. 2022. Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures. *Procedia Comput. Sci.* 196 (2022), 263–271.

[17] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *OSDI 2020*. USENIX Association, Article 51.

[18] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *NIPS 2017 Workshop: LearningSys*.

[19] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009*. Springer-Verlag, 237–252.

[20] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *SIGMOD 2022*. 587–601.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS 2019*, Vol. 32.

[22] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, et al. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (2022), 30–37. https://doi.org/10.1145/3552490.3552496

[23] Rapidsai. 2022. cuDF. https://github.com/rapidsai/cudf.

[24] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. An Empirical Performance Comparison between Matrix Multiplication Join and Hash Join on GPUs. In *ICDE 2023 Workshop: HardBD & Active*. To appear.

[25] Transaction Processing Performance Council. 2018. TPC Benchmark H. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf.

[26] Raphael Yuster and Uri Zwick. 2005. Fast Sparse Matrix Multiplication. *ACM Trans. Algorithms* 1, 1 (2005), 2–13. https://doi.org/10.1145/1077464.1077466