ExBoost: Out-of-Box Co-Optimization of Machine Learning and Join Queries

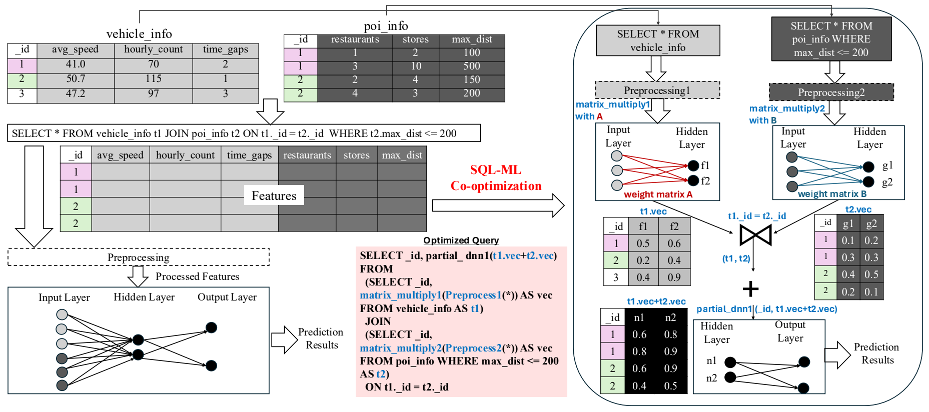
Kanchan Chowdhury, Lulu Xie, Lixi Zhou, Jia Zou ✉

Arizona State University
{kchowdh1,luluxie,lixi.zhou,jia.zou}@asu.edu

Abstract. Many real-world machine learning (ML) inference workflows involve data integration steps that join multiple data silos to assemble feature vectors. Such `join` operations are often expensive and easily become bottlenecks in the end-to-end inference process. We argue that factorizing ML computations and pushing down factorized computations through `join` can effectively accelerate the end-to-end processing of such queries. However, existing work on factorized ML focuses on the learning processes of traditional ML models and the computation of separate linear algebra operators, with two drawbacks. First, they never considered and evaluated the factorization of the end-to-end deep neural network inference process. Second, existing optimizers did not consider the relative overheads of the decomposable part and non-decomposable part and the alignment of the cost estimation for the SQL processing and ML processing. To close the gaps, we (1) demonstrate an out-of-box AI/ML-SQL co-optimization approach for end-to-end inference workflows where the users specify a SQL query, preprocessing operators, and a pre-trained model exported in ONNX format, and the end-to-end processing will be automatically optimized and synchronized in our ExBoost system; and (2) propose a contrastive learning approach to automatically estimate the potential speedup by applying the factorization strategy.

1 Introduction

It is well-known that many data science pipelines are bottlenecked by `join` operations with high-cardinality and high-dimensional inputs/output [26,13]. While there exist many works [22][12][7][21][13][15] on factorizing the AI/ML processing and pushing down the factorized computations through `join`, these works suffer from three limitations. (1) They focus on the factorization of the learning process and the separate linear algebra operators. They utilize the iterative nature of the training process and reduce redundant computations by eliminating the need of denormalizing the underlying relations. However, they did not discuss how to factorize the end-to-end AI/ML inference pipelines containing arbitrary deep neural network architectures and common preprocessing operators, which are user-facing, non-iterative, and latency-critical. (2) The benefits of factorizing a given workload depend on many factors, such as the CPU and I/O costs of the ML operators and relational operators, as well as the relative size



vehicle_info

_id	avg_speed	hourly_count	time_gaps
1	41.0	70	2
2	50.7	115	1
3	47.2	97	3

poi_info

_id	restaurants	stores	max_dist
1	1	2	100
1	3	10	500
2	2	4	150
2	4	3	200

```sql
SELECT * FROM vehicle_info t1 JOIN poi_info t2 ON t1._id = t2._id  WHERE t2.max_dist <= 200
```

_id	avg_speed	hourly_count	time_gaps	restaurants	stores	max_dist
1						
1						
2						
2						

```sql
SELECT _id, partial_ dnn1(t1.vec+t2.vec)
FROM
  (SELECT _id,
matrix_multiply1(Preprocess1(*)) AS vec
FROM vehicle_info AS t1)
  JOIN
  (SELECT _id,
matrix_multiply2(Preprocess2(*)) AS vec
FROM poi_info WHERE max_dist <= 200
AS t2)
  ON t1._id = t2._id
```

```sql
SELECT * FROM
vehicle_info
```

```sql
SELECT * FROM
poi_info WHERE
max_dist <= 200
```

t1.vec

_id	f1	f2
1	0.5	0.6
2	0.2	0.4
3	0.4	0.9

t2.vec

_id	g1	g2
1	0.1	0.2
1	0.3	0.3
2	0.4	0.5
2	0.2	0.1

t1.vec+t2.vec

_id	n1	n2
1	0.6	0.8
1	0.8	0.9
2	0.6	0.9
2	0.4	0.5

Fig. 1: Illustration of SQL-ML Co-Optimization Enabled by ExBoost

of the factorizable part to the end-to-end model. Although one of the existing works [13] provides a cost model to estimate the factorization benefits, it only considers the join operators and the cost of multiplying features with regression coefficients, failing to capture other components in the end-to-end inference pipeline such as multilayer neural network models, non-join part of the SQL query, and preprocessing operators. (3) Existing in-database factorization systems implemented a limited set of specialized ML operators on top of the database system and require users to program the entire ML pipelines using UDF-centric SQL statements, while today's users prefer programming with Python and specialized ML systems such as Pytorch and TensorFlow.

Distinguished from existing works, this work focuses on an end-to-end inference pipeline consisting of data joining, preprocessing operators, and multilayer neural networks, as illustrated in Fig. 1, and explained below:

Feature-Decomposable ML Operators. A computing module f is **feature-decomposable** for a set of n features combined from k datasets, e.g., $\mathbf{x} = \mathbf{x_1}^\frown\mathbf{x_2}^\frown...^\frown\mathbf{x_k}$, if it could be independently calculated on each dataset without interactions: $f(\mathbf{x}) = \delta(g_1(\mathbf{x_1}), g_2(\mathbf{x_2}), ..., g_k(\mathbf{x_k}))$. Here, $\mathbf{x_1}, ..., \mathbf{x_k}$ have the same number of tuples and $^\frown$ represents concatenation, and $\mathbf{x_i}$ represents a subset of features. δ is a simple operation that aggregates the intermediate results, having negligible computing cost compared to g_i usually. Taking a fully connected (FC) layer with the weight matrix W as an example, the layer will convert the n input features $\mathbf{x}$ into m hidden features or outputs using $\sigma(\mathbf{x} \times W^T + b)$, where σ represents the activation function such as Relu. W can be partitioned into k submatrices corresponding to the k subsets of features, so that the i-th partition W_i has a shape of $m \times d(\mathbf{x_i})$, where $d(\mathbf{x_i})$ represents the number of features in the dataset $\mathbf{x_i}$. W_i represents the weight parameters at all edges that connect the features in $\mathbf{x_i}$ and all m output features. Then the matrix multiplication can be decomposed so that $\mathbf{x} \times W^T = \Sigma_{i=1}^{k}\mathbf{x_i} \times W_i^T$.

Factorization and Push-Down. For a feature-decomposable ML operator f, if the input features are joined from k datasets using join predicates $p_1, ..., p_{k-1}$, i.e., $\mathbf{x} = \mathbf{x_1} \bowtie_{p_1} \mathbf{x_2} \bowtie_{p_2} ... \bowtie_{p_{k-1}} \mathbf{x_k}$, $f(x)$ could be independently calculated on each dataset, i.e., $f(\mathbf{x}) = \delta((\mathbf{y_1} = g_1(\mathbf{x_1})) \bowtie_{p_1} (\mathbf{y_2} = g_2(\mathbf{x_2})) \bowtie_{p_2} ... \bowtie_{p_{k-1}} (\mathbf{y_k} = g_k(\mathbf{x_k})))$. Here, $\bowtie_{p_i}$ represents a join using the predicate p_i. Following the previous example of f being an FC layer, each component $\mathbf{x_i} \times W_i^T$ can be pushed down to the dataset $\mathbf{x_i}$. The output is denoted as $\mathbf{y_i}$. If $\mathbf{x_i}$ has n tuples, W_i^T has a dimension of $d(x_i) \times m$ (i.e., $d(x_i)$ and m are defined earlier), $\mathbf{y_i}$ has a shape of $n \times m$. Then, we join the output datasets $\mathbf{z} = \mathbf{y_1} \bowtie_{p_1} \mathbf{y_2} \bowtie_{p_2} ... \bowtie_{p_{k-1}} \mathbf{y_k}$, and for each joined tuple $z = (y_1 \in \mathbf{y_1}, ..., y_k \in \mathbf{y_k})$, $y_1 + ... + y_k$ will be computed as output y'. Finally, the remaining operations in f will be applied to y'. The potential benefits of such factorization and push-down include:

• If $m << \Sigma_{i=1}^{k} d(\mathbf{x_i})$, the `join` dimensions will be significantly reduced without affecting the accuracy of the output.

• If $|z| >> \Sigma_{i=1}^{k} |\mathbf{x_i}|$, where $|\mathbf{x_i}|$ represents the number of tuples in dataset $\mathbf{x_i}$, a significant amount of redundant matrix multiplication computations will be avoided without affecting the accuracy.

In this work, we provide a library, called ExBoost, for dynamically implementing such co-optimizations. Its API takes in an SQL query (for data processing), an ML application in a supported format, such as the ONNX [5], which runs inferences over the output of the SQL query, and optional preprocessing operators as input. Under the hood, it analyzes the graph-based intermediate presentations (IRs) of the ML application and the SQL query to detect the decomposable ML operators and whether these decomposed operators can be co-optimized with join operators. It will also estimate the speedup brought by the co-optimization, which is a "challenging" task due to the diversity of the workloads, the lack of training data, and the disparate nature of relational and ML operators.

We address these challenges using a two-step approach. The first step adopts a technique similar to contrastive learning [8] to reduce the data labeling overhead and leverage the attention mechanism to accurately estimate the ratios of the latencies of the joins and the decomposable ML operators to the latency of the end-to-end workflow. To bridge the cost estimation gap between the relational and ML operators, the second step estimates the speedup by forming a novel knowledge representation, which is sent to a decision forest model for predicting the speedups. If the estimated speedup meets user-specified requirements, this optimization technique will be applied by rewriting the workflow automatically, leveraging IRs and user-defined functions (UDFs).

The proposed library can be applied to many use scenarios, such as data science workflows prototyped in Python scripts or Jupyter Notebooks, where developers invoke our API to gain speedups for workflows containing the identified `join=>(preprocessing)=> inference` patterns.

To summarize, our main contributions are as follows: (1) We investigate the factorization and push-down of the deep neural network models through the `join` operators to reduce the end-to-end latency when the feature vectors for inferences are joined from multiple datasets. (2) We provide a novel ExBoost library that automatically determines whether the proposed co-optimization strategy would benefit a given workload by unifying graph-based ML and SQL intermediate representations and adopting a two-step learning-based approach for unified ML and SQL cost estimation. (3) Finally, we conduct an experimental evaluation on synthetic and real-world query workloads to demonstrate the effectiveness of the proposed ExBoost approach. Our evaluation results show that our proposed approach can reduce runtime by up to **17.7**×.

2 User Interface

An example of using our ExBoost library is illustrated in Listing 1.1. The user is required to provide the path to a model exported to any supported format, such as ONNX IR [5], and a SQL query, as illustrated in `line 15` of the example code in Listing. 1.1. The ExBoost library also provides a set of optional preprocessing operators that can be pushed down to each feature of the underlying data, including range scaler [4], discretization operator [11], binarization operator, standardization scaler, power transformer [27], quantile transformer [10], and several commonly used missing value imputation functions. The identifiers and parameters of these preprocessing operators that need to be applied to the output of the SQL query in order can be represented in a JSON string, used as input to the `CoOptimizer.run()` API, as illustrated in `line 18` in Listing 1.1.

```
 1 from exboost import sql.DBConfig
 2 from exboost import CoOptimizer
 3 
 4 if __name__ == '__main__':
 5     model_path = "data/model_dnn_2000_64_2.onnx"
 6     table_name1 = 'input1'
 7     table_name2 = 'input2'
 8 
 9     sql_query = f"SELECT * FROM {table_name1} INNER JOIN {
           table_name2} ON {table_name1}.col_0 > {table_name2}.
           col_0"
10     ## To access user tables
11     DBConFigureset_db_config(username='user1', password='
           12345', hostname='localhost', port='5432',
           database_name='myDB')
12 
13     ## Method for co-optimization w/o preprocessing operators
14     result1 = CoOptimizer.run(sql_query=sql_query, model_path
           =model_path)
15 
16     ## Method for co-optimization w/ preprocessing operators
17     result2 = CoOptimizer.run(sql_query=sql_query,
           preprocessors={"operator_name": "MinMaxScaler", "
           parameter": {}}, model_path=model_path)
```

Listing 1.1: Illustration of ExBoost Library Usage

The ExBoost API will analyze the SQL query, the preprocessing operators, and the AI/ML application or model to perform the co-optimization and execute the co-optimized query plan by invoking external libraries, such as DuckDB [19] APIs, ONNX [5] APIs, and Scikit-learn APIs (for preprocessing), while returning the results in data frame format.

3 Query analyzer

The analyzer parses the input SQL query, the preprocessing operators, and the model to graph-based intermediate representations (IRs) to facilitate the detection of the proposed co-optimization opportunity and feature extraction for unified cost estimation. For SQL queries, we adopted the graph-based relational algebra representation as IR and used the Python wrapper of the existing SQL parsing library libpg_query [3] to convert the query into a relational algebra tree. It traverses the tree to find the presence of the join operators and the number of data silos involved in the query. In case of the presence of a join operator, the analyzer looks for the availability of any non-join operators that follow the join operators, but cannot be pushed down, termed push-down breakers, such as aggregation operators that involve columns from multiple data silos. If push-down breakers exist between the joins and the decomposable ML operators, the input query and the ML model cannot be co-optimized.

Similarly, the unified query analyzer checks whether any preprocessing operators exist and whether these operators can be performed before the join processing. If the results are positive, the analyzer further parses the ML model and complicated feature processing into a computational graph IR. For example, in our implementation, we adopted the ONNX IR for fully connected neural network models and other neural network models. Within the graph representation of the ONNX model, the analyzer traverses the graph, starting from the first operator that processes the input data (i.e., a root node).

Many works have discussed the factorization of linear algebra operators [15] that are used in Deep Neural Networks (DNNs) such as matrix multiplication and matrix addition. We found other feature-decomposable operators which include but are not limited to decision forest (i.e., leveraging the QuickScorer algorithm [16]), convolutional neural network (i.e., if data is partitioned across the channels), nearest-neighbor search using product quantization, and chained/nested AI/ML models that require inferences over features combined from the inference results of multiple AI/ML models over separate feature sets. This work will focus on DNN inference (as illustrated in Fig. 1) and feature preprocessing operators, of which the factorization mechanisms are first explored by this work. (1) **Range Scalers** such as MaxAbsScaler and MinMaxScaler [4]. The former scales the values of each feature into a specific range, and the latter scales the values of each feature into a range defined by the minimal value and maximum value among all the values in the feature. Such feature-specific operators can be directly pushed down to each dataset (i.e., a subset of features). (2) **Discretization [11] and Binarization Operators.** For example, threshold-based binarization is widely used to present a high correlation with the target label. If a value is higher than a threshold, it will be transformed to 1, otherwise it will be transformed to 0. Such feature-specific transformations can be directly pushed down. (3) **Encoding Categorical Values.** Encoding functions such as one-hot and ordinal encoding are applied to separate features and can be pushed down directly. (4) **Constant-based Feature Imputation.** Imputing features with constant values can be directly pushed down. (5) **Standardization Scaler.** The standardization scaler transforms each value x in a feature by subtracting the feature's mean value (μ) and scales it by dividing the standard deviation (σ), i.e., $\frac{x-\mu}{\sigma}$. For 1 to 1 join operators that will not change the total number and distribution of values in the feature, such scalarization operators can be pushed down to the dataset that contains the feature column, before it is joined with other datasets. For n to m joins, based on a synopsis counting the predicate selectivity for each distinct value, it can compute the μ and σ of each feature after the join processing so that the standardization scaler can be pushed down to each data silo. Denoting distinct values of a column as v_i, frequency of the value v_i in a data silo as $freq_i$, and join selectivity ratio of v_i as s_i, we can redefine the mean and standard deviation as $\mu = \frac{\sum(v_i * freq_i * s_i)}{\sum(freq_i * s_i)}$ and $\sigma = \sqrt{\frac{\sum((v_i-\mu)^2 * freq_i * s_i)}{\sum(freq_i * s_i)}}$. The standardization scaler can be pushed down to each data silo with the updated values of mean and standard deviation. Similarly, we also support the push-down of **PowerTransfomer** [27], **QuantileTransfomer** [10], and feature imputation based on mean, median, and most frequent values for 1 to 1 join and/or m to n joins with profiled synopsis that describes the selectivity ratio of each distinct value.

If an operator node belongs to the decomposable ML operators, the node will be pushed into a stack, and each child of the node will be examined in a similar way recursively. Otherwise, the path from the corresponding root node to this node, representing a sequence of decomposable ML operators, if not empty, will be recorded and combined with the subquery starting from the last join operator that has a path to the root node (identified by traversing the relational algebra of the SQL query from the root node), called a co-optimizable subgraph.

Extracting Features The query analyzer also extracts the features from the SQL and the ML IRs to facilitate cost estimation. The features related to the SQL part include but are not limited to the number of columns in each join input and output, the estimated cardinality of each join input and output, the number of data silos, the number of rows and features in each data silo, the ratio of output cardinality to input cardinality, and whether the user's database supports transfer acceleration, e.g., using ConnectorX [25]. The ML model features include but are not limited to the type, the shape of the input, the number of model parameters, the output shape, and the number of floating-point operations (FLOPs) of each decomposable ML operator in the co-optimizable subgraph.

4 Cost model

Unified cost estimation (i.e., predicting the improvement in latency brought by ExBoost co-optimization) is challenging due to the workload diversity and the disparate natures of ML and relational operators. To address the challenge, we integrate (1) a novel neural network architecture with contrastive learning and attention mechanism to estimate the ratios of the latencies of the co-optimizable subgraph and its join operators to the latency of the entire query; and (2) a novel knowledge representation for estimating the speedup of co-optimization.

4.1 Learning-based Assessment of Processing Bottleneck

Should the latency attributable to the co-optimizable subgraph constitute merely a minor fraction of the total end-to-end latency, co-optimization will hardly lead to any speedups. Therefore, we first estimate the ratio of the latency of these parts to the end-to-end latency. We divide the end-to-end pipeline into two parts. The first part is the co-optimizable subgraph, denoted as the white-box component. On the other hand, the rest of the query, including all remaining ML and relational algebra nodes, is termed a black-box component. We aim to estimate the fractions of the end-to-end latency attributed to the entire white-box component and the join operations in the white-box, respectively. These two ratios will compose part of the knowledge representation as described in the symbolic step (Section 4.2).

The ratio of the joining latency to the end-to-end latency is termed by $R_{join}^{total} = \frac{C_{join}}{C_{wb}+C_{bb}}$ where C_{join}, C_{wb}, C_{bb} represent the latencies for executing the join operator, white-box component, and black-box component, respectively. We define the ratio of white-box latency to the end-to-end latency as $R_{wb}^{total} = \frac{C_{wb}}{C_{wb}+C_{bb}}$. Although the estimations of these ratios can be formalized as regression models consisting of deep neural networks, it is challenging due to the limited availability of labeled training data.

To alleviate the problem, we propose an idea similar to contrastive learning [8]. Taking the estimation of R_{wb}^{total} as an example, the neural network architecture consists of two input networks to predict the costs of the white- and black-boxes respectively, and then compute the ratio R_{wb}^{total} using these two costs. Given a pair of white- and black-box components *<wb, bb>*, with measured running latency $\hat{t_{wb}}$ and $\hat{t_{bb}}$ respectively, we use two separate networks to predict the latencies of the white-box component and the black-box component, denoted as t_{wb} and t_{bb}. Then, the predicted ratio R_{wb}^{total} is estimated as $R_{wb}^{total} = \frac{t_{wb}}{t_{wb}+t_{bb}+\epsilon}$, while $\hat{R_{wb}^{total}} = \frac{\hat{t_{wb}}}{\hat{t_{wb}}+\hat{t_{bb}}}$ stands for the ground truth label. The constant ϵ in $\frac{t_{wb}}{t_{wb}+t_{bb}+\epsilon}$ prevents the divide by zero error during the initial epochs of training.

Each input network model takes the corresponding features detailed at the end of Section 3 as the input, which is first passed to an attention layer to capture cross-feature correlations and the relative importance of the features. The attention score for each feature is between 0 and 1, so that the summation of all scores is 1. The attention layer also performs an element-wise multiplication between attention scores and the feature vector. The output is sent to a fully connected (FC) layer with ReLU activation. Then, the output is further concatenated with the input features and sent to multiple FC layers to get the final predicted cost. The output costs from both networks (t_{wb} and t_{bb}) are used to compute R_{wb}^{total} as formulated earlier. We define the loss in terms of the Huber loss as indicated in Equation 1 to balance Mean Absolute Error (MAE) and Mean Squared Error

(MSE), where $_\delta$ is a Huber loss parameter to be tuned during the training process.

$$L_\delta(R_{wb}^{total}, \hat{R_{wb}^{total}}) = \begin{cases} \frac{1}{2}(R_{wb}^{total} - \hat{R_{wb}^{total}})^2 & \text{for } |R_{wb}^{total} - \hat{R_{wb}^{total}}| \le \delta \\ \delta|R_{wb}^{total} - \hat{R_{wb}^{total}}| - \frac{1}{2}\delta^2 & \text{for } |R_{wb}^{total} - \hat{R_{wb}^{total}}| > \delta \end{cases} \quad (1)$$

To prepare the labeled training data, we first prepare two sets of samples for white- and black-box components, respectively, with the runtime recorded by running each component. Then these two sets of samples are applied with a cross-product to form a large set of training examples. We model the ratio R_{join}^{total} similarly, based on combinations of the samples of join queries and the samples of the rest of the query graph to form a relatively large training dataset.

4.2 Knowledge Representation for Estimating the Speedup

We define the speedup by the co-optimization of a co-optimizable subgraph as $R_{speedup} = \frac{L_{regular}}{L_{decompose}}$ where $L_{decompose}$ and $L_{regular}$ represent the end-to-end latencies incurred by the decomposed pipeline and regular pipeline, respectively. A larger value of $R_{speedup}$ means a high speedup, while a value smaller than 1 indicates a slowdown instead. The system applies the proposed co-optimization strategy if $R_{speedup}$ exceeds a user-defined threshold. We mainly use a symbolic approach based on a knowledge representation with six key factors for estimating the speedup:
Factor 1: The reduction in the volume of data to be joined (R_J), brought by the co-optimization strategy. Denoting the estimated join cardinality and the number of columns in the original and the decomposed join as $|z|$, d_{in}, and $d_{decompose}$ respectively, $R_J = \frac{\alpha * |z| * d_{decompose}}{\alpha * |z| * d_{in}}$, which simplifies to $R_J = \frac{d_{decompose}}{d_{in}}$, where α represents a joining latency factor.
Factor 2: The reduction in the latency of the decomposable part of the model (R_I), brought by the co-optimization strategy. Without co-optimization, the decomposable model part, denoted as M_{dc}, is executed against $|z|$ samples. On the other hand, a decomposed strategy runs M_{dc} for $\Sigma_{i=1}^{i=k}|\mathbf{x_i}|$ samples where $|\mathbf{x_i}|$ represents the number of tuples in data silo $\mathbf{x_i}$. The latency of executing M_{dc} can be modeled by $\beta * S * FP$, with β, S, and FP, representing the inference latency factor, #samples and FLOPs (floating point operations) of M_{dc} respectively. Therefore, R_I can be represented as $R_I = \frac{\Sigma_{i=1}^{i=k}|x_i| * FP_{dc}^i}{|z| * FP_{dc}}$, where FP_{dc} is the total FLOPs in the model M_{dc} and FP_{dc}^i stands for FLOPs in the decomposed part of model M_{dc} corresponding to the data silo x_i.
Factor 3 and 4: R_{join}^{total} and R_{wb}^{total} are obtained by the neural approach in Section 4.1. A high value of the ratio R_{join}^{total} means that the execution time of the join operators constitutes a significant portion of the end-to-end latency. Therefore, optimization of the join operations may significantly accelerate the execution of the pipeline. Moreover, a higher value of the ratio R_{wb}^{total} indicates that we have the opportunity to reduce the end-to-end latency of M_{dc} as discussed in Factor 2.
Factor 5 and 6: The total number of columns across all data silos (D), denoted as $D = \Sigma_{i=1}^{i=k}|\mathbf{d_i}|$; and cardinality of the join output ($|z|$). The intuition is based

on the observation that if the total number of input features D is small, the reduction in join latency is not significant, irrespective of the value of the ratio R_J. Similarly, a lower value of $|z|$ reveals that the latencies of executing the join operators and the decomposable model M_{dc} are already small, and the effects of co-optimizing these operators will be trivial.

Finally, we modeled the speedup using the knowledge representation of $(R_J, R_I, R_{join}^{total}, R_{wb}^{total}, D, |z|)$. We trained a random forest model and created the training and testing samples from the pairs we previously created for contrastive learning (Section 4.1).

The details of the contrastive learning model and the random forest model, and their training processes, can be found in Section 6.2.

5 Query rewriter

Once the cost model recommends co-optimization, the query rewriter will rewrite the IRs. First, it rewrites the relational algebra IR to pull up join operations as top as possible using existing rewriting rules [6,14]. Second, it rewrites the co-optimizable subgraph's IR (e.g., ONNX IR). In our implementation, a decomposable ONNX model, M, is first split into decomposable (M_{dc}) and non-decomposable (M_{ndc}) parts. Then, M_{dc} is further decomposed into k models: $M_{dc}^1, M_{dc}^2, ...M_{dc}^k$, where k stands for #data silos in the input query. We implemented a set of user-defined functions (UDFs) in DuckDB and Velox. Each UDF implements a decomposed or non-decomposable component or an aggregation function for the supported models. Taking a fully connected neural network as an example, we implemented three UDFs to support its decomposition and push down. The first UDF implements matrix multiplication by invoking LibTorch's corresponding API, and it can be parameterized at run-time by supplying a weight matrix and its shape. Then, the UDF is invoked from an SQL query to run matrix multiplication between each input vector (or a batch of input vectors) and the weight matrix. The second UDF implements the non-decomposable part of the model by invoking LibTorch's APIs for matrix addition, ReLU, matrix multiplication, softmax, etc, where the number of layers and the weight matrix for the rest of the layers could be configured when constructing the UDF. The third UDF transforms each joined tuple into an aggregated format by invoking the matrix addition from the LibTorch library, e.g., t1.vec + t2.vec in Figure 1.

During the rewriting processes, these functions are automatically selected and parameterized for the decomposable and non-decomposable parts identified by the query analyzer. Then these selected UDFs are nested with the rewritten SQL query to compose the final query, as illustrated in the pink box of Figure 1. If a decomposable neural network model is identified, but there are no corresponding UDFs for the decomposable and non-decomposable parts, we can leverage ONNX IR to automatically generate such functions. ONNX supports converting a subgraph into an ONNX model, which can be converted to PyTorch or TensorFlow models utilizing the libraries such as onnx2torch [9].

The supported preprocessing operators also have corresponding UDFs predefined by invoking Scikit-Learn functions. Such functions will be selected by parsing the JSON string (as illustrated in Listing 1.1), and nested with the SQL query in a way that they were pushed down to the data silos.

By executing the rewritten query in the corresponding UDF-centric database environments (i.e., in this paper, DuckDB is used for most workloads), data from a silo x_i (loaded from an external database as a Pandas dataframe using ConnectorX [25] if possible) is sent to preprocessing UDFs and a UDF encapsulating the model M_{dc}^{i} to generate the intermediate results. These intermediate results will be joined, and further transformed into the final prediction results.

6 Experimental evaluation

The experimental evaluation focuses on several questions: **R1.** How effective is the proposed ExBoost co-optimization strategy for different types of AI/ML applications? **R2.** How does the effectiveness change with the varying data distribution across datasets to be joined? **R3.** How accurate are the proposed cost ratio and speedup estimations? Since our factorization strategy converts the input model into an equivalent one that generates similar output, the prediction/classification accuracy of a factorized model is exactly similar to that of the corresponding input model. Therefore, we do not need to evaluate the prediction/classification accuracy of the factorization strategy.

6.1 Experimental Environment

AI/ML applications. We considered classification tasks using FC neural networks with different numbers of layers and different neurons at each layer.

Workloads. In this evaluation we mainly considered two synthetic workloads and one real-world workload. Each workload consists of one `join` query and one or more AI/ML applications that consume the query output. **(1) Synthetic Workloads (Epsilon and Bosch).** The Epsilon dataset is a well-known dataset from the Pascal data challenge [24], with $2,000$ features and 0.5 millions of tuples. The Bosch dataset [17] is a well-known manufacturing dataset that represents measurements of parts as they move through Bosch's production lines, having 968 features and 1.18 millions of tuples. We vertically split both datasets into 2, 5, and 10 parts. The SQL queries in the workloads will join these parts based on the similarity of a shared feature column after min-max scalarization. **2. Real-world Workload (NYC).** This workload consists of an SQL query that prepares features by preprocessing and joining the NYC block-level earnings dataset [2] (D1) and the NYC education dataset [1] (D2). D1 and D2 contain $108,487$ and $2,216$ records, respectively. We extracted 56 and 47 features from D1 and D2, respectively, after removing identification-related information and performing constant-based feature imputations. Both datasets include geographical coordinates denoted as *coord*, which were utilized to join D1 and D2 using a complicated join predicate `euclidea_distance(D1.coord,` `D2.coord) <` t, and the threshold t was later adjusted in the experiments to manage the output cardinality of the join queries.

System Environments We used an Ubuntu Linux machine with 48 CPU cores (Intel Xeon CPU E5-2687 3.00GHz) and 32GB memory size. For all workloads, the input data is stored in PostgreSQL 15. We use the ConnectorX 0.2.3 library to fetch data as a Pandas dataframe (Pandas version 2.2).

6.2 Optimizer Implementation Details

Contrastive Learning for Assessment of Processing Bottleneck Training Data Preparation. To determine the ratios of the join processing and the processing of the decomposed ML to the end-to-end inference workflow latency, we prepare the training and test data for contrastive learning. We generated 250 unique end-to-end inference workflows using different combinations of white-box architectures (i.e., `join` query and decomposable part of the ML model) and black-box architectures (i.e., non-decomposable part of the ML model). We record the latencies corresponding to joins, white- and black-box components for these 250 executions in files. In addition, we also extract and record the feature vectors corresponding to the white- and black-box components in separate files. After removing duplicate feature vectors, we get 159 unique feature vectors of white-box components and 118 of the black-box component. We split these feature vectors into 80% training and 20% test sets. Leveraging the idea of contrastive learning, after pairing vectors corresponding to the white box and black box component with cross products, we get 11938 training pairs and 768 test pairs. We ensure that feature combinations used for testing in our experimental evaluation fall under the set of test pairs.

Model Architecture and Training Details. In our implementation, the network has 7 fully connected (FC) layers consisting of 64, 128, 256, 128, 64, 32, and 1 output neuron respectively. Each FC layer is followed by a ReLU activation function. Each of the second, third, fourth, and fifth activation layers follow a dropout of 0.4, 0.5, 0.4, and 0.4, respectively. We apply ReLU activation after the last FC layer to force the predicted latencies to be positive. We train the model for 150 epochs with Adam optimizer using a learning rate of 0.00005.

Estimation of the Speedup Based on the Proposed Knowledge Representation Training Data Preparation. We leverage the 250 inference workflows prepared in Section 6.2 to extract features following our proposed knowledge representation and implement the proposed model decomposition and push-down technique to these workflows to obtain the measured speedup.

Model Architecture Details. By performing random sampling on different combinations of the number of trees and depth of trees, we found that the model performs the best for 200 trees with a max depth of 20, which we used for our cost estimation task.

6.3 Baselines

We compare the following approaches in terms of memory consumption and runtime requirements. (For all of these approaches, the underlying join algorithm (hash join or sort-merge join) is carefully selected for each test case to achieve the lowest latency.)

– **Regular Join**: This baseline simply joins multiple data silos based on the equality of the join key, and each joined record includes all features.
– **Index Join**: This is an optimized join strategy for dimension reduction [18], in which each joined record only includes the IDs of the joined tuples from both data silos and the joining key. Later, the feature vector is lazily created from the tuples indexed by their IDs, during the inference process.
– **ExBoost**: This approach implemented our proposed co-optimization strategy based on model decomposition and push-down as illustrated in Figure 1.

It should be noted that existing factorization approaches [22,12,21,13] do not support our target inference workloads containing multilayer neural networks.

6.4 R1. Effectiveness of ExBoost

The evaluation results of ExBoost on the synthetic and real-world workloads are reported in Tables 1. Both synthetic datasets, Epsilon and Bosch, are vertically partitioned into two splits with equal number of features respectively (we will discuss the case of using 5 and 10 partitions in Section 6.5). For the evaluation with various neural networks, when varying the number of neurons (N) in the split layer (i.e., the layer that separates the decomposable ML operators from the rest of the ML operators), cardinality ratio (R, i.e., ratio of the output cardinality to the input cardinality of join processing) is kept fixed to 10, while N is set to 1024 for evaluating various R values.

As illustrated in Table 1, ExBoost achieved up to 8× speedup compared to the regular join on Epsilon, Bosch, and NYC datasets. It also exhibits 2.65× speedup compared to the index join on these datasets. Our approach with the varying number of neurons at the split layer of the ML model, consistently outperforms other baselines with significant latency reduction, while the performance gain slowly improves as the size of the split layer increases. Results on various cardinality ratio indicate that the decomposed join performs significantly better as the cardinality ratio increases (which is the usual case for most join scenarios). Besides, ExBoost outperforms regular join even when the cardinality ratio is 1. Although the datasets (D1 and D2) in the NYC workload have fewer #features than Epsilon and Bosch datasets and the input cardinality of D2 is significantly low, yet the speedup achieved by ExBoost is similar to that of synthetic workloads. It indicates that ExBoost is effective regardless of the combinations of input cardinalities and input #features, as long as the cardinality of the join query is higher than the total cardinalities of input datasets.

Table 1: Evaluating Elapsed Time (Milliseconds) of Neural Networks

	Method	Neurons in split layer (N)				Ratio of cardinality (R)				
		N=64	N=128	N=512	N=1024	R=1	R=5	R=10	R=15	R=20
Epsilon	Regular Join	118,216	128,001	177,432	236,926	20,255	85,084	236,926	357,019	541,200
	Index Join	40,780	44,393	61,013	81,579	**18,878**	44,128	81,579	117,055	176,294
	ExBoost	**22,910**	**24,663**	**32,980**	**43,393**	19,665	**28,842**	**43,393**	**53,207**	**68,067**
Bosch	Regular Join	134,914	140,732	193,814	255,400	24,810	100,241	255,400	373,100	564,224
	Index Join	47,220	49,055	67,227	88,846	**21,958**	48,894	88,846	123,096	186,902
	ExBoost	**25,945**	**26,806**	**35,759**	**46,170**	23,360	**31,342**	**46,170**	**55,200**	**70,529**
NYC	Regular Join	97,832	107,467	152,807	213,988	20,016	69,801	213,988	300,429	389,608
	Index Join	31,604	34,653	50,130	71,607	**16,760**	38,172	71,607	94,571	123,173
	ExBoost	**20,130**	**21,932**	**30,199**	**41,632**	17,121	**27,266**	**41,632**	**47,763**	**52,864**

6.5 R2. Varying Data Partitions (Silos)

To assess the effectiveness of ExBoost on datasets with more than two data silos, we partition the features of the Epsilon and Bosch datasets into five distinct groups with equal number of features. From each data silo, we choose one column as the joining key. Table 2 presents the latency comparisons across different methods using datasets partitioned into five silos. Unlike two-silo experiments, the cardinality ratio (R) is kept fixed at 20 instead of 10 while varying the number of neurons (N) during the evaluation with neural networks. The results indicate that the ExBoost's decomposed join approach outperformed regular and index join baselines, achieving a speedup of up to 9× and 2.18× respectively. In addition, speedup tends to increase as the ratio of output cardinality to input cardinality rises. Note that the ratios of output cardinalities to input cardinalities are higher in the five-silo datasets compared to the two-silo experiments because the output cardinality increases with each join operation.

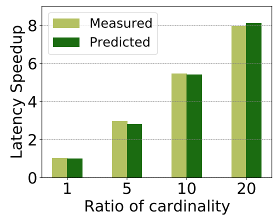
Impact of Input Cardinality Distributions. We partition the features of the Epsilon dataset into 10 distinct groups, each having an equal number of features. To evaluate the impact of input cardinality distributions on the speedup of the proposed approach, we create three versions of this 10-silo workload by varying the number of records in various silos such that it follows three different distributions: uniform, normal [23], and fat tail [20]. For all three distributions, we keep the total number of records combining all 10-silos equal, which is $100k$. Table 3 shows a comparison among three types of distributions in terms of the end-to-end execution time (milliseconds) where parameters N and R are kept similar to 5-silo experiments. Although end-to-end latency for the same baseline differs from one distribution to another (because joining latency varies significantly across distributions), ExBoost's decomposed join always outperforms the other two baselines. For a higher ratio of output cardinality to input cardinality, the proposed method can outperform regular join and index join baselines by up to 17.7× and 1.9×, respectively.

Table 2: Elapsed Time (Milliseconds) of Neural Networks with 5 silo datasets

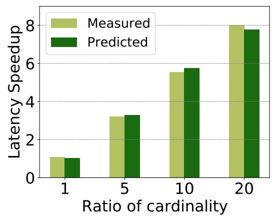
	Method	Neurons in split layer (N)				Ratio of cardinality (R)				
		N=64	N=128	N=512	N=1024	R=1	R=10	R=20	R=40	R=60
Epsilon	Regular Join	1,722,786	1,797,079	1,981,414	2,081,700	909,185	1,561,562	2,081,700	4,402,580	7,881,382
	Index Join	320,481	335,098	368,559	386,300	**94,060**	196,042	386,300	983,650	1,949,888
	Exboost	**238,944**	**247,873**	**270,685**	**287,765**	95,220	**134,592**	**287,765**	**524,400**	**893,184**
Bosch	Regular Join	2,350,348	2,396,388	2,542,048	2,700,111	900,055	1,145,138	2,700,111	4,930,205	8,515,354
	Index Join	431,237	442,637	477,631	501,518	**161,281**	278,399	501,518	1,044,626	2,019,326
	Exboost	**315,483**	**321,361**	**340,529**	**360,299**	167,318	**222,104**	**360,299**	**614,904**	**941,964**

Table 3: Impact of Input Cardinality Distribution on a 10 silo Dataset

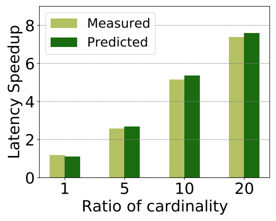
	Method	Neurons in split layer (N)				Ratio of cardinality (R)				
		N=64	N=128	N=512	N=1024	R=1	R=10	R=20	R=40	R=60
Uniform Distribution	Regular Join	273,778	276,530	311,946	327,470	14,492	165,210	327,470	677,797	1,166,274
	Index Join	36,853	39,526	55,296	71,683	5,095	34,758	71,683	143,042	210,780
	ExBoost	**35,267**	**35,560**	**37,985**	**39,775**	**4,361**	**19,960**	**39,775**	**72,324**	**116,971**
Normal Distribution	Regular Join	283,191	329,201	420,524	473,023	34,015	147,481	473,023	1,035,922	1,367,417
	Index Join	38,310	44,729	57,883	67,737	6,580	34,798	67,737	143,059	173,534
	ExBoost	**19,645**	**22,190**	**30,092**	**34,941**	**5,843**	**19,959**	**34,941**	**65,369**	**97,744**
Flat Tail Distribution	Regular Join	798,285	831,109	906,501	950,450	40,805	432,023	950,450	2,233,557	3,092,676
	Index Join	118,626	125,723	139,345	149,631	82,533	113,991	149,631	228,894	298,079
	ExBoost	**88,203**	**93,121**	**106,026**	**113,112**	**81,329**	**96,074**	**113,112**	**149,757**	**174,028**



(a) Epsilon-cardinality



(b) Bosch-cardinality



(c) NYC-cardinality

Fig. 2: Evaluating the cost model by comparing the predicted and the measured speedup

6.6 R3. Optimizer and Cost Estimation

To address R3, we compare the speedup of the ExBoost's decomposed join strategy as estimated by our optimizer's cost model (Section 4) to the measured speedup with different configurations of the ML operators, such as the number of neurons at the split layer of linear models and the varying ratio of cardinality of the join operator. As illustrated in Figure 2, we find the values predicted by the cost model are very close to the actual speedups, within $\pm 5\%$ of the actual ratios for more than 93% of testing cases, which proves the effectiveness of our novel cost model.

Table 4: Ablation Study of the Optimizer

Ratio	**Method**	**MSE**	**MAE**
R_{wb}^{total}	W/O Attention	0.0021	0.0258
	W/ Attention	0.0014	0.0182
R_{join}^{total}	W/O Attention	0.0021	0.0273
	W/ Attention	0.0016	0.0192

(a) Impact of Attention

Method	**MSE**	**MAE**
W/O R_{join}^{total} and R_{wb}^{total}	0.4202	0.3422
W/O R_{join}^{total}	0.0314	0.1313
W/O R_{wb}^{total}	0.0432	0.1478
W/ R_{join}^{total} and R_{wb}^{total}	0.0259	0.1274

(b) Impact of R_{join}^{total} and R_{wb}^{total}

Effectiveness of Attention Mechanism in the Assessment of Processing Bottleneck. In Section 4.1, we proposed an attention-based neural network to predict what fraction of the end-to-end latency is constituted by the joining and the entire white box component, which were denoted as R_{join}^{total} and R_{wb}^{total}, respectively. To evaluate the effectiveness of the attention mechanism in predicting R_{join}^{total} and R_{wb}^{total}, we also predict these ratios by skipping the attention layer. We report the Mean Squared Error (MSE) and Mean Absolute Error (MAE) of these two networks (without attention and with attention) in Table 4. Based on the results, it is evident that the attention mechanism can reduce the errors of predicting R_{wb}^{total} and R_{wb}^{total} by up to 33% and 30%, respectively. The results of this study prove that the attention mechanism can capture the relative importance of various features which results in a lower prediction error.

Effectiveness of R_{join}^{total} and R_{wb}^{total} in Estimating Speedup. We study the impact of joining latency to end-to-end latency ratio (R_{join}^{total}) and white-box latency to end-to-end latency ratio (R_{wb}^{total}), proposed in Section 4.1, in estimating the speedup achieved by ExBoost. Based on the prediction errors (MSE and MAE) reported in Table 4, we can state that the model performs the worst without R_{join}^{total} and R_{wb}^{total}. Appending any of these two ratios reduces the prediction error significantly, and we achieve the best performance in the presence of both ratios, a reduction of 94% in MSE and 63% in MAE, which validates the effectiveness of R_{join}^{total} and R_{wb}^{total} in estimating the speedup.

7 Conclusion

In this work, we explore the co-optimization of the join queries and ML inference operations in data science pipelines, where features for ML inference are integrated by performing joins between various data silos. We identify a set of decomposable ML operators and propose a novel optimizer that automatically determines whether the proposed model decomposition and push-down strategy (i.e., ExBoost) can accelerate the end-to-end pipeline execution using a neural-symbolic cost model and automatically rewrite the data science workflow to apply the co-optimization. Evaluation results on popular ML datasets show that the proposed co-optimization strategy can speedup the execution by up to 17.7×.

References

1. Nyc education (2000). `https://geodacenter.github.io/data-and-lab/NYC-Census-2000/` (2000), accessed: 2024-03-10
2. Block-level earnings in nyc (2002-14). `https://geodacenter.github.io/data-and-lab/LEHD_Data/` (2014), accessed: 2022-02-05
3. Pypi libpg query. PyPI (2017), `https://pypi.org/project/pg_query/0.16/`
4. de Amorim, L.B., Cavalcanti, G.D., Cruz, R.M.: The choice of scaling technique matters for classification performance. Appl. Soft Comput. **133**(C) (2023)
5. Bai, J., Lu, F., Zhang, K., et al.: Onnx: Open neural network exchange. `https://github.com/onnx/onnx` (2019)
6. Chaudhuri, S., Shim, K.: An overview of cost-based optimization of queries with aggregates. IEEE Data Eng. Bull. **18**(3), 3–9 (1995)
7. Chen, L., Kumar, A., Naughton, J., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (Aug 2017). `https://doi.org/10.14778/3137628.3137633`, `https://doi.org/10.14778/3137628.3137633`
8. Chen, T., Kornblith, S., Norouzi, M., Hinton, G.: A simple framework for contrastive learning of visual representations. ICML'20, JMLR.org (2020)
9. developers, E., Kalgin, I., Yanchenko, A., Ivanov, P., Goncharenko, A.: onnx2torch. `https://enot.ai/` (2021), version: x.y.z
10. Gilchrist, W.: Statistical modelling with quantile functions (01 2000)

11. Han, J., Kamber, M., Pei, J.: Data mining concepts and techniques, third edition (2012)
12. Huang, Z., Sen, R., Liu, J., Wu, E.: Joinboost: Grow trees over normalized data using only sql. Proc. VLDB Endow. **16**(11), 3071–3084 (Jul 2023). https://doi.org/10.14778/3611479.3611509, https://doi.org/10.14778/3611479.3611509
13. Kumar, A., Naughton, J., Patel, J.M.: Learning generalized linear models over normalized data. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1969–1984 (2015)
14. Levy, A.Y., Mumick, I.S., Sagiv, Y.: Query optimization by predicate move-around. In: VLDB. vol. 94, pp. 12–15 (1994)
15. Li, S., Chen, L., Kumar, A.: Enabling and optimizing non-linear feature interactions in factorized linear algebra. In: Proceedings of the 2019 International Conference on Management of Data. pp. 1571–1588 (2019)
16. Lucchese, C., Nardini, F.M., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. p. 73–82. SIGIR '15 (2015)
17. Mangal, A., Kumar, N.: Using big data to enhance the bosch production line performance: A kaggle challenge. In: 2016 IEEE international conference on big data (big data). pp. 2029–2035. IEEE (2016)
18. Psaropoulos, G., Legler, T., May, N., Ailamaki, A.: Interleaving with coroutines: a practical approach for robust index joins. Proceedings of the VLDB Endowment **11**(2), 230–242 (2017)
19. Raasveldt, M., Muehleisen, H., et al.: Duckdb. In: Proceedings of the 2019 International Conference on Management of Data. ACM (2019)
20. Roh, H.: Understanding fat-tailed distribution. https://towardsdatascience.com/journey-to-tempered-stable-distribution-part-1-fat-tailed-distribution-958d28bc20c (2020)
21. Schleich, M., Olteanu, D., Abo Khamis, M., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: Proceedings of the 2019 International Conference on Management of Data. p. 1642–1659. SIGMOD '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3299869.3324961, https://doi.org/10.1145/3299869.3324961
22. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: Proceedings of the 2016 International Conference on Management of Data. p. 3–18. SIGMOD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2882903.2882939, https://doi.org/10.1145/2882903.2882939
23. Siegel, A.F.: Chapter 7 - random variables: Working with uncertain numbers. In: Siegel, A.F. (ed.) Practical Business Statistics (Sixth Edition), pp. 155–186. Academic Press, sixth edition edn. (2012)
24. Sonnenburg, S., Franc, V., Yom-Tov, E., Sebag, M.: Pascal large scale learning challenge. In: 25th International Conference on Machine Learning (ICML2008) Workshop. vol. 10, pp. 1937–1953 (2008)
25. Wang, X., Wu, W., Wu, J., Chen, Y., Zrymiak, N., Qu, C., Flokas, L., Chow, G., Wang, J., Wang, T., et al.: Connectorx: accelerating data loading from databases to dataframes. Proceedings of the VLDB Endowment **15**(11), 2994–3003 (2022)
26. Yan, C., He, Y.: Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. p. 1539–1554. SIGMOD '20 (2020)
27. Zheng, A., Casari, A.: Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists. O'Reilly Media, Inc., 1st edn. (2018)