APPENDIX

A FACTORIZED LINEAR ALGEBRA OPERATIONS

Transpose. The rewrite for transpose is different from other rewrites. Instead of computing the transpose of the data matrix, we add a binary flag indicating that the matrix has been transposed. We do not make any other changes to our matrix. When a matrix is transposed, other LA operations must be adapted to maintain correctness. We describe the standard and transposed rewrite rules for each of the following operations.

Scalar function f. The scalar function f can be, for instance, a log function, sin function, or exp. Scalar functions also return matrices. The rewrite of the scalar function is the following.

$$f(T) \rightarrow [f(S), I, M]$$

Since this rewrite returns matrices, which can be transposed using a flag, the result in the transposed case is written as follows.

$$f(T^T) \rightarrow f(T)^T$$

Row summation. The row summation operation creates a vector of size $r_T \times 1$. The rewrite of row summation is the following.

$$\text{rowSums}(T) \rightarrow I_1\text{rowSums}(S_1) + ... + I_k\text{rowSums}(S_k)$$

Performing row summation on a transposed matrix is the same as performing column summation over a non-transposed matrix. We can define the rewrite in the transposed case as follows.

$$\text{rowSums}(T^T) \rightarrow \text{colSums}(T)$$

Column summation. The column summation operation creates a vector of size $1 \times c_T$. The rewrite of column summation is the following.

$$\text{colSums}(T) \rightarrow \text{colSums}(I_1)S_1M_1^T + ... + \text{colSums}(I_k)S_kM_k^T$$

Performing column summation on a transposed matrix is the same as performing row summation over a non-transposed matrix. We can define the rewrite in the transposed case as follows.

$$\text{colSums}(T^T) \rightarrow \text{rowSums}(T)$$

Right matrix Multiplication. Right multiplication is also referred to as right matrix multiplication (RMM). The result of RMM between T and another matrix X is a matrix of size $r_X \times c_X$. Our rewrite of RMM is as follows.

$$XT \rightarrow XI_1S_1M_1^T + ... + XI_KS_KM_K^T$$

Performing RMM between two matrices where the second matrix is transposed is the same as multiplying the untransposed second matrix with the transposed first matrix and transposing this result. We can define the rewrite in the transposed case as follows.

$$XT^T \rightarrow (TX^T)^T$$

To remove source redundancy during computations, we set values in the source tables to zero. If there is an overlapping value between S_i and S_j, then the value will be set to zero in S_j. As we implement the source tables using sparse matrices in which zero values are not stored, this value is removed from memory. Source redundancy should, in theory, not negatively impact the speedups we can achieve through factorization. In practice, we expect that source redundancy will introduce overhead, resulting in a decrease in performance for factorization.

Materialization. Although materialization is not a linear algebra operation, we still include it in this section. Materialization is important to consider for other purposes such as testing and evaluation, and plays a role in our cost estimation. We use the definition below when we want to perform learning over materialized data.

$$T \rightarrow I_0S_0M_0^T + ... + I_kS_kM_k^T$$

B LINEAR ALGEBRA LEVEL OPTIMIZATIONS

Row summation. The rewrite of row summation is only affected by the second optimization. The adapted rewrite rule is the following:

$$\text{rowSums}(T) \rightarrow \text{rowSums}(S_1) + I_2\text{rowSums}(S_2) + ... + I_k\text{rowSums}(S_k)$$

Column summation. The rewrite of column summation is affected by both optimizations. Therefore, we describe three cases: one case without $\mathbf{M}$, one case without I_1, and one case without both. First, we describe the case without $\mathbf{M}$. To maintain correctness, we horizontally stack the intermediate results for source tables. The corresponding rewrite rule is the following:

$$\text{colSums}(T) \rightarrow [\text{colSums}(I_1)S_1, ..., \text{colSums}(I_k)S_k]$$

For the case without I_1, the rewrite rule of column summation is the following:

$$\text{colSums}(T) \rightarrow \text{colSums}(S_1)M_1^T + \text{colSums}(I_1)S_1M_1^T + ... + \text{colSums}(I_k)S_kM_k^T$$

For the cases without $\mathbf{M}$ and I_1, we are again horizontally stacking instead of aggregating the intermediate results. The adapted rewrite rule is the following:

$$\text{colSums}(T) \rightarrow [\text{colSums}(S_1), \text{colSums}(I_1)S_1, ..., \text{colSums}(I_k)S_k]$$

Right matrix Multiplication. The operation RMM is also affected by both optimizations. For the case without $\mathbf{M}$, the adapted rewrite of RMM involves horizontal stacking and is described below.

$$XT \rightarrow [XI_1S_1, ..., XI_KS_K]$$

For the case without I_0, the adapted rewrite of RMM is the following.

$$XT \rightarrow XS_1M_1^T + ... + XS_KM_K^T$$

For the case without $\mathbf{M}$ and without I_1, the rewrite of RMM again involves horizontal stacking and is described as follows.

$$XT \rightarrow [XS_1, ..., XI_KS_K]$$

C MACHINE LEARNING MODELS

To make a fair comparison with state-of-the-art, we implemented all the below ML algorithms the same as [16]. All pseudocodes below are from [16]. We include them here as the preliminary of conducting complexity analysis of ML models in Sec. D.

Logistic regression. Logistic regression is another ML model based on supervised learning. It aims to predict the probability of a binary dependent variable based on one or more independent variables using a logistic function. The logistic regression algorithm used in our system is described in algorithm 3. This algorithm also uses gradient descent.

Algorithm 2 Linear regression using Gradient Descent ([16])

Input: X, y, w, γ
  for $i \in 1 : n$ **do**
    $w = w - \gamma(T^T((Tw) - Y))$
  end for

Algorithm 3 Logistic regression using Gradient Descent

Input: X, y, w, γ
  for $i \in 1 : n$ **do**
    $w = w - \gamma(T^T \frac{Y}{1+e^{Tw}})$
  end for

Gaussian Non-Negative Matrix Factorization. Gaussian Non-Negative Matrix Factorization (Gaussian NMF) is a supervised ML algorithm that represents its input matrix as two smaller matrices, of which the product approximates the input. The size of the smaller matrices is determined by the hyperparameter rank r. It is used for purposes such as clustering, dimensionality reduction, and feature extraction. The Gaussian NMF algorithm used in our system is described in algorithm 4 and is based on the multiplicative update rule by [54].

Algorithm 4 Gaussian NMF

Input: X, W, H
  for $i \in 1 : n$ **do**
    $H = H \times (\frac{W^T T}{W^T W H})$
    $W = W \times (\frac{TH^T}{W(HH^T)})$
  end for

K-Means. K-Means is an unsupervised ML algorithm used for clustering. It groups data in a user-specified number of clusters by defining a centroid for each cluster. The number of clusters is specified through hyperparameter k. Then, data points are assigned to the closest cluster centroid. The K-Means algorithm used in our system is described in algorithm 5.

Algorithm 5 K-Means

Input: X, k
  1. Initialize centroids matrix $C_{r_X \times k}$
  $\mathbf{1}_{a \times b}$ represents a matrix filled with ones with dimension $a \times b$, it is used for replicating a vector row-wise or column-wise.
  2. Pre-compute l^2-norm of points for distances.
  $D_T = \text{rowSums}(T^2)\mathbf{1}_{1 \times k}$
  $T_2 = 2 \times T$
  for $i \in 1 : n$ **do**
    3. Compute pairwise squared distances; $D_{r_X \times k}$ has points on rows and centroids/clusters o columns.
    $D = D_T - T_2 C + \mathbf{1}_{r_X \times 1}\text{colSums}(C^2)$
    4. Assign each point to the nearest centroids; $A_{r_X \times k}$ is a boolean assignment matrix.
    $A = (D == \text{rowMin}(D))\mathbf{1}_{1 \times k}$
    5. Compute new centroids; the denominator counts the number of points in the new clusters, while the numerator adds up assigned points per cluster.
    $C = (T^T A)/(\mathbf{1}_{d \times 1}\text{colSums}(A))$
  end for

D COMPLEXITY OF MACHINE LEARNING MODELS

Gaussian NMF. We define the complexity of Gaussian NMF on data matrix T below. From the parameter r supplied to Gaussian NMF we can infer the shapes of matrices W and H. The shape of matrix W is $r_T \times r$, the shape of matrix H is $c_T \times r$. We assume both matrices are dense, so the number of nonzero elements $nnz(W) = r_W \times c_W$ and $nnz(H) = r_H \times c_H$. The complexity in the standard case is shown below.

$$O_{\text{standard}}(T) = \underbrace{c_T \cdot nnz(W) + c_W \cdot nnz(T)}_{W^T T}$$
$$+ \underbrace{r_H \cdot nnz(T) + r_T \cdot nnz(H)}_{TH^T}$$

The complexity in the factorized case is shown below.

$$O_{\text{factorized}}(T) = \underbrace{\sum_{k=1}^{K} c_{S_k} \cdot nnz(W) + c_W \cdot nnz(S_k) +}_{W^T T}$$
$$\underbrace{\sum_{k=1}^{K} r_H \cdot nnz(S_k) + r_{S_k} \cdot nnz(H)}_{TH^T}$$

K-Means. We define the complexity of K-Means on data T below. From the parameter k supplied to K-Means we can infer the shapes of C and A. The shape of C is $c_T \times k$, the shape of A is $r_T \times k$. We assume both C and A are dense, so $nnz(C) = r_C \times c_C$ and $nnz(A) = r_A \times c_A$. The complexity in the standard case is shown below.

$$O_{\text{standard}}(T) = \underbrace{2nnz(T)}_{\text{rowSums}(T^2)} + \underbrace{nnz(T) + c_C \cdot nnz(T) + r_T \cdot nnz(C)}_{(2 \times T)C}$$
$$+ \underbrace{c_T \cdot nnz(A) + c_A \cdot nnz(T)}_{T^T A}$$

The complexity in the factorized case is shown below.

$$O_{\text{factorized}}(T) = 2 \underbrace{\sum_{k=1}^{K} nnz(S_k)}_{\text{rowSums}(T^2)}$$
$$+ \underbrace{\sum_{k=1}^{K} nnz(S_k) + c_C \cdot nnz(S_k) + r_{S_k} \cdot nnz(C)}_{(2 \times T)C}$$
$$+ \underbrace{\sum_{k=1}^{K} c_{S_k} \cdot nnz(A) + c_A \cdot nnz(S_k)}_{T^T A}$$

E IMPLEMENTATION OF LMM WHEN NO COLUMN OVERLAP

In the scenarios where there are no column overlap, we can simplify the rewriting rules, which we introduce here. We continue to use the example of Left Matrix Multiplication. LMM is affected by both optimizations. To create the adapted rewrites, we need a new notation. We define c'_k as $\sum_{k=0}^{k-1} c_{S_k}$, which defines the number of combined columns in all source tables before source table k. For the case without $\mathbf{M}$, LMM does not require horizontally stacking the result. Instead, we are indexing matrix X using c'_k. The adapted rewrite of LMM is the following.

$$TX \rightarrow I_1 S_1 X[0 : c_{S_1}] + ... + I_K S_K X[c'_k : c'_k + c_{S_k}]$$

For the case without I_0, the rewrite of LMM is the following.

$$TX \rightarrow S_1 M_1^T X + I_2 S_2 M_2^T X + ... + I_K S_K M_K^T X$$

For the case without $\mathbf{M}$ and I_0, we are again indexing matrix X using c'_k. The rewrite of LMM is the following:

$$TX \rightarrow S_1 X[0 : c_{S_1}] + I_2 S_2 X[c_{S_1} : c_{S_1} + c_{S_2}] + ... + I_K S_K X[c'_k : c'_k + c_{S_k}]$$

F POSSIBLE EXTENSIONS TO EGDS

For eliminating redundancies during normalization, an important class of dependencies is functional dependencies (FDs), which can be expressed as equality generating dependencies (egds). An egd is a first-order formula of the form $\forall \mathbf{x} \, (\varphi(\mathbf{x}) \rightarrow (x_1 = x_2))$, where $\varphi(\mathbf{x})$ is a conjunction of atomic formulas, all with variables among the variables in $\mathbf{x}$; every variable in $\mathbf{x}$ appears in $\varphi(\mathbf{x})$ [31]. Now we generalize our assumption and improve the applicability, i.e., consider the existence of egds in source and/or target schemas. We extend our formal framework in Sec. 2.2 and 4.1 to $\mathcal{M} = \langle \mathbf{S}, T, \Sigma \rangle$, where we change Σ to a set of three types of dependencies: s-t

tgds, source egds and target edgs. Source and target egds could be functional dependencies over source and target tables, respectively. Compared to tgds, edgs have different formal properties [14, 8]. Extending our logical pruning rules to egds is non-trivial and deserving of dedicated study. Here we merely demonstrate such a possibility with the following example. Since the context is clear, we omit universally quantified variables, e.g., $\forall x_1$. We added more explanation about egds in [27].

EXAMPLE F.1. *Consider source tables S_1 and S_2. f_1 and f_2 are FDs over source tables S_1 and S_2. $f_3 - f_5$ are FDs over target table T.*
S-t tgd:
$m_1 : S_1(x_1, x_2, x_3) \wedge S_2(x_2, x_4) \rightarrow T(x_1, x_2, x_3, x_4)$
Source egds over source tables S_1 and S_2:
$f_1 : S_1.x_1 \, S_1.x_2 \rightarrow S_1.x_3 \implies S_1(a, b, c_1) \wedge S_1(a, b, c_2) \rightarrow c_1 = c_2$
$f_2 : S_2.x_2 \rightarrow S_2.x_4 \implies S_2(b, d_1) \wedge S_2(b, d_2) \rightarrow d_1 = d_2$
Target egds over target table T:
$f_3 : x_1 x_2 \rightarrow x_3 \implies T(a, b, c_1, d_3) \wedge T(a, b, c_2, d_4) \rightarrow c_1 = c_2$
$f_4 : x_1 x_2 \rightarrow x_4 \implies T(a, b, c_3, d_1) \wedge T(a, b, c_4, d_2) \rightarrow d_1 = d_2$
$f_5 : x_2 \rightarrow x_4 \implies T(a_1, b, c_1, d_1) \wedge T(a_2, b, c_2, d_2) \rightarrow d_1 = d_2$

To explain Example F.1, we take FD f_1 over source table S_1 for example. The two columns $S_1.x_1$ and $S_1.x_2$ determines $S_1.x_3$. Thus, in the corresponding edg, the left-hand-side of the implication $S_1(a, b, c_1) \wedge S_1(a, b, c_2)$ indicates the condition, i.e., when we have two instances of S_1 sharing the same values of x_1 and x_2 (a and b); the right-hand-side of the implication indicates that they also share the same values of column $S_1.x_3$, i.e., $c_1 = c_2$. Notably, for FD f_3 over target table T, since x_1 and x_2 determine x_3, we have $c_1 = c_2$. However, we do not know whether $d_3 = d_4$, as this is not implied by the function dependency f_3.

At the logical level, it is easy to tell: most likely there is more redundancy in the target table than in source tables[6], and the optimizer will choose factorization, i.e., area I in Fig. 6.

G OPTIMIZATIONS OF MATRIX MULTIPLICATION ORDER

To reduce computation overhead, we reorder the matrix multiplication sequence, similar to the optimization of join ordering in databases. The problem of ordering matrix multiplications to minimize cost is also known as the *matrix chain multiplication problem* or the *matrix chain ordering problem*. The optimal ordering can be computed from the dimensions of the matrices. We calculate an optimal ordering of intermediate computations to optimize our implementation of the LA rewrite rules.

As an example, we inspect how to compute an optimal ordering for the multiplication $I_k \cdot S_k \cdot M_k^T$ for a source table k. Let the size of S_k be $r_k \times c_k$ and the size of T be $r_T \times c_T$. The size of I_k is $r_T \times r_k$ and the size of M_k is $c_T \times c_k$. The size of X is $c_T \times c_X$. We describe our matrix chain ordering problem as follows:

$$\underbrace{I_k}_{r_T \times r_k} \cdot \underbrace{S_k}_{r_k \times c_k} \cdot \underbrace{M_k^T}_{c_k \times c_T}$$

---

[6]It is similar to the second normal form. For instance, there might be redundant values of x_2, x_4 in target table T.
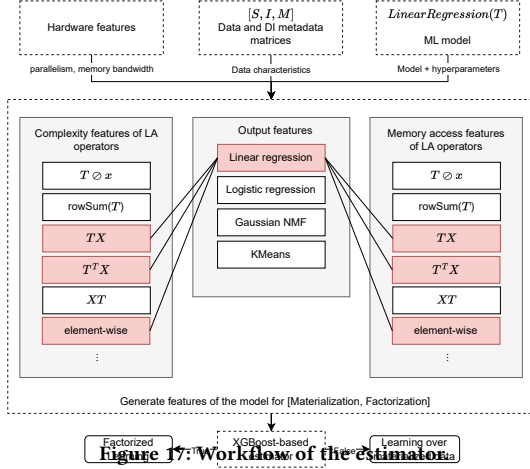
Figure 17: Workflow of the estimator.

We denote the number of matrices as n, and the number of possible parenthesizations as $P(n)$ that can be computed as following [22].

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

There are two possible multiplication orderings for the three matrices, which can also be described as parenthesization options. These are $(I_k \cdot S_k) \cdot M_k^T$ and $I_k \cdot (S_k \cdot M_k^T)$. The cost of the first ordering is $r_T \cdot r_k \cdot c_k + r_T \cdot c_k \cdot c_T$. The cost of the second ordering is $r_k \cdot c_k \cdot c_T + r_T \cdot r_k \cdot c_T$. We calculate the optimal ordering where $r_T = 10$, $c_T = 6$, $r_1 = 10$, $c_1 = 1$, $r_2 = 1$ and $c_2 = 5$. For table S_1, the first option has a lower cost, as $10 \cdot 1 \cdot (10 + 6) < (10 \cdot 6 \cdot (10 + 1))$. For table S_2, the second option has a lower cost, as $10 \cdot 5 \cdot (1 + 6) > 1 \cdot 6 \cdot (10 + 5)$. We adopted our implementation of the optimal matrix multiplication order algorithm from the version included in NumPy[7] for SciPy matrices, computed using matrix dimensions.

Details of the Estimator's Pipeline Figure 17 illustrates the workflow of our proposed estimator with the example of linear regression as input ML model.

Step 1: Process input. To apply our approach in general integration scenarios for ML use cases, we do not make specific assumptions about the properties of the input data. Thus, from the data matrices, we infer parameters necessary for complexity analysis, e.g., m_T, m_k, r_T, r_k in Table 3. These characteristics include the dimensions and sparsities of the source tables and the dimensions and sparsity of the materialized table.

Step 2: Feature engineering. We calculate linear algebra (LA) operation complexities as detailed in Sec. 4.2, incorporating these complexities into the ML model as outlined in Sec.4.3.[8] The estimator also computes theoretical memory read and write quantities for each operator within the input ML algorithm to be trained. Since factorization and materialization have different complexities and memory I/O amounts, we normalize these features by dividing by total amounts, further dividing these normalized features by parallelism and memory bandwidth. In addition, we include the

complexity ratio and tuple ratio into our estimator, comparing feature importance during evaluation (Sec. 5). Due to the limited space, we will showcase the features in our technical report [27].

Step 3: Train XGBoost for the estimator. Training our estimator commences with creating a comprehensive training dataset that spans a wide array of data characteristics and input models. The data, as outlined in Sec. 5.1, is randomly generated within defined ranges. It's subsequently used for training four distinct ML models supported by our system, under both factorization and materialization. We extract features from the raw input data, and assign binary labels based on the speedup of factorization over materialization. Our objective is to optimize the estimator's performance across all training configurations.

Step 4: Performance estimation and strategy determination. Once the estimator is trained, it is used for binary classification, i.e., to determine whether to materialize or factorize. Specifically, if the inference result is True, Ilargi trains for the input ML algorithm, such as linear regression, using the factorization approach as outlined in Section 3.2. Alternatively, if the output is False, Ilargi first join the source tables, subsequently training the linear regression model on the materialized target table.

H FEATURE LIST

f0,f1 materialized memory read, write
f2,f3 factorized memory read, write
f4-f6 scalar, LMM, RMM complexity with materialization
f7-f9 scalar, LMM, RMM complexity with factorization
f10 col-major ops complexity with materialization
f11 col-major ops complexity with factorization
f12 dense scalar op complexity/total materialization complxity
f13 dense scalar op complexity/total factorization complxity
f14 dense MM op complexity/total materialization complxity
f15 dense MM op complexity/total factorization complxity
f16-f19 dense scalar read and write, dense MM read and write / total materialization memory
f20-f23 dense scalar read and write, dense MM read and write / total factorization memory
f24-f27 colSum read and write, rowSum read and write / total materialization memory
f28-f31 colSum read and write, rowSum read and write / total factorization memory
f23,f33 colSum and rowSum complexity / total materialization complexity
f34,f35 colSum and rowSum complexity / total factorization complexity
f36 selectivity
f38 complexity ratio
f39,f40 Tuple ratio, feature ratio

---

[7]https://numpy.org/doc/stable/reference/generated/numpy.linalg.multi_dot.html
[8]Hyperparameters such as the rank r for GNMF and the number of clusters k for KMeans are factored into the cost estimation. However, other hyperparameters like the learning rate γ for linear and logistic regression do not affect the cost estimation.