Efficient ML Model Training over Silos: to Factorize or to Materialize

Wenbo Sun* Jessie van Schijndel* Asterios Katsifodimos Rihan Hai
Delft University of Technology
{w.sun-2,j.vanschijndel,a.katsifodimos,r.hai}@tudelft.nl

ABSTRACT

This study aims to address the challenge of training machine learning (ML) models using data from disparate sources. The existing approaches for this problem include materialization and factorization. Materialization involves integrating all source tables before loading the joined result into an ML tool, while factorization enables pushing down linear algebra operators to underlying source tables to enhance efficiency. However, existing solutions fall short in two areas when dealing with data sources outside a single database. Firstly, they lack adequate support for the complex relationships between source tables and the desired target table, which are typically defined by schema mappings and row matching. Secondly, there is no systematic method for distinguishing between cases when materialization or factorization is more efficient.

To address these challenges, we propose *Ilargi*, a system that executes the algorithmic operations of a linear ML model across separated source tables. We cover a broad range of typical dataset relationships in ML use cases, representing them in matrix forms, which leads to a unified execution of data transformation operations and linear algebra operations. With the aid of first-order logic-based pruning rules and logical-level cost estimation, Ilargi can automatically choose between materialization and factorization. Through intensive experiments, we demonstrate that Ilargi effectively improves the time-wise efficiency of model training.

1 INTRODUCTION

When the training data of a machine learning model comes from different sources, is it faster to train the model over the source tables or the materialized join result?

Data is at the core of machine learning. The goal of ML methods is to automatically extract information from training data for accurate predictions over unseen data [12, 44]. In real world applications, often data is not stored in a central database or file system, but spread over different data silos. Take, for instance, drug risk prediction: the features can reside in datasets collected from clinics, hospitals, pharmacies, and laboratories distributed geographically [5]. Another example is training models for keyboard stroke prediction: training requires data from millions of phones [26].

Data integration as a preprocessing step of ML pipeline. When training data is spread over different sources, the common practice is to integrate them into one training dataset, as a preprocessing step of ML pipeline. Fig. 1 illustrates an example of predicting the mortality (binary classification) of patients based on tables maintained by different departments in the same hospital, S_1 and S_2. Data integration systems enable interoperability among multiple,

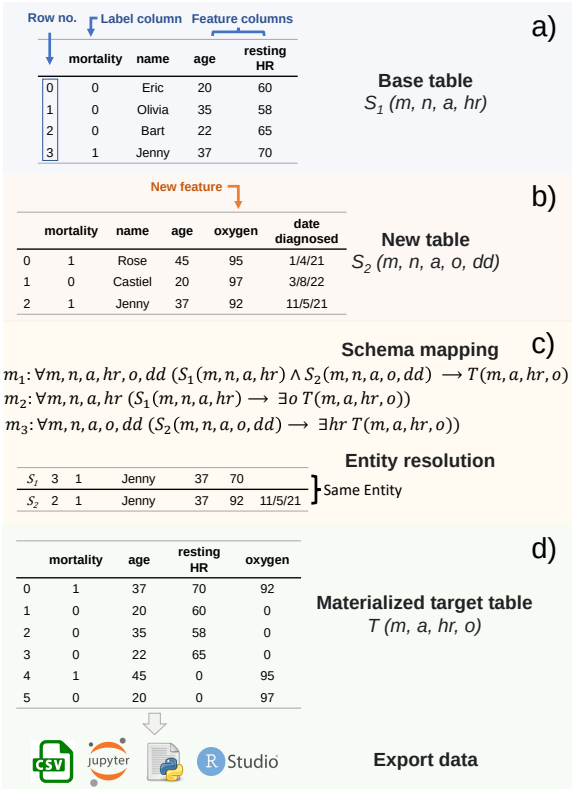

Figure 1: Traditional DI workflow before ML tasks

heterogeneous sources, and provide a unified view for users. Notably, they allow us to describe data sources and their relationships [14]: *i*) mappings between different source schemata, i.e., schema matching and mapping [47, 22] and *ii*) linkages between data instances, i.e., data matching (also known as record linkage or entity resolution) [6]. Yet, a data integration system's goal is to facilitate query answering or data transformation over silos, and not to directly support machine learning applications. Consequently, we can materialize the data according to the global schema into a target table T and export T to downstream ML applications.

Factorization as a new paradigm. Pushing ML models over joins is coined as the problem of factorized learning and has been intensively studied [34, 35, 7, 2, 40, 8, 31, 15, 16, 49, 50]. In a nutshell, factorized learning can be seen as an extended application of data dependency and redundancy studies over relational models. To efficiently train linear ML model over multiple tables of a single database, *learning over factorized joins* [49] has been proposed, also known as *factorized learning* [34]. The training process of an ML

*Authors have contributed equally to this work.

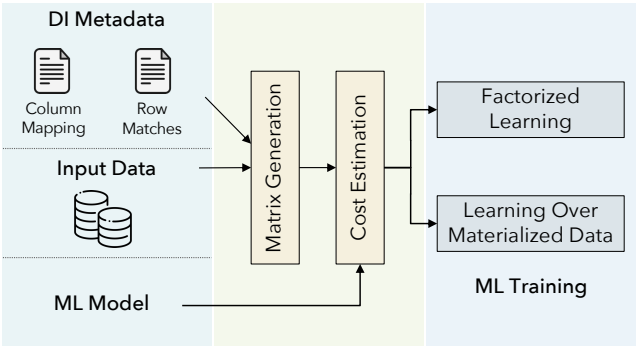

Figure 2: Our approach.

model requires complex arithmetic computations of linear algebra (LA) operators. Given an ML model and joinable tables of a database, a general approach [7] is proposed to reformulate LA operators and push down their computation to each table (elaborated in Sec. 2). Compared to materialization, factorized learning does not affect model training accuracy but helps with time-wise efficiency.

Research challenges. It is natural to ask whether factorized learning can help with ML model training over *multiple* sources instead of single database. The main challenges are two-fold:

First, existing factorized learning approaches *do not cover typical DI scenarios*. When two input tables have primary key-foreign key (PK-FK) constraint, their joined result often contains redundant data. Factorized learning [7] proposes to execute the computation of LA operators over input tables instead of their joined results. The approach also supports multi-table joins over PK-FK constraints or natural joins over shared join columns. However, as illustrated in Table 1, when datasets come from different sources, there are more possibilities for how to join them, depending on the use cases and users. We are still missing a comprehensive and formal analysis of possible relationships between source datasets for ML use cases.

Second, factorized learning *does not always yield speedups in DI scenarios*. In DI scenarios, sources may have data overlap, e.g., a patient's personal details may be recorded in the databases of two hospitals she visited. Our problem setting poses a greater challenge than prior factorization work [7]. Depending on the relationships between datasets, source tables may contain redundant data, and their joined result (the target table) may also have redundancies. Therefore, the question arises: which factor plays a more critical role in the efficiency of linear algebra execution? Specifically, given a downstream ML model and a set of data sources, is it more efficient to materialize with data integration a preprocessing step, or to factorize by computing LA operations within each data source?

Our Solution. We first analyze the possible dataset relationships in ML use cases. By formalizing these DI scenarios using a schema mapping language, i.e., tuple-generating dependencies [4, 18], we lay the foundation for future theoretical work on ML model training over data sources. As shown in Fig. 2, we propose an approach that *represents* dataset relationships as matrices, *computes* DI tasks and ML tasks, and *optimizes* the workflow based on cost estimation.

We summarize our contribution as follows.

- *Problem formalization (Sec. 2).* We analyze the redundancies in source and target tables in typical data integration scenarios for ML applications. We extend the application from single database to DI scenarios, i.e., Fig. 3. We formalize the problem of cost estimation of factorization and materialization over data sources.
- *Representation & System (Sec. 3).* We propose *Ilargi*, a system that represents schema mappings and row matching between sources in the form of matrices. Illargi enables a unified computation of data transformation and linear algebra.
- *Cost estimation (Sec. 4).* We propose a workflow optimizer, which decides whether to factorize or to materialize given the data sources, their schema mappings and row matching, and the downstream ML model. We provide the complexity analysis for factorization and materialization.
- *Experiments (Sec. 5).* Through an extensive evaluation using datasets with diverse characteristics, we show that our cost estimation method is effective in telling apart cases of factorization and materialization. We also show the cost estimation overhead is omissible. We enlighten future hardware optimization and parallelization opportunities.

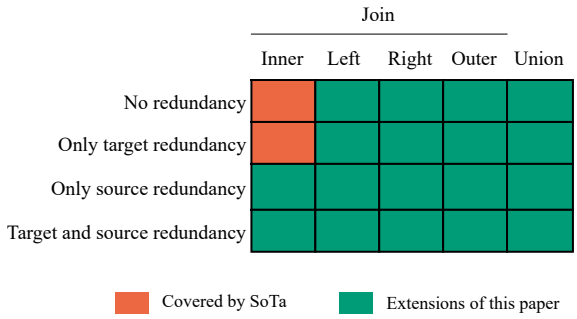

Figure 3: Ilargi vs. SoTA [7].

2 DATA INTEGRATION & FACTORIZATION

We first explain the two existing strategies of training ML models over source tables: materialization and factorization. To represent the schema-level relationships in typical DI scenarios for ML use cases, we discuss our formal framework based on first-order logic. Finally, we analyze the gaps and reveal the core research questions.

2.1 Preliminary: LA Computation Strategies

The training process of an ML model requires complex arithmetic computations. Materialization and factorization are the two possibilities for conducting LA computations of a linear ML model over separated tables from different sources.

Materialization. As depicted in Fig. 1, data instances from different source tables are combined and materialized as the target table, e.g., through joins. Later the target table is exported from data integration systems before being imported into ML tools for model training. Data integration is a preprocessing step before training.

Factorization. Given an ML model and joinable tables of a database, factorized learning [34, 7] is the process of reformulating the execution rules of LA operations of an ML model and pushing down the computation to each table.[1] Compared to materialization, factorized learning does not affect model training accuracy but often

[1] *Factorized learning* [34] or *learning over factorized joins* [49] is a broad topic, including many challenges. In this work, we mainly focus on the problem of decomposing linear ML models over joinable tables, which is close to the goal of this paper.

Table 1: Four example data integration scenarios for feature augmentation and federated learning [25].

No.	Dataset Relationship	Schema mappings	Example use cases
1	Full outer join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(m,n,a,o,dd) \rightarrow T(m,a,hr,o))$ $m_2: \forall m,n,a,hr\ (S_1(m,n,a,hr) \rightarrow \exists o\ T(m,a,hr,o))$ $m_3: \forall m,n,a,o,dd\ (S_2(m,n,a,o,dd) \rightarrow \exists hr\ T(m,a,hr,o))$	Feature augmentation, Federated learning, …
2	Inner join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(m,n,a,o,dd) \rightarrow T(m,a,hr,o))$	Feature augmentation, (Vertical) federated learning, …
3	Left join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(n,a,o,dd) \rightarrow T(m,a,hr,o))$ $m_2: \forall m,n,a,hr\ (S_1(m,n,a,hr) \rightarrow \exists o\ T(m,a,hr,o))$	Feature augmentation, (Vertical) federated learning, …
4	Union	$m_2: \forall m,n,a,hr,o\ (S_1(m,n,a,hr,o) \rightarrow T(m,a,hr,o))$ $m_3: \forall m,n,a,hr,o,dd\ (S_2(m,n,a,hr,o,dd) \rightarrow T(m,a,hr,o))$	Data sample augmentation, (Horizontal) federated learning, …

helps to improve the training efficiency [34, 35, 7, 2, 40, 8, 31, 15, 16, 49, 50]. Similar to [7], in this work, we focus on linear regression, logistic regression, K-Means, and Gaussian Non-Negative Matrix Factorization. Gradient descent is used in both linear regression and logistic regression algorithms. For simplicity, we use linear regression to explain factorized learning.

Rewriting rules for arithmetic operations. Given two tables S and R connected by PK-FK relationship, they can be joined to obtain the target table T, i.e., $T \leftarrow S \bowtie R$. Given a LA operator over target table T, and joinable tables S and R, the rewriting rules of factorizing the LA operator and creating new LA operations over S and R are proposed in the framework *Morpheus* [7]. After transforming S and R into their matrix form, their row-matching relationship can be defined as a matrix. This is referred to as *indicator matrix*, which we elaborate its definition in Sec. 3.1. Morpheus covers the common LA operators used in linear ML models. The LA operators that benefit from the speed-up brought by factorization, can be divided into four groups, element-wise scalar operators (binary arithmetic operations $\oslash$ such as $+, -, *, /$ and $\hat{}$, transpose T^T, etc.), aggregation (sum up matrix elements by row/column/all) and matrix multiplication, and matrix inversion. Consider the linear regression. It contains two LA operators that may benefit from factorization, transpose T^T, and left matrix multiplication Tw.

Algorithm 1 Linear regression using Gradient Descent ([7])

Input: X, y, w, γ
0: **for** $i \in 1 : n$ **do**
0: $\quad w = w - \gamma(T^T((Tw) - Y))$
0: **end for**=0

We showcase the rewriting rules of factorizing two operators from Morpheus (c_S, c_R and c_T are the column numbers of tables S, R, T): transpose T^T (x is a scalar, and the transpose is implemented using a flag), and left matrix multiplication TX (X is a model or weight vector). I_1 and I_2 are the indicator matrices of S and R.

$$T^T \oslash x \rightarrow (T \oslash x)^T$$
$$TX \rightarrow I_1(SX[1 : c_S,]) + I_2(RX[c_R + 1 : c_T,])$$

Research Gap. The main focus of existing factorized learning works [7] is to perform factorized computation of LA operators over joinable tables. The joinable tables in a single database are connected with the primary key-foreign key constraint (a special case of inclusion dependencies) or join dependencies. In this work, we are interested in joinable tables from different databases of data sources, where PK-FK crossing sources is *not* a realistic assumption of the data integration setting. Next, we introduce the tuple-generating dependencies as the formalism to unify the dependencies in a single database [4], and crossing multiple databases of data sources [23].

2.2 Unifying Formalism for DI & Factorization

Schema mappings lay at the heart of data integration. Schema mappings are semantic descriptions of the contents of data sources and the relationships between data source schemas and the target schema. In a traditional DI system, with schema mappings, we can use the data instances in source tables to generate new instances for the target table, i.e., materialization. Alternatively, given a user query, we can utilize schema mappings to rewrite the input query to subqueries over source tables, i.e., virtual data integration. One of the most commonly used mapping languages is *source-to-target tuple generating dependencies (s-t tgd)* [4, 18], which are also known as Global-Local-as-View (GLAV) assertions [39].

Tgd definition. Let $\mathbf{S}$ and $\mathbf{T}$ be a source relational schema and a target relational schema sharing no relation symbols. A *schema mapping* $\mathcal{M}$ between $\mathbf{S}$ and T is a triple $\mathcal{M} = \langle \mathbf{S}, T, \Sigma \rangle$, where Σ is a set of dependencies over $\langle \mathbf{S}, T \rangle$. Σ can be expressed as logical formulas over source and target schemas. An s-t tgd is a first-order sentence in the form of $\forall \mathbf{x}\ (\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\ \psi(\mathbf{x}, \mathbf{y}))$, where $\varphi(\mathbf{x})$ is a conjunction of atomic formulas over the source schema $\mathbf{S}$, and $\psi(\mathbf{x}, \mathbf{y})$ is a conjunction of atoms over the target schema $\mathbf{T}$.

Example 2.1. *Consider the example of integrating S_1 and S_2 in Fig 1 via a full outer join. The dataset relationship can be expressed by three tgds, i.e., m_1, m_2, and m_3. We represent mapped attributes with the same variable names, e.g., $S_1.m$ and $S_2.m$. The tgd m_1 specifies that the overlapped rows of S_1 and S_2 are added to T ($\wedge$ denotes a natural join between S_1 and S_2); m_2 and m_3 indicate that all rows of S_1 and S_2 will be transformed to generate new tuples in T, respectively. Among the three tgds, it is the union relationship. The three tgds together describe that the instances in T are obtained via a full outer join between the datasets S_1 and S_2.*

Tgds as unifying formal framework. S-t tgds are special cases of tgds. That is, the left-hand-side of an s-t tgds are atoms over source schemas while the right-hand-side are target schema atoms. Tgds, in general, can be used to describe inclusion dependencies and join dependencies [21]. In this work, we employ tgds to describe the dataset relationships in different scenarios of ML use cases.

2.3 TGDs for ML in Practice

In an earlier work [25], we studied two representative ML use cases where training data could come from silos, i.e., feature augmentation and federated learning. *Feature augmentation* is the exploratory process of finding new datasets and selecting features that help improve the ML model performance [9, 17, 36]. Figure 4b shows an example: starting from a base table S_1, we augment the features by introducing the table S_2 and selecting the new feature o (*oxygen*). *Federated learning* [54] studies how to build joint ML models over data silos (e.g., enterprise data warehouses, edge devices) without compromising privacy, which follows a decentralized learning paradigm. Similar to the problem setting of virtual data integration [14], FL assumes that the source data is not collected and stored at a central data store but stays at the local data stores. For vertical federated learning, data sources share the overlapping data instances, but the feature columns partially overlap or not. For horizontal federated learning, data sources share the overlapping feature columns, while the data instances may overlap.

As shown in Table 1, for the use cases of feature augmentation and federated learning (and possibly many more), we formalize the dataset relationships between source tables and desired target table with s-t tgds. Full outer join is explained in Example 2.1. It can also represent a general case of federated learning, where sources have similar schemas, and data instances (entities) that may or may not overlap with each other. Without repeating too many details of [25], we explain the union example in Table 1. The union example can be seen as a special case of full outer join, where S_1 and S_2 do not share any rows. We modify the schemas of S_1 and S_2 such that they share the same set of feature columns which are mapped to the target schema T. The union example can represent the scenario when a new table is selected to bring more data samples. Alternatively, it can describe the HFL scenario where data sources share feature columns but not data samples.

2.4 Open Problems & Challenges

Representation and execution. At the logical level, we can use tgds to express both inclusion dependencies in a database (e.g., PK-FK constraint) and schema mappings in data integration scenarios in Table 1. However, in a data integration scenario, the data sources are autonomous and their schemata, e.g., inclusion dependencies, are designed independently. Thus, it is non-trivial to extend existing factorization work [7], due to the following reasons.

i) *Column-overlap*. In existing works, the tables are joined with a single join column, in PK-FK table joins or multi-table joins. As shown in Table 1, in data integration scenarios, it is possible for two source tables to share more than one common attribute as the result of schema matching. In this work, we make such a general assumption and are interested in conducting a systematic investigation of diverse possibilities of join columns.

ii) *Row-overlap*. Inclusion dependencies, e.g., PK-FK, indicate a containment relationship between two columns coming from two tables, while two joinable columns from two sources (expressed as the same variables in the left-hand-side of an s-t tgd) may or may not have instance overlaps. Although multi-table join is discussed, a fine-grained classification, e.g., Table 1, leads to more optimization opportunities, which will be discussed in Sec. 4.1.

iii) *Joins*. As shown in Table 1, depending on ML application and use cases, the source tables can be combined in different manners. In existing works, including state-of-the-art solution [7], only inner-joins are discussed. On the logical level, the inner-join of multiple source tables over the same join column, can be expressed as single tgd, while the rest three cases may require the union of multiple tgds as in Table 1. To summarize, the dataset relationships in data integration scenarios are more complicated than PK-FK or multi-table joins. Moreover, column matching (specified by schema mappings) and row matching are not natively expressed in linear algebras.

Research question Q1

> *How to utilize DI metadata, e.g., column and row matchings, for computing linear algebra operations?*

Optimization. As outlined in Sec. 2.1, we can compute LA operations in two manners: materialization or factorization. To choose the more efficient strategy, the key is to understand *redundancy* and *sparsity*.

Redundancy. Normalization is one of the most fundamental database concepts. When designing a relational table schema, to store less redundant data, a good database designer normalizes the table through lossless join decomposition, leveraging inclusion dependencies and functional dependencies. Factorized learning works the same way: when joinable tables have PK-FK relationship, the joined result (i.e., the target table) may have more data redundancy. We refer to such redundancy as *target redundancy*. This is the reason why executing the same LA operator, the overhead of executing it on the target table is larger than executing it on the source tables. However, the redundancy analysis in data integration scenarios is more complicated than in the cases of a single database. First, depending on the schema mapping and entity resolution results, source tables may have overlapping columns and rows. We refer to such redundancy as *source redundancy*. Fig. 4 shows an example: the patient *Eric Woodsen*'s record of weight exists in two hospital databases. In Sec. 4.1, we explain source and target redundancies in a more formal way.

Sparsity. By *sparsity*, we refer to elements with values of zero in the matrices that participate in linear algebra computation. A matrix element is zero, either the data value is zero, or transformed to zero during data preprocessing, e.g., one-hot encoding. A common practice is to replace null values with zeros. For data integration scenarios, an instance in the target table may have no corresponding value from the source table, i.e., *labeled nulls* [20]. In Sec. 4, we will reveal that the sparsity plays an important role in cost estimation when choosing between materialization and factorization.

Research question Q2

> *Given data sources and ML tasks, how to choose between materialization and factorization?*

3 ILARGI: ML OVER DIVERSE SOURCES

To tackle research question $Q1$, we introduce the workflow of our proposed system, and rewrite rules for factorizing ML models. For research question $Q2$, we elaborate on cost estimation in Sec. 4.

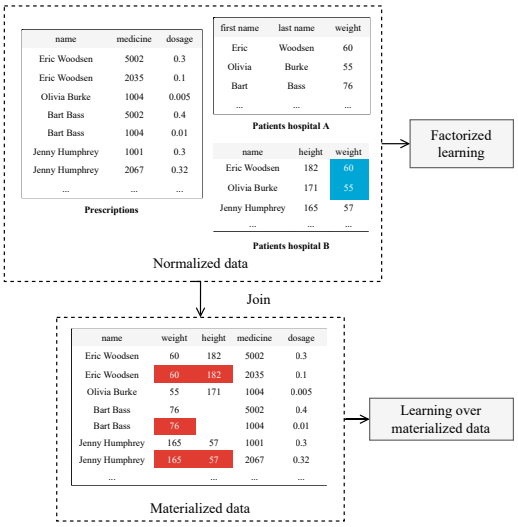

Figure 4: Running example: source redundancy (blue) vs. target redundancy (red).

Overview. Fig. 2 illustrates the workflow of our proposed system Ilargi. Ilargi takes three inputs: input source datasets, a user-defined ML model (e.g., Python scripts), and metadata about the source datasets and their data integration metadata, e.g., row matches and column matches. The output of Ilargi is the trained ML model. Ilargi has three main functions, as follows. Given the input source datasets and their metadata, Ilargi first generates their matrix-based representation (Sec.3.1). Such matrix-based representations enable a unified computation of data transformation and linear algebra operations (Sec.3.2). Second, with the metadata, the optimizer decides to factorize or to materialize based on the cost estimation (Sec. 4). Finally, the ML model is trained in the chosen strategy.

Running example. Fig. 4 illustrates a running example extended from Fig. 1. A pharmacist tries to build a linear regression model to predict medicine dosage. Data is from three source tables S_1, S_2, and S_3 with patients' weight, height, and medicine as features.

3.1 Data and DI Metadata as Matrices

Prior to our approach, source tables are preprocessed and transformed into matrices. Fig. 5 shows the matrix form of three source tables of the running example.

Mapping matrices. We represent schema mappings with a set of mapping matrices, denoted as $\mathbf{M}$. Given a source table S_k and target table T, a mapping matrix $M_k \in \mathbf{M}$ of size $c_T \times c_k$, describes how the columns of S_k mapped to the columns of T.

Definition 1 (Mapping matrix). Mapping matrices between source tables $S_1, S_2, ..., S_n$ and target table T are a set of binary matrices $\mathbf{M} = \{M_1, ..., M_n\}$. M_k $(k \in [1, n])$ is a matrix with the shape $c_T \times c_k$, where

$$M_k[i,j] = \begin{cases} 1, & if\ j^{th}\ column\ of\ S_k\ is\ mapped\ to\ the\ i^{th}\ column\ of\ T \\ 0, & otherwise \end{cases}$$

Intuitively, in $M_k[i, j]$ the vertical coordinate i represents the target table column while the horizontal coordinate j represents the mapped source table column. A value of 1 in M_k specifies the existence of column correspondences between S_k and T, while the value 0 shows that the current target table attribute has no corresponding column in S_k. Fig. 5 shows the mapping matrices M_1, M_2, M_3 for source tables S_1, S_2, S_3, respectively.

Indicator matrices. We use the *indicator matrix* [7] to preserve the row matching between each source table S_k and the target table T. An indicator matrix I_k of size $c_T \times c_k$ describes how the rows of source table S_k map to the rows of target table T, as in Fig. 5.

Definition 2 (Indicator matrix [7]). Indicator matrices between source tables $S_1, S_2, ..., S_n$ and target table T are a set of row vectors $\mathbf{I} = \{I_1, ..., I_n\}$. I_k $(k \in [1, n])$ is a row vector of size r_T, where

$$\mathrm{I}_k[i,j] = \begin{cases} 1 & \text{if the } j\text{-th row of } S_k \text{ is mapped to the } i\text{-th row of } T \\ 0 & \text{otherwise} \end{cases}$$

Implementation. It is easy to see that the binary mapping and indicator matrices are often sparse. After trying out straightforward implementation following above definitions, and alternative data structures in Python, our current implementation of mapping and indicator matrices is in sparse matrix format, SciPy CSR[2], which copes with the matrix sparsity problem, and improves performance in later steps of factorization/materialization.

3.2 Rewriting Rules for Factorization

With data and metadata represented in matrices, next, we explain how to rewrite a linear algebra over target schema to linear algebras over source schemas. The rewriting rules, although for arithmetic operations, share similar principles of view-based query rewriting [13, 10]. Here we use the example of LA operator *left matrix multiplication (LMM)*. The full set of LA rewrite rules based on mapping and indicator matrices is elaborated in the technique report [53].

Left Matrix Multiplication. Given a matrix X with the size $c_T \times c_X$, the LMM of T and X is denoted as TX. The result of LMM between our target table matrix and another matrix X is a matrix of size $r_T \times c_X$. Our rewrite of LMM goes as follows.

$$TX \rightarrow I_1 S_1 M_1^T X + ... + I_K S_K M_K^T X$$

We first compute the local result generation $I_k S_k M_k^T$ for each source table, which are assembled for the final results.

Why mapping matrices? To answer research question $Q1$ in Sec. 2.4, we aim to support general DI scenarios, where tables can be joined more than just inner joins, and the matched columns do not necessarily follow set containment relationships. This is why we proposed the mapping matrix. For scenarios when source tables have column overlaps, mapping matrices provide flexibility to different possibilities of schema mapping in typical DI scenarios.

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.

[2]https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

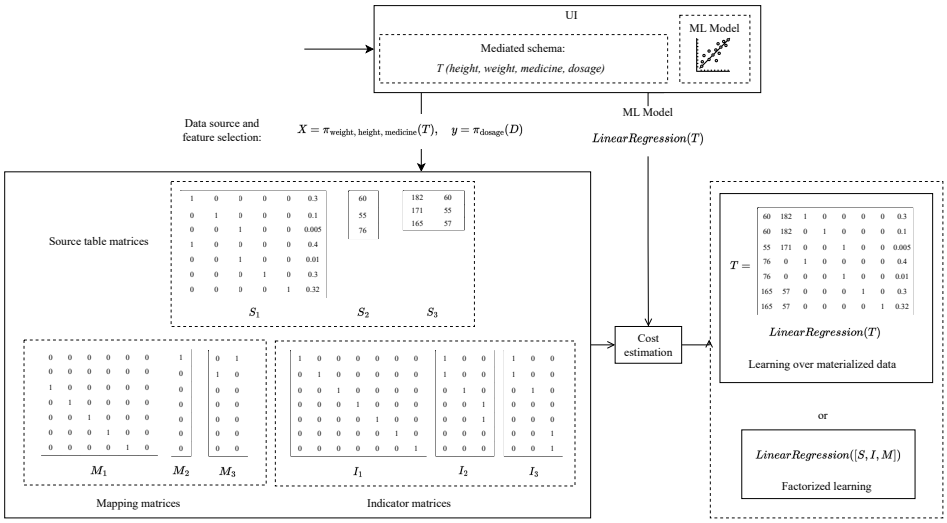

Figure 5: Ilargi workflow and intermediate results of running example.

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:

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

We denote the number of matrices as n, and the number of possible parenthesizations as $P(n)$ that can be computed as following [11].

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

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[3] for SciPy matrices, computed using matrix dimensions.

Applicability, assumptions, and extensibility. Fig. 3 summarizes Ilargi's applicability compared to State-of-the-art solution [7]. Ilargi can handle both normalized and denormalized tabular data. We follow the common data integration assumptions: for each column in the target table source, we can find its corresponding column from at least one source schema. We follow the common data science assumption that the normal preprocessing steps include null value replacement and transformation of categorical variables to numerical values. Our running example and current implementation replace null values to zero, and apply one-hot-encoding for categorical variables. It is straightforward to extend the implementation to replace the null values to mean/median values or user-specified values, and other methods for handling categorical variables [30].

3.3 Is Cost Estimation Necessary?

To verify whether cost estimation is required to decide whether to materialize or to factorize, we test Illargi with seven real datasets that the state-of-the-art [37, 51, 7, 41] use (detailed in Table 4). Each dataset consists of two or more tables and is connected in a star schema with PK-FK relationships. We then train four ML models: *i*) linear regression, *ii*) logistic regression, *iii*) Gaussian Non-Negative Matrix Factorization (GNMF) and *iv*) K-Means. During our experiments we have verified that factorization does not affect ML model performance.

Fig. 6 shows the comparison of model training time in materialization and factorization, for different join types, datasets, and models[4]. We elaborate on the datasets and experiment setting in Sec. 5.1. We observe that with the dataset `books` in the first row of figures in Fig. 6, materialization is faster or similar to factorization, in all join cases and tested ML models. It is the opposite for the datasets `movie` and `yelp`: factorization is faster over all cases (the fifth and last rows of Fig. 6). It is more complex for the rest of datasets `expedia`, `flights`, `lastfm` and `walmart`: depending on the join type and ML model, sometimes materialization is faster than factorization or vice versa.

> ***Experiment Takeaway:*** *The choice of factorization vs. materialization cannot be known in advance: cost estimation is required.*

We illustrate the problem of cost estimation intuitively in Fig. 7. Assume that there exists a borderline (the curvy purple line), between the cases where factorization is faster and the cases where materialization is faster. Areas I and II cover the cases when it is easy to decide on factorization or materialization, respectively, while area III covers the harder cases. The state-of-the-art solution [7] is a conservative method, which mainly resolves the cases in Area I, missing many potential cases in Area III where factorization

[3] https://numpy.org/doc/stable/reference/generated/numpy.linalg.multi_dot.html

[4] For now, ignore the Cost Est. bar - we revisit cost estimation time in Sec. 5.

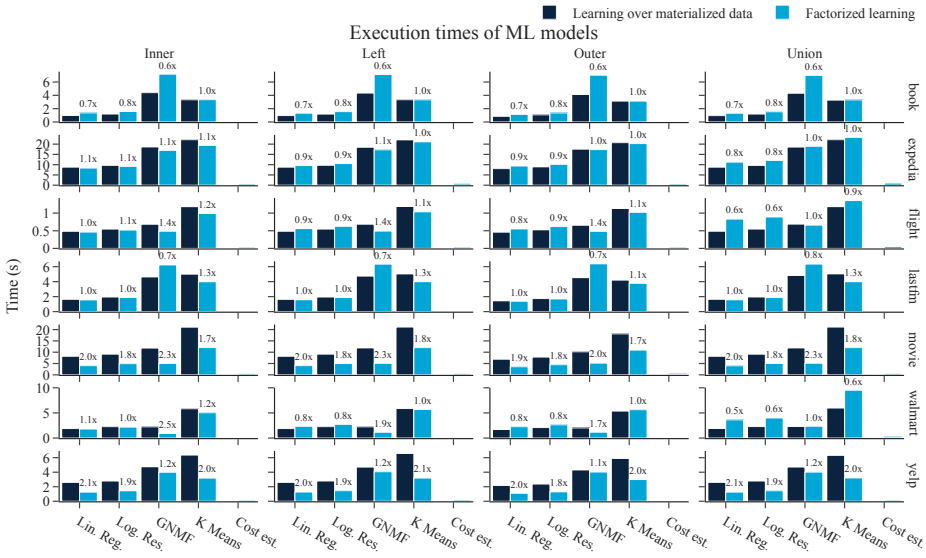

Figure 6: Models training time over seven real datasets: materialization vs. factorization

is faster. We show this also through experiment results in Sec. 5.3. To this end, in the next section we show how Ilargi makes cost estimations to decide whether to factorize or to materialize.

4 COST ESTIMATION IN ILARGI

We explain Ilargi's optimizer depicted in Fig. 8, which performs cost estimation given source datasets, DI metadata, and the ML model.

4.1 Schema Mappings as Pruning Rules

Formal framework. We use schema mappings to specify the schema-wise dataset relationships in our system. A *schema mapping* $\mathcal{M}$ between source schemas $\mathbf{S}$ and target schema T is a triple $\mathcal{M} = \langle \mathbf{S}, T, \Sigma \rangle$, where Σ is a set of dependencies over $\langle \mathbf{S}, T \rangle$. The dependencies Σ can be expressed as tgds, as shown in Table 1. We refer to an instance I over $\mathbf{S}$ as the source instance, and an instance J over T as the target instance. Intuitively, if storing the source instance I takes more space compared to J, The optimizer considers it more source redundancy and recommends materialization, and vice versa. During implementation and preliminary experiments, we observed that factorization brings extra computation overhead. Thus, if the sizes of source instances and target instances are the same, our optimizer will recommend materialization.

Special case. Consider the union example in Table 1. Following the table unionability definition in [48, 46, 33], here we say the instances from two source tables S_1 and S_2 are union-compatible if: 1) S_1 and S_2 have the same number of feature columns; 2) each corresponding pair of feature columns have the same domain. Both S_1 and S_2 have the same set of columns $\{m, a, hr, o\}$ that also exist in the target table T, which are later used as features. Note that a source table may have non-feature columns, e.g., $S_2.dd$, which will be dropped during preprocessing. Each matched attributes in the common feature set, e.g., m, have the same semantics and domain. We analyze the redundancy of unionable source tables in two cases. 1) if S_1 and S_2 have no row overlap (entity resolution result is empty): the sizes of source instances and target instances are the same. The optimizer will recommend materialization.
2) if S_1 and S_2 have row overlaps (entity resolution result is not empty): the sizes of source instances are larger than target instances. The optimizer will recommend materialization.

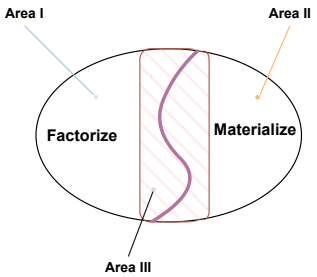

Figure 7: An abstraction of different decision areas

The union example is a typical illustration for area II in Fig. 7. This is, when source tables are unionable, materialization is more efficient. In the logical level, we can describe such relationships as tgds whose LHS contains only predicates from a single source. For instance, in Example 4 $LHS(m_2) = S_1$, $LHS(m_3) = S_2$. We derived the following rule for favoring materialization solely based on schema mappings.

Proposition 3. *If a data integration scenario can be expressed by one or a set of tgds whose LHS only contains predicates from a single source table, materialization is more efficient than factorization.*

For the rest of the cases of Example 1-3 in Table 1, we have more complicated tgds, i.e., the left-hand-side of the implication contains multiple predicates of source tables. The relative comparison between target redundancy and source redundancy also depends on the row matching. Tgds alone are not sufficient to tell apart the three areas in Fig. 4. We propose a more general method based on complexity analysis in Sec. 4.2 for area III in Fig. 7.

Symbol	Meaning
S_k	The k-th source table
T	Target table
I_k	The indicator matrix for S_k
M_k	The mapping matrix for S_k
j_T	The join type of T (inner/left/outer/union)
r_k/r_T	Number of rows in S_i/T
c_k/c_T	Number of columns in S_i/T
$m_k/m_T/m_X/m_w$	Number of nonzero elements in $S_i/T/X/w$

Table 2: Notations used in the paper

Possible extensions to EGDs. Tgds and equality generating dependencies (egds) are the two major types of database dependencies. Keys and functional dependencies (FDs) can be expressed as 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 $\varphi(\mathbf{x})$ [19]. We also extend our formal framework $\mathcal{M} = \langle \mathbf{S}, T, \Sigma \rangle$. That is, Σ is a set of three types dependencies: s-t tgds, source egds and target edgs. Source and target egds could be functional dependencies over source and target tables, respectively. Extending our logical pruning rules to egds is an abroad topic deserving dedicated study. Here we merely demonstrate such a possibility with the following example. We omit universal variables since the context is clear.

Example 4.1. *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:

$f_1 : S_1.x_1 S_1.x_2 \rightarrow S_1.x_3 \implies S_1(a, b, c_1) \wedge S_1(a, b, c_2) \rightarrow c_1 = c_2$

$f_2 : S_2.x_2 \rightarrow S_2.x_4 \implies S_2(d, e_1) \wedge S_2(d, e_2) \rightarrow c_1 = c_2$

Target egds over target table T:

$f_3 : x_1 x_2 \rightarrow x_3 \implies T(a, b, c_1, d_3) \wedge T(a, b, c_2, d_4) \rightarrow c_1 = c_2$

$f_4 : x_1 x_2 \rightarrow x_4 \implies T(a, b, c_3, d_1) \wedge T(a, b, c_4, d_2) \rightarrow d_1 = d_2$

$f_5 : x_2 \rightarrow x_4 \implies T(a_1, b, c_1, d_1) \wedge T(a_2, b, c_2, d_2) \rightarrow d_1 = d_2$

At the logical level, with the given FDs, it is easy to tell there is more redundancy in the target table than in source tables, and the optimizer will choose factorization, i.e., area I in Fig. 7.

Implementation, applicability, and limitations. Tgds are first-order sentences. We currently implement tgds in Python, which can also be implemented in other query languages (e.g., SQL) or programming languages (e.g., Java). Given a legacy data integration system, where schema matching and mapping are semi-automatically or manually obtained, the aforementioned logical rules are stored in our system's metadata catalog [25]. The optimizer can run such schema-only rules quickly even without performing more costly estimation, i.e., serving as an early-pruning rule. The limitation of the mapping-only method is that it cannot cover all the cases, for which we propose a complexity-based approach next.

4.2 Complexity Analysis of LA Operations

We conduct a comparative analysis of materialization and factorization regarding the computational complexity of LA operations. For materialization, we consider the total number of computations involved in performing an LA operation on the materialized table. For factorization, we consider the number of operations involved in performing an LA operation on each source table. Below we continue to use LMM for the explanation.

Operation	Materialization	Factorization
$T \oslash x$	$f(T)$	$\sum_{k=1}^{K} m_k$
$T^T \oslash x$		
$f(T^T)$		
rowSum(T)	m_T	
rowSum(T^T)		
colSum(T)		
colSum(T^T)		
TX	$c_X \cdot m_T + r_T \cdot m_X$	$\sum_{k=1}^{K} c_X \cdot m_k + r_k \cdot m_X$
$T^T X$	$c_T \cdot m_X + c_X \cdot m_T$	$\sum_{k=1}^{K} c_k \cdot m_X + c_X \cdot m_k$
XT	$c_T \cdot m_X + r_X \cdot m_T$	$\sum_{k=1}^{K} c_k \cdot m_X + r_X \cdot m_k$
XT^T	$r_X + m_T + r_T \cdot m_X$	$\sum_{k=1}^{K} r_X + m_k + r_k \cdot m_X$

Table 3: LA operator computations cost comparison

Complexity analysis in existing works. In [7], a straightforward complexity analysis is provided. For each LA operator, the decision of factorization and materialization is based on the arithmetic computation complexity, i.e., the size comparison between joinable tables and materialized table. For instance, given a matrix T with the shape $r_T \times c_T$, and a matrix X with the shape $c_T \cdot c_X$ ($r_X = c_T$), the complexity of their left matrix multiplication TX is $c_X \cdot c_T \cdot r_T$. Thus, in [7] the complexity of computing TX for materialization is $c_X \cdot c_T \cdot r_T$, while for factorization is $\sum_{k=1}^{K} c_X \cdot c_k \cdot r_k$.

Complexity analysis in Ilargi. Table 3 summarizes the cost comparison between materialization and factorization cases. At the logical level, the choice between factorization and materialization depends on the relative source and target redundancy, as discussed in Sec 2.4. However, when it comes to lower-level arithmetic operation computation, the sparsity of the matrices also matters. A *sparse matrix* is a matrix whose number of nonzero elements is negligible compared to the number of zeros, while a *dense matrix* has more non-zero elements. Given two matrices A and B, the naive matrix multiplication complexity is $O(r_A \cdot c_A \cdot c_B)$ as in [7]. When matrices are stored in sparse format, the complexity AB is $O(c_B \cdot m_A + r_A \cdot m_B)$, where m_A, m_B are the numbers of nonzero elements in matrix A and B [27]. The matrix form of the source and target tables can be sparse or dense. As discussed in Sec. 3.1, our mapping and indicator matrices are sparse. Therefore, We compute the computation cost as in Table 3. We also note the complexity of transposed operations. As described in Section 3.2, $T^T X$ is actually defined as $(X^T T)^T$ and XT^T is actually defined as $(TX^T)^T$. Here, we ignore the complexity added by the transposes, as these are not executed on data matrices. Notably, the differentiation between sparse and dense matrices is more about an intuition rather than a rigorous measurement [27]. Take the example of TX for materialization, and we have $m_T \leq r_T \cdot c_T$. When the target table matrix is

dense, the non-zero element number m_T is close to the size of T, i.e., $r_T \cdot c_T$. The time for multiplying T and X is at most $O(r_T \cdot c_T \cdot c_X)$.

Computing the sparsity of T. Some elements in the complexity analysis can be inferred directly from the source tables, while others require estimation. Elements which can be directly inferred include r_X, c_X, r_k, c_k and m_k, as these are stored in SciPy objects. The elements r_T and c_T cannot be inferred from the source tables directly, but can be inferred from the indicator matrices and mapping matrices, respectively. The element that requires the most complicated computations is m_T, which depends upon the source tables' sparsities, the indicator matrix and the mapping matrix. In order to compute m_T, we compute T using the materialization computation $T \leftarrow I_1 S_1 M_1^T + ... + I_K S_K M_K^T$. Then, we compute the sparsity directly from the result of this computation.

In preliminary experiments, we tried avoiding materialization and estimating sparsity, leading to unsatisfactory effectiveness results. Instead, we discovered that table materialization time is omittable compared to model training time, as elaborated in Sec. 5.4. Therefore, we chose to materialize the target table for a more accurate sparsity value. The goal of this work is not to avoid materialization [37]. Instead, the primary goal of this research is to ascertain whether, for a set of given source tables and learning tasks, the training of a model is expedited by conducting it over the source tables or through the use of a materialized target table.

4.3 Complexity Analysis of ML Models

With the above complexity analysis of LA operations, we estimate the complexity of ML models. The complexity analysis on the model level is model-specific, depending on which linear algebras are involved in the model. The speedup brought by factorization is only applicable for a subset of LA operators [7]. Thus, here the discussion at the model level also focuses on such LA operators that bring speeds. We explain the complexity analysis of linear and logistic regression, with the details of the rest models in the technical report [53].

Linear and logistic regression. We define the complexity of linear and logistic regression on target table matrix T below. The shape of weights vector w is $c_T \times 1$. We make the common assumption that w is dense, then $m_w = r_w \times c_w$. Matrix X denotes an intermediate result in the linear regression algorithm, of which the size is $r_T \times 1$. We assume this intermediate result is also dense, so $m_X = r_X \times c_X$. First, we define the complexity in the materialized case.

$$O_{\text{materialized}}(T) = \underbrace{c_w \cdot m_T + r_T \cdot m_w}_{Tw} + \underbrace{c_T \cdot m_X + m_T}_{T^T X}$$

Next, we define the complexity of the factorized case.

$$O_{\text{factorized}}(T) = \underbrace{\sum_{k=1}^{K} c_w \cdot m_k + r_k \cdot m_w}_{Tw} + \underbrace{\sum_{k=1}^{K} c_k \cdot m_X + m_k}_{T^T X}$$

Complexity ratio. We define a variable *complexity ratio* to indicate whether materialization or factorization leads to more redundancy.

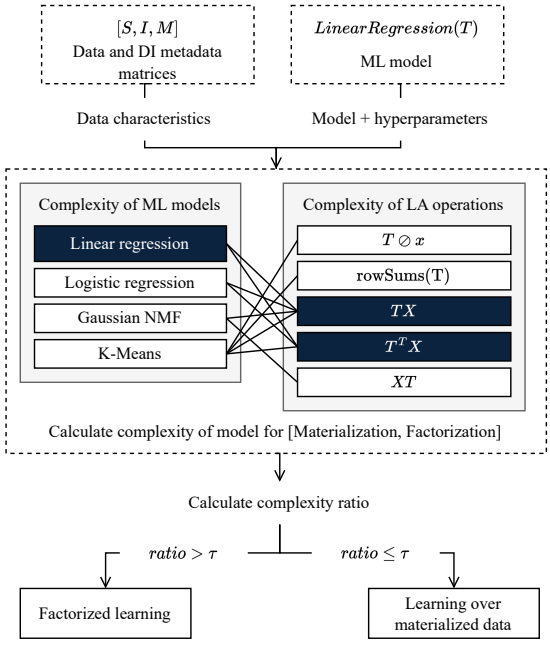

Figure 8: Cost estimation pipeline of Ilargi.

The complexity ratio is measured as the ratio of the materialization complexity divided by the factorization complexity.

$$ratio = \frac{O_{\text{materialization}}(T)}{O_{\text{factorization}}(T)}$$

Cost estimation pipeline. Fig. 8 shows how Ilargi's optimizer works. Due to space limitation, we do not list all operators for each model. The input to the cost estimation procedure are the ML model and source table matrices, mapping matrices and indicator matrices. To perform cost estimation for general data integration scenarios, we do not make special assumptions about the characteristics of input data but infer them from the data matrices. These characteristics include the dimensions and sparsities of the source tables and the dimensions and sparsity of the materialized table. In addition, we consider the ML model and model hyperparameters selected by the user. Based on these data characteristics and data integration metadata, and model hyperparameters, we calculate LA operation complexities as described in Sec. 4.3, which are aggregated to the complexity of ML models in Sec. 4.3.[5] With the obtained value of the complexity ratio, we compare it with threshold τ. In Sec. 5.2 we discuss how empirical threshold value τ is obtained. If the complexity ratio is higher than the threshold τ, Ilargi trains the model in a factorization manner, as explained in Sec. 3.2; otherwise, the source tables will be joined and the model will be trained over the materialized target table.

5 EVALUATION

5.1 Datasets and Setup

Real datasets. As shown in Table 4 we use the same datasets from the state-of-the-art system [37, 51, 7, 41]. These datasets developed

[5] Hyperparameters such as rank r for GNMF and the number of clusters k for KMeans, are included in cost estimation. Other hyperparameters such as the learning rate γ for linear and logistic regression, do not influence cost estimation.

dataset	r_T	c_T	$nnz(T)$	n_S	$nnz(S)$	r_S	c_S
book	253120	81663	2024960	2	83628	27876	28022
					249860	49972	53641
expedia	942142	52282	28263069	3	5652852	942142	27
					107451	11939	12013
					555315	37021	40242
flight	66548	13669	1385834	4	55301	66548	20
					3240	540	718
					22169	3167	6464
					22190	3170	6467
lastfm	343747	55252	4468711	2	39992	4999	5019
					250000	50000	50233
movie	1000209	13348	27005643	2	30200	6040	9509
					81532	3706	3839
walmart	421570	2441	5901781	3	421570	421570	1
					23400	2340	2387
					135	45	53
yelp	215879	55606	8635160	2	380655	11535	11706
					307111	43873	43900

Table 4: Characteristics of real datasets.

for Project Hamlet[6] [37, 51] contain seven real-world datasets. Each dataset consists of two or more tables and is connected in a star schema with PK-FK relationships. *nnz* indicates the number of nonzero elements.

Synthetic datasets. To examine our proposed cost estimation method with a wide range of data characteristics, we designed a data generator to populate synthetic datasets. Existing benchmarks and data generators, either do not support machine learning tasks [3, 1], or cannot provide the data integration scenarios for our problem setting [43]. The parameters of the generator include the row/column numbers of the source and target tables, row/column overlaps between source and table tables, table sparsity (nonzero value ratio), join type, etc. Our data generator is able to generate scenarios including source redundancy, target redundancy or both, and scenarios with sparsity both as a result of the data and/or sparsity as a result of a join. It covers inner join, left outer join, full outer join, and union. More implementation details about our data generator can be found in the technical report [53].

Hardware & Software. We run our experiments with 16 cores and 48 cores of AMD EPYC 7H12 CPU. The system uses Ubuntu 20.04.4 LTS as the OS. Each experiment is run on one CPU. We use Python 3.10.4, SciPy 1.8.0 and NumPy 1.22.4. To enable parallelized sparse matrix multiplication, we choose MKL as the backend of NumPy. The number of iterations of model training is 20.

5.2 Effectiveness of Illargi's Optimizer

We examine the effectiveness of our cost estimation method in Sec. 4. We generate synthetic data with varying values of complexity ratios and observe its correlation with the actual speedup of factorization against materialization. We first examine the performance at LA operator level, and then the ML model level.

[6]Project Hamlet datasets: https://adalabucsd.github.io/hamlet.html

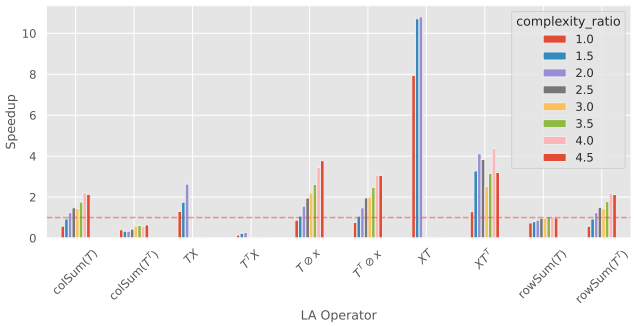

Figure 9: Speedups of factorized operators against materialzed operators with varying complexity ratios. The red horizontal line is speedup=1 (no speedup).

Figure 10: Speedups of factorized operators against materialized operators (complexity ratio is 1.5).

Cost estimation at LA operator level. In Sec. 4.2, we have analyzed the complexity comparison of materialization against factorization of a single LA operator. Fig. 9 illustrates the speedup of factorization against materialization per LA operator. First, most operators, present an increasing trend of speedups as the complexity ratio ($\frac{O_{\text{materialization}}}{O_{\text{factorization}}}$, based on Table 3) increases. $colSum(T^T)$, T^TX, and $rowSum(T)$ do not exhibit significant speedup in comparison to materialization due to the overhead of transposing CSR matrices.

Establishing Threshold τ. For each operator, we can find an empirical threshold where the factorized operator starts to outperform the materialized counterpart. For example, when the complexity ratio is greater than 1.5, for the $colSum(T)$ operator, factorization is a better choice. The threshold values vary with each operator.

Impact of join types. In addition to analyzing the impact of complexity ratio, we also examined the influence of join types on speedup. As illustrated in Fig. 10, the speedup is lower in outer joins. This can be attributed to the fact that, given two tables, outer joins produce larger target tables compared to inner and left joins. Consequently, this leads to higher computational complexity for operators that involve outer joins.

Cost estimation at ML model level. Following the speedup evaluation for operators, Fig. 11 illustrates the speedup of factorization for machine learning model training. The speedup of model training increases with complexity ratio. Notably, the GaussianNMF model shows a more significant speedup compared to others. Most

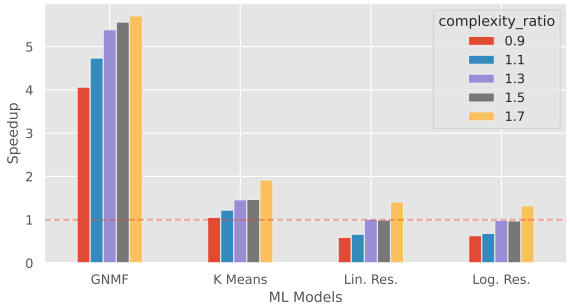

Figure 11: Training speedups with varying complexity ratios.

Figure 12: Mean speedups (factorization is faster) and slowdowns (materialization is faster) of Morpheus: correct vs. incorrect estimations (red bar: FN).

of the operators in GaussianNMF model are left and right matrix multiplications, TX and XT, which have significant speedups in Fig. 9. The large number of *colSum* and *rowSum* operators in regression models and KMeans, do not benefit from the acceleration of factorization. The attainable speedup of model training depends on particular operators of the model.

Takeaways:
i) The speedup up of factorized model training depends on the model's operator composition.
ii) Our cost estimation is more effective for large complexity ratios.

5.3 Comparison to Morpheus

Our baseline is the cost estimation procedure in Morpheus [7, 29], which considers the tuple ratio ($TR = \frac{r_i}{r_j}$) and feature ratio ($FR = \frac{c_i}{c_j}$) between the two join tables S_i and S_j. To make a fair comparison, we use the same threshold values as [7]. We test over synthetic datasets described in Sec. 5.1, and evaluate whether the predictions of Morpheus and Ilargi make the correct prediction of factorization and materialization. We run the cost estimation methods of Morpheus and Ilargi to make a prediction, and then train models to see whether the predictions were correct. There are four distinct cases to measure here: *i*) true positive (TP, the model predicted speedup and speedup was observed), *ii*) true negatives, (TN, the model predicted slowdown, and slowdown was observed), *iii*) false negative (FN, the model predicted slowdown but speedup was observed) and *iv*) false positives (FP, the model predicted speedup but slowdown was observed).

Morpheus. In Fig. 12 the mean speedup for true positives is higher than the mean speedup for false negatives. We do not see any false positives. Morpheus' cost estimation method is conservative: many cases are missed when there is a speedup of factorization.

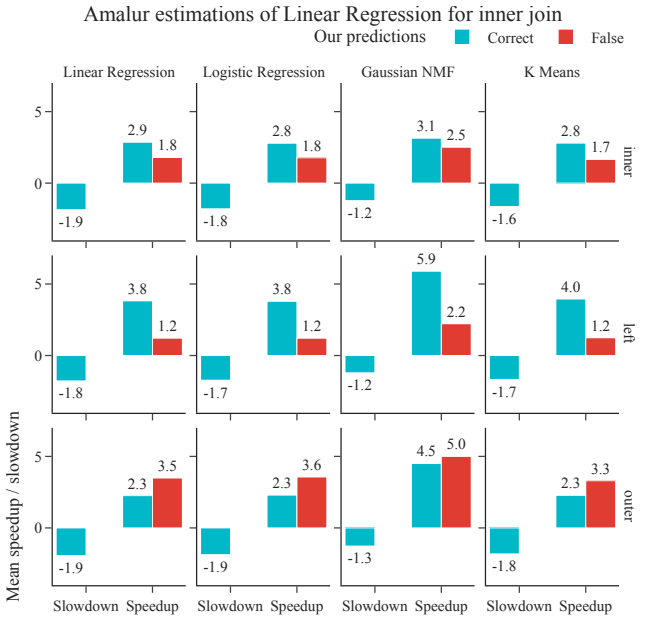

Figure 13: Mean speedups of Ilargi : correct vs. incorrect estimations (red bar: FN).

Ilargi. As shown in Fig. 13, our cost estimations, like Morpheus, do not include false positives. We observe that the mean speedup for true positives is higher than the mean speedup for false negatives for inner joins and left joins, while for outer joins we see the opposite. We omitted the results of union scenarios here, for which materialization is *always* faster. Overall, the mean speedup for true positives is higher than the mean speedup for false negatives. For left joins, the difference in mean speedup between true positives and false negatives is larger than for the inner joins. Notably, when the speedup or slowdown is small, e.g., 1.2 in the first sub-figure of the second row, the training time difference between materialization and factorization may be insignificant.

Takeaways:
i) Compared to the baseline, Ilargi is more effective in the cases of slowdowns and high speedups, and less effective for small speedups.
ii) Over the common case of inner join, Ilargi demonstrates a mild improvement of TN/FN compared to Morpheus.
iii) Ilargi covers more scenarios than baseline. It is more effective for inner join and left join scenarios.

5.4 Overhead of Cost Estimation

Our cost estimation method requires obtaining parameters about the size and sparsity of each source table S_k and the target table T, and the size of intermediate results. These parameters can be directly obtained from the source table matrix, except for the sparsity of T that needs to be calculated after performing materialization. That is, the overhead of cost estimation includes target table materialization and complexity ratio calculation, with the time duration dominated by table materialization. We inspect the time of our cost

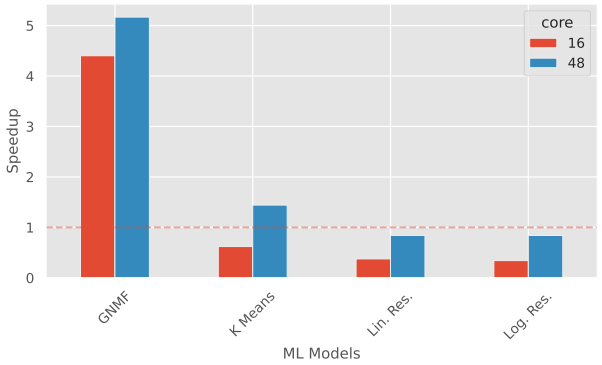

Figure 14: The speedup of factorization over materialization, for training executed on different numbers of CPU cores.

estimation approach. In each sub-figure of Fig. 6, the last bar shows the cost estimation overhead. We observe that compared to the training time of ML model with iterations, the overhead of our cost estimation time is negligible. In our experiments, we set the iteration number as 20 (the same as [7]) , which is a relatively low number of iterations compared to the number of iterations usually required for an ML model to converge.

Takeaways: *Ilargi's cost estimation overhead is negligible.*

5.5 Outlook: Hardware-aware Cost Modeling

Given that many linear algebraic operators within the model training process can be efficiently parallelized, we investigate the impact of parallelism on LA operators and model training tasks, and focus on the parameter of number of CPU cores. Notably, hardware-aware cost estimation is a complex topic itself, requiring intensive empirical studies, which is not our focus here. Here we explore hardware as a relevant aspect for the choice of factorization and materialization for the sake of completeness.

GNMF Models. Fig. 14 shows the speedup of factorization against materialization when training the models with 16 or 48 CPU cores. We observe that GNMF exhibits higher speedup even with low parallelism settings. The reason is that GNMF has multiple matrix multiplication operators, which dominate the computation of GNMF training.

Regression Models. For linear regression and logistic regression, in this experiment, materialization is faster. These two regression models contain a large proportion of element-wise operators (e.g., $T \oslash x$) and reduction operators (e.g., $colSum(T)$), for which the speedup of factorization with parallelized implementation is small.

K-Means. Interestingly, for KMeans the relative performance of factorization and materialization changes, as parallelism increases. Therefore, in addition to the complexity ratio evaluated in Sec. 4, the interactions between operator composition and hardware parallelism also influence the decision boundary of factorized and materialized learning. This observation suggests that further research should focus on developing comprehensive cost models that incorporate hardware-related factors.

Takeaways: *i) Hardware parallelism indeed affects the speedup of factorized learning against materialized learning.*
ii) High parallelism is particularly beneficial to models with intensive matrix multiplication operators.
iii) The specific speedup depends not only on the complexity ratio but also on the operator composition of the model training.

6 RELATED WORK

Avoiding joins. As discussed in Sec. 2, joins can lead to data redundancy. Therefore, earlier work has investigated whether joins can be avoided before training ML models [37, 51, 55]. Avoiding a join, means training a model on a single base table instead of multiple joinable tables. Hamlet [37] showed that for data with PK-FK relationships, we often do not need to join other tables as long as we keep the FKs in the base table. This notion is extended to non-linear models [51], and these models are even more resilient to avoid joins. Another approach for reducing the cost of materialization is to return a random sample of the join, and train ML models on this sample, which was also proven to be an effective method [55]. This line of work is orthogonal to this work, since in our problem setting, useful features are scattered in different tables, and combining them is necessary for training ML models.

Factorized learning. Existing works [34, 35, 49, 31, 50] on factorized learning, manually rewrite individual ML models to their factorized versions. Morpheus and it successors [40, 29, 8] provide a general, automated framework of LA rewrite rules for linear models. We have intensively compared our approach with the state-of-the-art approach Morpheus [7]. Illargi is applicable in more general data integration scenarios for ML use cases than Morpheus. Moreover, Morpheus [7] proposed a cost estimation using heuristics based on the dimensions of the source tables. In the Python extension of Morpheus [42], although sparsity is mentioned as a factor for arithmetic calculation overhead, it is not included in cost estimation as in our approach.

7 CONCLUSION

In this paper, we address the challenge of training machine learning models over disparate data sources. We introduce a formal framework based on tgds that links existing factorization approaches based on inclusion dependencies and join dependencies with data integration scenarios specified by schema mappings. Ilargi leverages matrices to encode schema mapping and row matching information enabling a unified computation of data transformation and linear algebra computation. Moreover, Ilargi utilizes schema mappings as pruning rules and complexity ratios to choose between materialization and factorization. Our experiments demonstrate the effectiveness and efficiency of our cost estimation method, which covers more scenarios than state-of-the-art solutions.

Outlook. Future research can build on our work by exploring theoretical directions in line with existing data integration theory [39, 32, 24]. An interesting research direction would be to extend our work to non-linear models [40]. With recent advancements in hardware [45, 28, 52], there is potential to improve the efficiency of training and inference for ML models across silos and enhance the accuracy of cost estimation.

REFERENCES

[1] Bogdan Alexe, Wang Chiew Tan, and Yannis Velegrakis. "STBenchmark: towards a benchmark for mapping systems". In: *PVLDB* 1.1 (2008), pp. 230–244.

[2] R. Alotaibi et al. "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries". In: *SIGMOD*. 2021, pp. 23–35.

[3] Patricia C Arocena et al. "The iBench integration metadata generator". In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 108–119.

[4] C. Beeri and M. Y. Vardi. "A proof procedure for data dependencies". In: *JACM* 31.4 (1984), pp. 718–741.

[5] Jacqueline M Bos et al. "Prediction of clinically relevant adverse drug events in surgical patients". In: *PloS one* 13.8 (2018), e0201645.

[6] David Guy Brizan and Abdullah Uz Tansel. "A. survey of entity resolution and record linkage methodologies". In: *Communications of the IIMA* 6.3 (2006), p. 5.

[7] L. Chen et al. "Towards Linear Algebra over Normalized Data". In: *PVLDB* 10.11 (2017).

[8] Zhaoyue Cheng et al. "Efficient Construction of Nonlinear Models over Normalized Data". In: Apr. 2021, pp. 1140–1151.

[9] N. Chepurko et al. "ARDA: automatic relational data augmentation for machine learning". In: *VLDB* 13.9 (2020), pp. 1373–1387.

[10] Rada Chirkova, Jun Yang, et al. "Materialized views". In: *Foundations and Trends® in Databases* 4.4 (2012), pp. 295–405.

[11] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[12] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[13] Alin Deutsch, Lucian Popa, and Val Tannen. "Query reformulation with constraints". In: *ACM SIGMOD Record* 35.1 (2006), pp. 65–73.

[14] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[15] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. "AIDA: abstraction for advanced in-database analytics". en. In: *Proceedings of the VLDB Endowment* 11.11 (July 2018), pp. 1400–1413.

[16] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. "Making an RDBMS Data Scientist Friendly: Advanced In-database Interactive Analytics with Visualization Support". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1930–1933.

[17] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. "COCOA: COrrelation COefficient-Aware Data Augmentation." In: *EDBT*. 2021, pp. 331–336.

[18] R. Fagin. "Tuple-Generating Dependencies". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 3201–3202.

[19] Ronald Fagin. "Equality-Generating Dependencies". In: *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, pp. 1009–1010.

[20] Ronald Fagin, Phokion G Kolaitis, and Lucian Popa. "Data exchange: getting to the core". In: *ACM Transactions on Database Systems (TODS)* 30.1 (2005), pp. 174–210.

[21] Ronald Fagin and M Vardi. "The theory of data dependencies". In: *Mathematics of Information Processing* 34 (1986), p. 19.

[22] Ronald Fagin et al. "Clio: Schema mapping creation and data exchange". In: *ER*. Springer, 2009, pp. 198–236.

[23] Ariel Fuxman and Renée J. Miller. "Schema Mapping". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 2481–2488.

[24] Behzad Golshan et al. "Data Integration: After the Teenage Years". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA*. 2017, pp. 101–106.

[25] Rihan Hai et al. "Amalur: Data Integration Meets Machine Learning". In: *ICDE*. 2023, To appear.

[26] Andrew Hard et al. "Federated learning for mobile keyboard prediction". In: *arXiv preprint arXiv:1811.03604* (2018).

[27] Ellis Horowitz and Sartaj Sahni. "Fundamentals of data structures". In: (1982).

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

[29] David Justo et al. "Towards a Polyglot Framework for Factorized ML". In: *Proceedings of the VLDB Endowment* 14.12 (July 1, 2021), pp. 2918–2931.

[30] John D Kelleher, Brian Mac Namee, and Aoife D'arcy. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT press, 2020.

[31] Mahmoud Abo Khamis et al. "AC/DC: In-Database Learning Thunderstruck". en. In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. Houston TX USA: ACM, June 2018, pp. 1–10.

[32] Phokion G Kolaitis. "Schema mappings, data exchange, and metadata management". In: *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2005, pp. 61–75.

[33] Christos Koutras et al. "Valentine: Evaluating Matching Techniques for Dataset Discovery". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2021, pp. 468–479.

[34] A. Kumar, J. Naughton, and J. M. Patel. "Learning generalized linear models over normalized data". In: *SIGMOD*. 2015, pp. 1969–1984.

[35] Arun Kumar et al. "Demonstration of Santoku: optimizing machine learning over normalized data". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1864–1867.

[36] Arun Kumar et al. "To join or not to join? thinking twice about joins before feature selection". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 19–34.

[37] Arun Kumar et al. "To Join or Not to Join?: Thinking Twice about Joins before Feature Selection". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD/PODS'16: International Conference on Management of Data. San Francisco California USA: ACM, June 14, 2016, pp. 19–34.

[38] Daniel Lee and H Sebastian Seung. "Algorithms for non-negative matrix factorization". In: *Advances in neural information processing systems* 13 (2000).

[39] Maurizio Lenzerini. "Data integration: A theoretical perspective". In: *PODS*. ACM. 2002, pp. 233–246.

[40] Side Li, Lingjiao Chen, and Arun Kumar. "Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra". In: *SIGMOD*. 2019, pp. 1571–1588.

[41] Side Li, Lingjiao Chen, and Arun Kumar. "Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam Netherlands: ACM, June 25, 2019, pp. 1571–1588.

[42] Side Li and Arun Kumar. *MorpheusPy: Factorized Machine Learning with NumPy*. Tech. rep. Technical report, 2018. Available at https://adalabucsd. github. io/papers . . .

[43] Yu Liu et al. "MLbench: benchmarking machine learning services against human experts". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1220–1232.

[44] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[45] Supun Nakandala et al. "A Tensor Compiler for Unified Machine Learning Prediction Serving". In: *OSDI 2020*. USA: USENIX Association, 2020.

[46] Fatemeh Nargesian et al. "Table union search on open data". In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 813–825.

[47] Erhard Rahm and Philip A Bernstein. "A survey of approaches to automatic schema matching". In: *the VLDB Journal* 10.4 (2001), pp. 334–350.

[48] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*. Vol. 3. McGraw-Hill New York, 2003.

[49] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. "Learning Linear Regression Models over Factorized Joins". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. Association for Computing Machinery, 2016, pp. 3–18.

[50] Maximilian Schleich et al. "A Layered Aggregate Engine for Analytics Workloads". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1642–1659.

[51] Vraj Shah, Arun Kumar, and Xiaojin Zhu. "Are Key-Foreign Key Joins Safe to Avoid When Learning High-Capacity Classifiers?" June 4, 2017. arXiv: 1704.00485 [cs].

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

[53] Wenbo Sun et al. *Efficient ML Model Training over Silos: to Factorize or to Materialize: Extended Version*. Technical report. https://www.wis.ewi.tudelft.nl/data-management/ilargi-tech-report.pdf. 2023.

[54] Qiang Yang et al. *Federated Learning*. Morgan & Claypool Publishers, 2019.

[55] Zhuoyue Zhao et al. "Random sampling over joins revisited". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1525–1539.

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 **M**, one case without I_1, and one case without both. First, we describe the case without **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 **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 **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 **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 [7]. All pseudocodes below are from [7]. 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 2. This algorithm also uses gradient descent.

Algorithm 2 Logistic regression using Gradient Descent

Input: X, y, w, γ
0: **for** $i \in 1 : n$ **do**
0: $\quad w = w - \gamma(T^T \frac{Y}{1+e^{Tw}})$
0: **end for**=0

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 3 and is based on the multiplicative update rule by [38].

Algorithm 3 Gaussian NMF

Input: X, W, H
0: **for** $i \in 1 : n$ **do**
0: $\quad H = H \times (\frac{W^T T}{W^T W H})$
0: $\quad W = W \times (\frac{TH^T}{W(HH^T)})$
0: **end for**=0

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 4.

Algorithm 4 K-Means

Input: X, k
0: 1. Initialize centroids matrix $C_{r_X \times k}$
0: $\mathbf{1}_{a \times b}$ represents a matrix filled with ones with dimension $a \times b$, it is used for replicating a vector row-wise or column-wise.
0: 2. Pre-compute l^2-norm of points for distances.
0: $D_T = \text{rowSums}(T^2)\mathbf{1}_{1 \times k}$
0: $T_2 = 2 \times T$
0: **for** $i \in 1 : n$ **do**
0: $\quad$ 3. Compute pairwise squared distances; $D_{r_X \times k}$ has points on rows and centroids/clusters o columns.
0: $\quad D = D_T - T_2 C + \mathbf{1}_{r_X \times 1}\text{colSums}(C^2)$
0: $\quad$ 4. Assign each point to the nearest centroids; $A_{r_X \times k}$ is a boolean assignment matrix.
0: $\quad A = (D == \text{rowMin}(D))\mathbf{1}_{1 \times k}$
0: $\quad$ 5. Compute new centroids; the denominator counts the number of points in the new clusters, while the numerator adds up assigned points per cluster.
0: $\quad C = (T^T A)/(\mathbf{1}_{d \times 1}\text{colSums}(A))$
0: **end for**=0

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.

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

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 **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 **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 DATA GENERATOR

Our synthetic data generator is designed based on the real datasets. The Hamlet datasets were originally only used to evaluate inner joins. For datasets book, lastfm, movie, and yelp, the entity table has no columns except for foreign keys. Since the keys are dropped from the table in the data matrix, these entity tables have no columns and are therefore not included in the table characteristics. To be able to evaluate meaningful left outer and full outer joins, we remove rows from selected tables in each dataset. For an inner join, we remove data from one of the attribute tables. For the Expedia dataset we remove data from table S_3 *Searches*. For the Flights dataset we remove data from table S_4 *destAirports*. For the datasets Movies, LastFM, Yelp and Books we remove data from the table *Users* in each dataset.

Parameters and join types. Consider the case we generate two source tables. We generate one table S_1 and one table S_2. We generate datasets where the number of rows in target table T, $r_T \in [10, 20, 50, \ldots, 10000, 20000, 50000]$, the number of columns $c_T = [100, 1000]$, the number of columns in S_1 as a percentage of the number of columns in T, $q_{c_1} = 0.1$ and the sparsity of S_1, $p_1 = 0.1$. In addition, we vary the number of columns in S_2 as a percentage of the number of columns in T, q_{c_2}, from $1 - q_{c_1}$ for no column overlap to $1 - q_{c_1} + 0.1$ for column overlap. With a parameter j_T, we generate union and three types of joins: the inner join, left join and full outer join.

Source redundancy and target redundancy. Our data generator is able to generate scenarios including source redundancy, target redundancy or both, and scenarios with sparsity both as a result of the data and/or sparsity as a result of a join. It covers inner join, left outer join, full outer join and union. Each scenario generated by the data generator is structured according to a star schema. We distinguish two types of source tables: entity tables S_1 and the set of attribute tables S_k $(k > 1)$. To efficiently generate data, we add keys to our source table. For simplicity purposes, we only generate one key for each of S_k which is used to join all tables together. The generated data can be used to simulate joins on any set of columns or keys.