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

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 indicates integrating source datasets before loading the join result into an ML tool, while factorization enables pushing down linear algebra operators to underlying source datasets to enhance efficiency. However, existing solutions fall short in two areas when dealing with data sources outside a single database. First, they lack adequate support for the complex relationships between source datasets and the desired target dataset, which are typically defined by schema mappings and data matching. Second, 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 source datasets. Ilargi covers a broad range of typical dataset relationships in ML use cases, representing them in matrix format, 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 an ML-based cost estimation module, Ilargi can automatically choose between materialization and factorization with 70-96% accuracy depending on the dataset. Through extensive experiments, we demonstrate that Ilargi effectively improves the time-wise efficiency of model training.

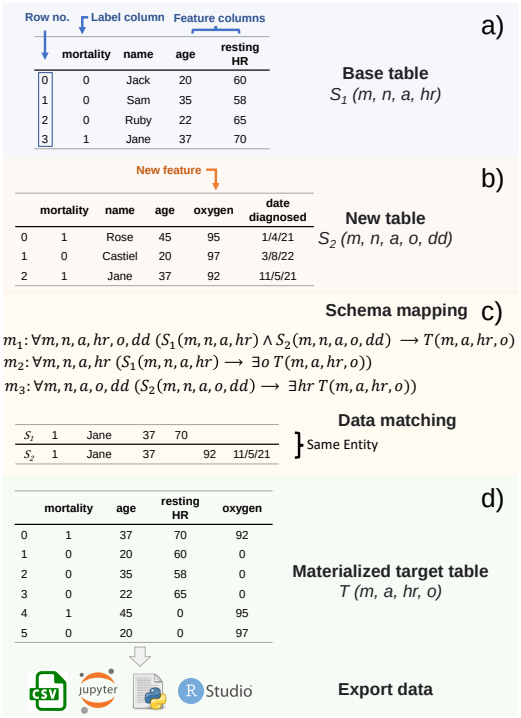

Figure 1: Data integration tasks before model training

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 datasets or the materialized join result?

Training data from different sources. The goal of ML methods is to automatically extract information from training data for accurate predictions over unseen data [24, 61]. In real world applications, data is often not stored in a central database or file system, but spread over different data silos. Take, for instance, drug risk prediction: features can reside in datasets collected from clinics, hospitals, pharmacies, and laboratories distributed geographically [12]. Another example is training models for keyboard stroke prediction: training requires data from millions of phones [40].

Data integration. 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 [74, 66]. Fig. 1 illustrates an example of predicting the mortality (binary classification) of patients based on tables, S_1 and S_2, maintained by different departments in the same hospital. *Data integration (DI)* systems facilitate interoperability among multiple, heterogeneous sources, and provide a unified view for users [39, 36]. The fundamental aspect of data integration lies in the description of data sources and their relationships [26]: *i*) mappings between different source schemata, i.e., schema matching and mapping [67, 33] and *ii*) linkages between data instances, i.e., data matching (also known as record linkage or entity resolution) [13]. We refer to such information obtained through data integration as *data integration metadata*.

Materialization as common practice. With DI metadata, the exact choice of merging source datasets may depend on the source data volume, integration scenarios, and technology stack, etc. A data scientist trains models with popular ML frameworks such as scikit-learn, PyTorch, TensorFlow, and Keras. It is straightforward to use Python libraries such as Pandas to merge datasets, when we only have a few CSV files and simple column and row matching. With large-scale datasets stored in databases or data lakes [35, 38], merging the datasets may require dedicated transformation scripts or data integration tools and services, e.g., Informatica [44], Pentaho [64], Talend [75], SSIS [73]. Data science processes are exploratory and interactive [66, 74, 70], possibly demanding more data preparation and integration tasks such as feature engineering and selection [76, 37], normalization [46], ontology matching [29, 6], and data cleaning [56, 69, 9]. All these solutions for generating the training dataset, have one thing in common: source datasets need to be merged and materialized, as shown in Fig. 1d.

Factorization as a new paradigm. Pushing ML models over joins is coined as the problem of factorized learning and has been intensively studied [51, 52, 71, 47, 16, 3, 45, 57, 19]. 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 models over multiple tables of a single

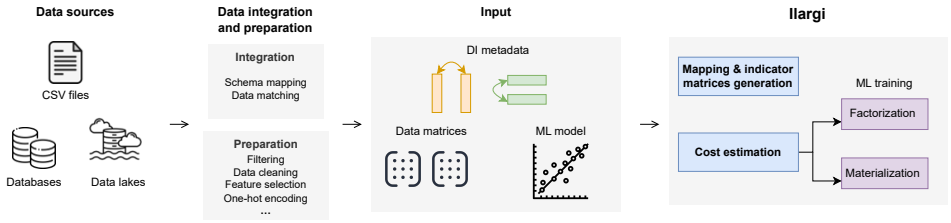

Figure 2: Ilargi's approach overview.

Figure 3: Ilargi vs. SOTA [16].

database, *factorized learning* [51] has been proposed, also known as *learning over factorized joins* [71]. The training process of an ML model requires complex arithmetic computations of linear algebra (LA) operators. Given an ML model and joinable tables of a database, a general approach [16] is proposed to reformulate LA operators and push down their computation to each table (see Sec. 2.1). Compared to materialization, factorized learning does not affect model training accuracy but can help time-wise efficiency.

Research challenges. It is natural to ask whether factorized learning can help with ML model training over *multiple* sources instead of a single database. The main challenges are two-fold:
(C1) Typical integration scenarios of multiple sources are not covered. When two input tables have a primary key-foreign key (PK-FK) relationship, the result of the join often contains redundant data. Factorized learning [16] proposes to execute the computation of LA operators over input tables instead of their join 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 ML applications. We are still missing a comprehensive and formal analysis of possible relationships between source datasets for ML use cases.
(C2) Factorized learning does not always yield speedups in integration scenarios. Factorized learning achieves speedup when joining tables leads to data redundancy in the training dataset. However, data sources may have overlapping values, e.g., a patient's personal details are recorded in two hospitals. That is, source tables may contain redundant data. The question arises: given a downstream ML model and source datasets, is it faster to train over the materialized result, or to factorize by computing LA operations within each data source? The choice is not obvious, as we will show in Sec. 5.

Our solution. In this work, we focus on a common scenario: training data comes from multiple sources. To train ML models efficiently, we address the above two challenges. i) To enable factorization over sources, we formalize possible integration scenarios and define matrix-based representations for DI metadata, such that we can develop LA rewriting rules over source datasets. ii) We leverage materialization and factorization and choose the more efficient option. We reveal that the choice depends on three factors: data (incl. data integration metadata), ML algorithms (LA operators), and hardware configuration (e.g., number of CPU cores). As shown in Fig. 2, we propose an approach that *represents* DI metadata as matrices, *computes* operations of data transformation and model training, and *optimizes* the workflow by choosing between materialization and factorization. Our contributions are as follows.

- *Problem formalization (Sec. 2).* We analyze the scenarios of integrating multiple source datasets for a training dataset with a widely used mapping formalism, i.e., tuple-generating dependencies [7, 30]. We identify research gaps for extending factorization from a single database to multiple sources as shown in Fig. 3.
- *Representation & computation (Sec. 3).* We propose *Ilargi*, a system that tensorizes DI metadata, i.e., represents schema mappings and data matching as matrices. Ilargi provides a unified computation of data transformation and linear algebra operations, which enables factorization over source datasets.
- *Optimization (Sec. 4).* We identify the relevant factors affecting the choice of materialization and factorization. We propose an estimator based on a tree boosting algorithm, which combines these factors and effectively predicts the decision boundary of materialization and factorization with up to 96% accuracy.
- *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 over both synthetic and real datasets.

2 DATA INTEGRATION AND FACTORIZATION

We first explain the existing strategies of factorization. To represent schema-level relationships in typical integration scenarios for ML use cases, we discuss our formal framework based on first-order logic. We analyze the gaps and reveal the core research questions.

2.1 Preliminary: Factorization

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. In Sec. 1 we have discussed common practices for materialization.

Factorization. Given an ML model and joinable tables of a database, factorized learning [51, 16] is the process of reformulating the execution rules of LA operations of the ML model and pushing down the computation to each table. Compared to materialization, factorized learning does not affect model training accuracy, as the arithmetic computation results of the original operators will remain the same. Similar to [16], 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* [16]. After transforming S and R into their matrix form, their row-matching relationship can

Table 1: Four example integration scenarios.

No.	Dataset Relationship	Schema mappings
1	Full outer join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(m,n,a,o,dd) \rightarrow T(m,a,hr,o))$ $m_2: \forall m,n,a,hr\ (S_1(m,n,a,hr) \rightarrow \exists o\ T(m,a,hr,o))$ $m_3: \forall m,n,a,o,dd\ (S_2(m,n,a,o,dd) \rightarrow \exists hr\ T(m,a,hr,o))$
2	Inner join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(m,n,a,o,dd) \rightarrow T(m,a,hr,o))$
3	Left join	$m_1: \forall m,n,a,hr,o,dd\ (S_1(m,n,a,hr) \wedge S_2(n,a,o,dd) \rightarrow T(m,a,hr,o))$ $m_2: \forall m,n,a,hr\ (S_1(m,n,a,hr) \rightarrow \exists o\ T(m,a,hr,o))$
4	Union	$m_2: \forall m,n,a,hr,o\ (S_1(m,n,a,hr,o) \rightarrow T(m,a,hr,o))$ $m_3: \forall m,n,a,hr,o,dd\ (S_2(m,n,a,hr,o,dd) \rightarrow T(m,a,hr,o))$

be defined as a matrix. This is referred to as *indicator matrix*, 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), 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 ([16])

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

We showcase the rewriting rules of factorizing two operators from Morpheus: 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. c_S, c_R and c_T are the column numbers of tables S, R, T.

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

Research gap. Existing factorized learning works [51, 52, 71, 47, 16, 3, 45, 57, 19] are applicable over multiple tables joinable with foreign key constraints (a special case of inclusion dependencies) or join dependencies. Inclusion dependencies and join dependencies are developed in the context of schema design given a single database schema. To describe relationships of multiple source datasets, we introduce the widely used schema mapping formalism next.

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. One of the most commonly used mapping languages is *source-to-target tuple generating dependencies (s-t tgd)* [7, 30], which are also known as Global-Local-as-View (GLAV) assertions [55].

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 $\mathbf{T}$ is a triple $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$, where Σ is a set of dependencies over $\langle \mathbf{S}, \mathbf{T} \rangle$. 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 for unifying formal framework. Besides schema mappings, tgds can be used to describe inclusion dependencies and join dependencies [32] for existing factorization works. Next, we employ tgds to describe the dataset relationships in different scenarios where source datasets are merged into the training dataset.

2.3 Integration Scenarios for Training Dataset

An *integration scenario* [5] is a triple of source schemas $\mathbf{S}$, target schema $\mathbf{T}$, and their schema mappings $\mathcal{M}$ that are typically specified by tgds. In this work, we consider the schema of the training dataset as the target schema T, and use an integration scenario to describe the dataset relationships between the source and training datasets.

Existing factorization works, including state-of-the-art [16], tackle inner joins, as elaborated in Sec. 6. Our approach covers more ways of merging source datasets, i.e., left joins, full outer joins, and unions. In Table 1, we list example integration scenarios covering these four basic types of dataset relationships. Understanding the dataset relationships is useful for many ML use case where training data comes from silos. Next, for clarity, we explain the four types of dataset relationships and the schema mappings in Table 1, through the use case of feature augmentation.

Use case. *Feature augmentation* is the exploratory process of finding new datasets and selecting features that help improve the ML model performance [20, 28, 58]. Figure 1 shows an example: starting from a base table S_1, a data scientist augments the features by introducing the table S_2 and selecting the new feature o ($oxygen$). Example 1 (full outer join) is explained in Example 2.1.

Example 2 (inner join) represents the integration scenario where only overlapped rows in two sources will be transformed, i.e., an inner join between S_1 and S_2 followed by a projection on columns m, a, hr, o. It can be used to describe the feature augmentation processes where fewer missing values are preferred.

Example 3 (left join) shows a left join between S_1 and S_2. Compared to Example 1, we slightly change the schema of S_2 by dropping the label column m. Example 3 describes another typical feature augmentation scenario for supervised learning: only the base table S_1 contains the label column. Thus, when adding features from the new table S_2, only rows overlapped with S_1 will be selected.

Example 4 (union) is a special case of Example 1, 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. Example 4 can represent the scenario when a new table is selected to bring more data samples.

2.4 Open Problems & Challenges

Representation and execution. Data sources are autonomous, and their schemata are designed independently [26]. It is non-trivial to extend existing factorization work [16] to aforementioned integration scenarios, due to the following gaps.

i) *Joins*. As shown in Table 1, the source tables can be combined in different manners depending on ML application. Only inner-joins are discussed in [16]. On the logical level, the inner-join of multiple source tables over the same join column, can be expressed as a single tgd, while the rest of the three cases may require the union of multiple tgds as in Table 1.

ii) *Row-overlap*. Inclusion dependencies, e.g., foreign key constraints, indicate a containment relationship between instances coming from two tables. In contrast, two joinable or unionable 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.

iii) *Column-overlap*. Morpheus has only discussed representation for row matching and does not have a declarative representation for complex scenarios of column overlap. For instance, tables are joined with a single join column, in PK-FK table joins or multi-table joins in [16]. As shown in Table 1, in integration scenarios, it is possible for two source tables to share more than one common attribute as the result of schema matching.

To summarize, the dataset relationships in integration scenarios are more complicated than PK-FK or multi-table joins in the problem setting of Morpheus [16], which leads to the below question.

Research question Q1

How to represent and utilize data integration metadata, e.g., schema mapping and data matching, to enable factorization over different integration scenarios?

Optimization. To choose between materialization or factorization, 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 functional dependencies, join dependencies, and inclusion dependencies. Factorized learning works the same way: when joinable tables have PK-FK relationship, the join result (i.e., the target table) may have more data redundancy. We refer to such redundancy as *target redundancy*. This is the reason why, when 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

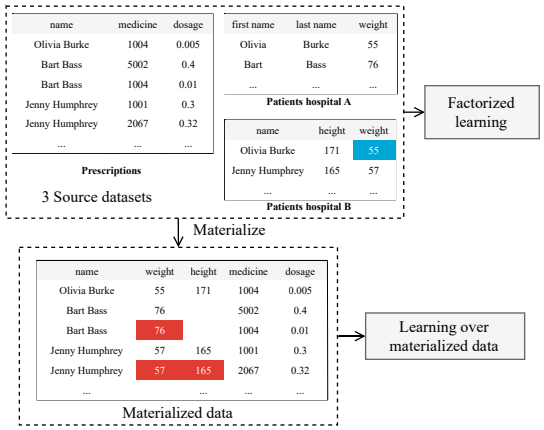

Figure 4: Source redundancy (blue) vs. target redundancy (red).

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: for the patient *Olivia Burke*, their record of weight exist in two hospital databases. As shown in Fig. 3, we aim to tackle both types of redundancies.

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 or null value replacement. In integration scenarios, an instance in the target table may have missing values, if no corresponding values exist in the source table. In Sec. 4, we will reveal that the sparsity plays an important role in choosing between materialization and factorization.

Research question Q2

Given source datasets and an ML algorithm, how to choose between materialization and factorization?

3 ILARGI: ML OVER DISPARATE SOURCES

For research question $Q1$, we propose matrix-based representations of DI metadata, and rewrite rules for factorizing ML models. For $Q2$, we elaborate our approach in Sec. 4.

Overview. Fig. 5 illustrates the workflow of Ilargi with three inputs: source datasets in matrices, a user-defined ML algorithm (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. First, given the input source datasets and their metadata, Ilargi 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 estimator decides to factorize or materialize based on the cost estimation (Sec. 4). Finally, the ML model is trained in the chosen strategy, i.e., materialization or facotrization.

Running example. Fig. 4 illustrates our running example. 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' height, weight, and medicine (ID) as features. We include patient names for understandability, not included in data matrices in Fig. 5.

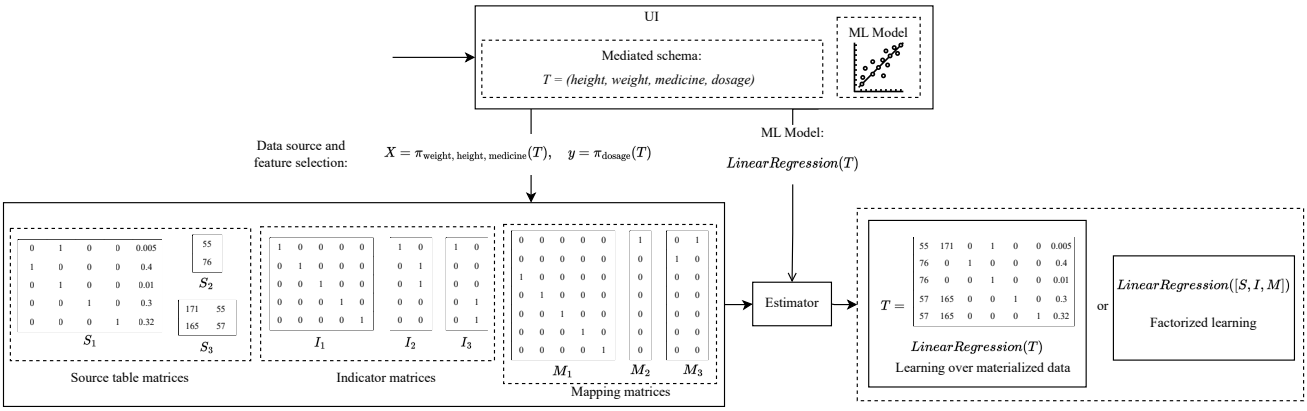

Figure 5: Ilargi workflow and intermediate results of running example.

3.1 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* [16] 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 [16]). 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

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

Implementation. Mapping and indicator matrices are simple, easy-to-implement representations for schema mapping and row matching. A key observation is that they are often sparse. After trying out straightforward implementation following the above definitions, and alternative data structures in Python, our current implementation of mapping and indicator matrices is in sparse matrix format, SciPy Compressed Sparse Row Format (CSR)[1], 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 [25, 21]. 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 our technique report [27].

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.

$$TX \rightarrow I_1 S_1 M_1^T X + ... + I_n S_n M_n^T X$$

We first compute the local result $I_k S_k M_k^T X$ ($k \in [1, n]$) for each source table, which are assembled for the final results.

Implementation-level optimization. To reduce computation overhead, our implementation adopts the optimal matrix multiplication order algorithm [22] from the version included in NumPy[2] for SciPy matrices, computed using matrix dimensions.

Why mapping and indicator matrices? In Sec. 2.4 we mentioned gaps regarding representation and execution. As discussed in Sec. 2.3, at the logical level, with schema mappings we bridge the gaps, i.e., we can merge tables more than just inner joins, and have multiple matched columns, not necessarily following set containment relationships. At computation level, with schema mapping and data matching encoded in matrices, the local data transformation is also computed in linear algebra, e.g., $I_k S_k M_k^T$ in the above example. It facilitates unified, LA-based rewrite rules of factorization.

Answer to Q1

We represent DI metadata, e.g., schema mapping and row matchings as matrices, which brings a unified computation with linear algebra operations for factorization over sources.

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html
[2] https://numpy.org/doc/stable/reference/generated/numpy.linalg.multi_dot.html

Applicability, assumptions, and extensibility. Fig. 3 summarizes Ilargi's applicability compared to SOTA solution [16]. Ilargi can handle both normalized and denormalized tabular data. To prepare the training data, feature and label columns are from source datasets, which means that for each column in the target table, we can find its corresponding column from at least one source schema. Similar to [16], our input source table matrices, e.g., S_1, S_2, S_3 in Fig. 5, are source datasets after preprocessing, e.g., in NumPy arrays. The schema mapping matrices are built between the schemas of source table matrices and the target schema of the training dataset. 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 [46].

4 COST ESTIMATION IN ILARGI

We illustrate the research question $Q2$ intuitively in Fig. 6. We explain Ilargi's estimator depicted in Fig. 17, which performs cost estimation given source datasets, DI metadata, the ML model, and the hardware configuration.

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 $\mathbf{T}$ is a triple $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$, where Σ is a set of dependencies over $\langle \mathbf{S}, \mathbf{T} \rangle$. 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 $\mathbf{T}$ 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.

Observation of source redundancy. Consider the union example in Table 1. Following the table unionability definition in [68, 63, 49], 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 are dropped during preprocessing. Each matched attribute in the common feature set, e.g., m, have the same semantics and domain. With or without row overlap between S_1 and S_2, the size of source instances is large or the same as target instances, i.e., source redundancy. The optimizer will recommend materialization.

Pruning rules. Union is a typical example for area II in Fig. 6. When source tables are unionable, materialization is more efficient [3]. In the logical level, we can describe such relationships as tgds whose left-hand side (LHS) contains only predicates from a single source. 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.

[3]Given the simplicity of unions, our implementation always choose to execute union over materialized data based on the pruning rule.

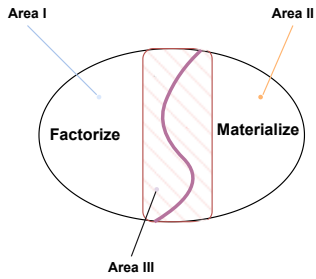

Figure 6: An abstraction of different decision areas

Table 2: Notations used in the paper

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_k/T
c_k/c_T	Number of columns in S_k/T
$m_k/m_T/m_X/m_w$	Number of nonzero elements in $S_k/T/X/w$

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

The bigger picture. Tgds and equality generating dependencies (egds) are the two major types of database dependencies [30]. With tgds and egds we can formalize relationships within a database or across different databases of data sources, e.g., join dependencies, inclusion dependencies, functional dependencies, and schema mappings [59, 1, 34]. We can express functional dependencies in edgs, and develop pruning rules for area I, e.g., identify target redundancies through functional dependencies. We demonstrate such a possibility with an example in [27]. We choose tgds, as they provide a high-level abstraction of dataset relationships, and are a convenient tool to analyze the algorithmic complexity [15, 65] and reasoning tasks [48]. It is a wide-open research field on logic-based formalism for model training and inference over data sources.

Implementation and limitations. The tgd-based pruning rules are implemented in Python, extensible to other query and programming languages, e.g., SQL, Java. Such schema-only rules can be checked quickly, but cannot cover all the cases, as the relative comparison between target redundancy and source redundancy also depends on the row matching, i.e., area III in Fig. 6. We propose a more general method 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.

Table 3: LA operator computations cost comparison

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

Complexity analysis in state-of-the-art. In [16], 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 [16] 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 [16]. In our approach, the matrices are stored in sparse format. Thus, the complexity of matrix multiplication 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 [42]. 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 [42]. 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.

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 [16]. Thus, the discussion at the model level also focuses on such LA operators that bring speedups. We explain the complexity analysis of linear regression, with details of remaining models in the technical report [27].

Linear regression. First, we analyze the complexity of linear regression in Algorithm 2 (Sec. 2.1) in the materialized case, which is dominated by two matrix multiplication operations, i.e., $T^T(Tw)$. For the first matrix multiplication Tw: we denote the shape of weights vector w as $c_T \times 1$. We make the general assumption that w is dense, then $m_w = r_w \times c_w$. We denote the result of Tw as X, which is an intermediate result of linear regression algorithm. The size of X is $r_T \times 1$, i.e., $r_X = r_T$, $c_X = 1$. We assume X is also dense, thus, $m_X = r_X \times c_X$. Now we define the complexity of linear regression in the materialized case based on Tw and $T^T X$.

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

Next, we define the complexity of the factorized case.

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

Complexity ratio. We define a variable *complexity ratio* to indicate whether materialization or factorization leads to more redundancy. The complexity ratio is measured as the ratio of the materialization complexity divided by the factorization complexity.

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

4.4 ML-based Estimator

The third factor. The complexity ratio incorporates two factors, data including DI metadata, and ML algorithms. Our preliminary experiments revealed hardware configurations as a third vital factor. Fig. 7 showcases how the choice varies with changing parallelism

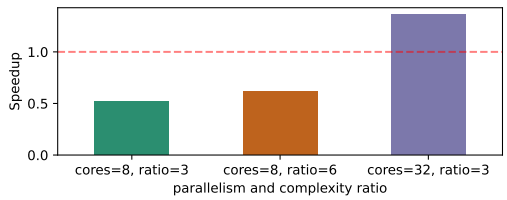

Figure 7: Speedups with different the numbers of CPU cores

w.r.t the numbers of CPU cores. Interestingly, factorization outperforms materialization when the parallelism increases to 32, even at a complexity ratio of 3. This result is attributed to factorization's ability to decouple source table records, thus enabling parallel computational tasks and leveraging extensive hardware parallelism.

Design goal. Our goal is to create an estimator capable of making a binary decision—factorization or materialization—based on data characteristics, computational complexity, and hardware. Our approach, therefore, creates a more efficient machine learning training pipeline over data sources, optimizing the overall efficiency.

Ilargi's tree boosting estimator. Among the various statistical models, tree boosting models are highly regarded in many cost estimators [18, 2] due to their explainability and prediction speed. Moreover, tree boosting has the capability to identify non-linear relationships among features. Given these advantages, we propose a tree-boosting estimator leveraging XGBoost [17]. The main challenge lies in identifying features that are relevant to the decision-making process between materialization and factorization.

Selection of hardware features. Our estimator is generalizable and portable, focusing on macro-level features instead of micro-level. Micro-level features (e.g., CPU cache speeds and memory latency), while valuable, can tie the model to specific hardware, limiting portability. Instead, we incorporate macro-level features like CPU cores and memory bandwidth. Shown in Sec. 5, combined with memory read/write costs, we can effectively estimate maximum memory access costs for both materialization and factorization.

Cost estimation pipeline. Fig. 17 illustrates the workflow of our proposed estimator with the example of linear regression.

1. Feature engineering. The estimator calculates the complexity ratio in Sec.4.3[4], and 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 total amounts, further dividing these normalized features by parallelism and memory bandwidth. We document and explain the used features in our technical report [27].

2. Train XGBoost & predict and recommend strategy. The key to a good performance of our estimator is creating a comprehensive training dataset for XGBoost, which spans a wide array of data characteristics and input models. We elaborate on our empirical approach and its effectiveness in Sec. 5.4.

Limitations and applicability. While the estimator offers benefits such as ease of construction and the ability to capture non-linear features, it requires prior training. This requirement implies the need for a diverse dataset collection. Additionally, the features are sensitive to operator combinations in supported ML algorithms, reducing its predictive accuracy for models with unseen operators. Moreover, the estimator assumes the presence of operator information in the given ML algorithm supporting factorization, which potentially limits the types of models the estimator can support. Once the estimator is trained, it is used for binary classification, i.e., to determine whether to materialize or to factorize.

[4]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.

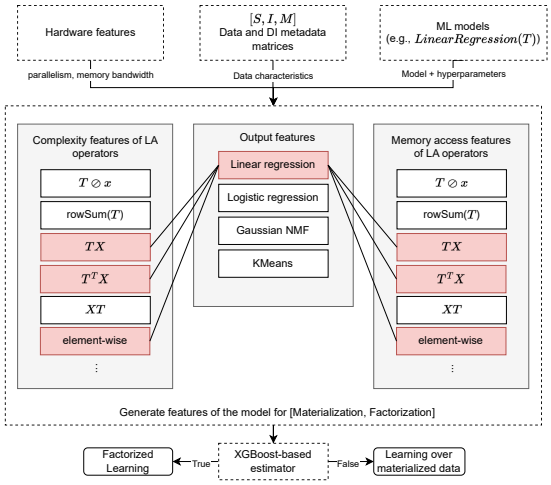

Figure 8: Workflow of the estimator.

Answer to Q2

1. Schema mappings can be used as pruning rules.
2. The choice of factorization and materialization, depends on the datasets and their relationships, ML algorithms and their LA operators, and hardware configuration.

5 EVALUATION

5.1 Datasets and Setup

Synthetic datasets. To assess the accuracy and generalizability of the proposed estimator, we needed to collect runtime samples from a diverse range of integration scenarios, varying in redundancy, sparsity, and complexity. To create such scenarios, we have designed a synthetic data generator. The generator takes as input the dimensions of the source and target tables, target table sparsity, and the type of join. The output is a pair of source tables that, when joined, produce a target table with the specified characteristics. These tables are in matrices and then used to capture the runtime of LA operations and ML models on both materialized and factorized data. Using the parameter ranges shown in Table 4, we created approximately 1800 datasets that allow us to accurately capture relations between data characteristics and operator speedup.

Real datasets. We use the seven real-world datasets from Morpheus and its extensions [53, 72, 16, 57], as provided by Project Hamlet[5] [53, 72]. Each dataset consists of two or more tables connected in a star schema with PK-FK relationships. *nnz* indicates the number of nonzero elements. We include the characteristics of these datasets in the technical report [27].

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

Parameter	Values	Description
r_T	100k, 500k, 1M	Target table rows
c_T	10-50	Target table columns
j_T	left, inner, outer	Jointype
sparsity	0.0 - 0.9	Fraction of 0 values in T
Scalar complexity ratio	0.5 - 9.5	-
S_1 Column ratio	0.1 - 1.0	Fraction of columns from T in S_k
S_2 Column ratio	0.1 - 1.0	

Table 4: Parameters used for synthetic dataset generation.

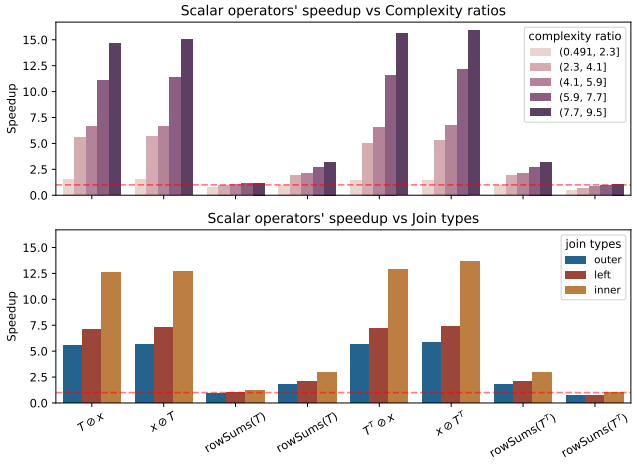

Figure 9: Speedups of factorized scalar operators w.r.t parallelism and complexity ratio. Element-wise operators deliver more significant speedups than reduction operators (e.g., rowSums).

Hardware & Software. We run experiments with 8, 16, and 32 cores of AMD EPYC 7H12 CPU. We use Ubuntu 20.04.4 LTS as the OS, and 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 model training iterations is 20.

5.2 Speedup of Factorized LA Operators

In this section, we examine the speedups achieved by factorized LA operators, in comparison to their materialized counterparts.

Scalar operators. Fig. 9 illustrates the speedups with respect to varying complexity ratios and the join types. Speedups increase as the complexity ratio grows. Among the results, scalar multiplication operators achieve significantly higher speedups compared to rowSums and colSums. Notably, the rowSum and transposed colSums (colSum(T^T)) do not exhibit any speedup when compared to their materialized counterparts. In integration scenarios with different join types, factorization over inner joins delivers the most significant speedups compared to other types.

Fig. 10 presents speedups of factorization over different target table sizes and parallelism. As the size of the target table increases, all operators achieve higher speedups to varying degrees, with scalar multiplications benefiting the most. In contrast, speedups of factorization slightly decreas as the parallelism increases. One might expect that as the number of available cores for parallel computation increases, the speedup achieved through factorization would also increase. However, it is important to note that the decreased

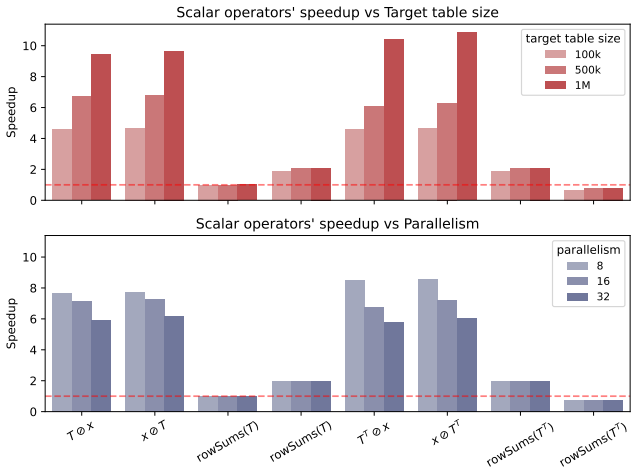

Figure 10: Speedups of factorized scalar operators w.r.t target table size and join type. Factorized scalar operators exhibit higher scalability on large datasets.

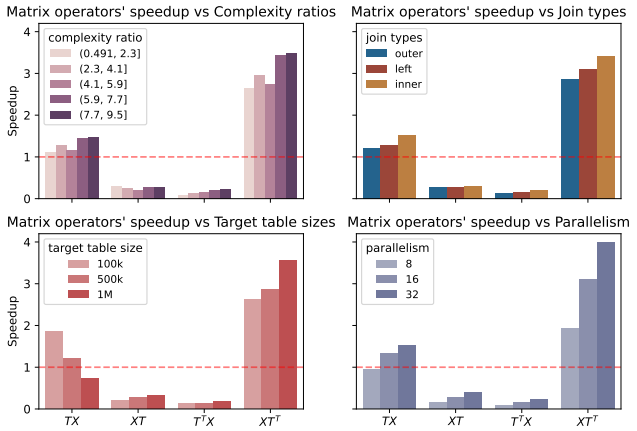

Figure 11: Speedups of factorized matrix operators w.r.t parallelism, complexity ratio, target size, and join type. LMM and transposed RMM show higher speedups with factorization, while RMM and transposed LMM with materialization.

speedup might be due to the fact that increasing parallelism can also benefit materialized execution, thereby reducing the relative speedup of factorized execution.

Matrix operators. Fig. 11 displays the speedups of factorized matrix operators. LMM (TX) and transposed RMM (XT^T) exhibit noticeable speedups in all four configurations. In contrast, RMM (XT) and transposed LMM (T^TX) are slower than materialization. All matrix operators deliver higher speedups with increasing parallelism and complexity ratios. Similar to the evaluation for scalar operators, matrix operators demonstrate better performance when applied to inner joins. However, in the evaluation that varies data sizes, the speedup of LMM declines as the target table size increases, indicating the LMM operator does not scale.

Analysis. The study reveals the impact of data characteristics on the speedup of factorization over materialization at the operator level. For instance, some operators (e.g., scalar multiplications,

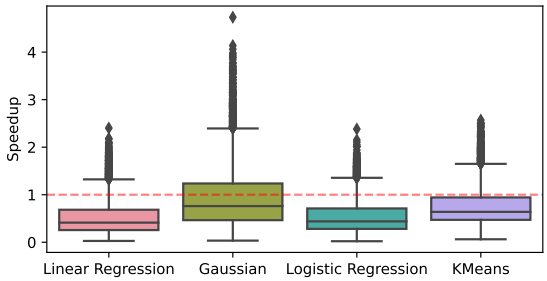

Figure 12: Speedups of factorization across all configurations. Although factorization often presents speedups over materialization, it's not consistently beneficial. It's crucial to understand this variable performance to apply the right strategy for different configurations.

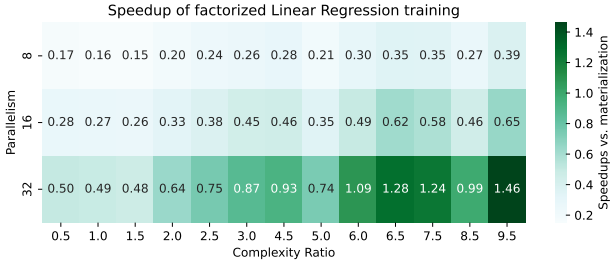

Figure 13: Speedups of factorized linear regression training with varying parallelism and complexity ratio. The decision boundary depends on a combination of complexity and hardware features.

LMM) perform better with factorization, while others (e.g., rowSum, RMM) perform worse due to varying utilization of parallel resources and memory access efficiency.

Element-wise scalar operators over factorization show notable speedups because they can leverage high parallelism: each element's result does not depend on other elements. Interestingly, rowSum and colSum (reduction operators) show different speedups with factorization due to synchronization primitives needed for aggregations. In addition, given our implementation's row-major memory layout, colSum's row-major access yields more efficient memory I/O than rowSum. Idem for LMM and transposed RMM.

Takeaways:
The speedups of LA operators using factorization fluctuate considerably depending on data characteristics, parallelism, and memory I/O efficiency.

5.3 Speedups of factorized ML-model training

The speedups of individual LA operators exhibit significant variation across different data and parallelism settings. Some scalar operators can achieve a speedup of over 10x, while certain matrix operators like RMM may underperform compared to their materialized counterparts. Consequently, ML model training, composed of combinations of these operators, may exhibit different speedups.

As shown in Fig. 12, speedups on various data configurations vary significantly, ranging from 0x to 5x, highlighting the need for an accurate estimator. Fig. 13 demonstrates that speedups for linear regression (other ML algorithms exhibited similar behavior) increase with parallelism and complexity ratio, supporting our theory in Sec. 4.3. That is, these factors are key for deciding between factorization and materialization.

Analysis. The speedup variability is attributed to the *mix* of operators in a model, with scalar operators and matrix multiplications having a significant impact. For instance, the linear regression model, rich in highly accelerated scalar multiplications, shows a higher average speedup than GaussianNMF, which includes slower RMM operators.

Takeaways: *The significant variability in speedup (ranging from 0.2x to 5x) depends on data and parallelism configurations, which calls for an accurate estimator. Thus, the estimator should consider key features such as data characteristics, operator complexities, and hardware.*

5.4 Effectiveness of ML-based Estimator

We elaborate on training configurations of our tree boosting based estimator in Sec. 4.4. We compare our solution with baselines over synthetic data and validate our estimator using real-world data.

5.4.1 *Configuration.*

Training and test data. Based on the ML algorithm speedup evaluation, we extracted features and generated training data with labels (True for speedup when factorizing and False for the opposite). We further categorize the labels into 4 categories: True-Positive, True-Negative, False-Positive, and False-Negative. For instance, True-Positive prediction means that the estimator predicts factorization is faster correctly, while a False-Positive is the opposite. The data was split into training (80%) and test (20%) sets.

Evaluation. We now evaluate the accuracy and F1-score of our estimator model and compare it with other estimators using a synthetic test set. Accuracy reflects the model's ability to accurately decide between factorization and materialization.

Besides quality metrics, we assess estimators' effectiveness by comparing their overall speedup over materialization on the test set, focusing on algorithms trained with parallelism of 32. The observed speedups with the estimator approach, achieving a maximum speedup of 1.24x (as observed in our experiments), underscore the advantages of our ML-based estimator over threshold-based estimators in practical ML workloads, demonstrating its potential to enhance the overall performance of ML tasks.

Five baselines for comparison. We compared the performance of our proposed estimator in Sec. 4.4 with the following five baselines.
i) To confirm the significance of hardware information, we train another tree boosting model using the same dataset, but *without hardware features.*
ii) A second baseline is applying *logistic regression* instead of tree boosting. As a linear model, it may not effectively capture complex feature interactions like the tree-based model.
iii) The third baseline is using only the *complexity ratio* in Sec. 4.3. A complexity ratio of 1 marks the complexity borderline between materialization and factorization.
iv) The *tuple ratio and feature ratio*, metrics used in the state-of-the-art [16], quantify the target table's redundancy relative to the joinable tables. Morpheus [16] suggests a threshold of 5 for tuple ratio, and 1 for feature ratio.
v) Finally, we consider the selectivity of joins, a traditional metric that measures the data proportion filtered out during join operations. Selectivity larger than 1 suggests that the result table has

Table 5: Accuracy, F1-score and Overall Speedups of estimators. Our tree boosting estimator shows superior predictive quality and effectiveness over synthetic test data

	Accuracy	F1-score	Overall Speedups
Tree Boosting (ours)	**0.971**	**0.936**	**1.24**
Tree Boosting w/o hardware	0.712	0.560	1.19
Logistic Regression	0.776	0.620	1.20
Complexity Ratio	0.447	0.420	1.16
tuple ratio + feature ratio	0.490	0.426	1.16
Selectivity	0.244	0.362	1.08

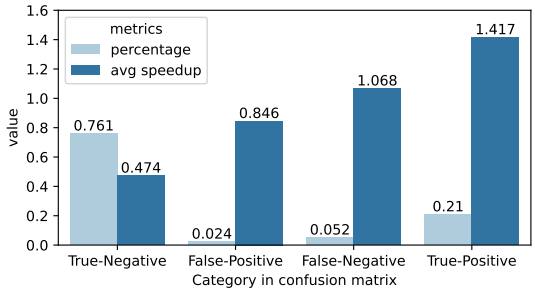

Figure 14: Confusion matrix of prediction result in test set. Over 20% instances gain speedup using factorized learning.

more data than the number of tuples of the largest source table, implying potential benefit using factorization.

5.4.2 Evaluation on synthetic data. Table 5 compares accuracy and F1-score of different estimators. The tree boosting model incorporating hardware features demonstrates 96.9% accuracy. Its high F1-score further emphasizes its balance between precision and recall, minimizing false predictions. In addition, the overall speedup with our estimator achieves 1.24, surpassing the baselines.

Performance of baselines. The tree boosting estimator without hardware features still achieves relatively high accuracy and overall speedups but presents a significant F1-score drop, verifying our hypothesis on the importance of hardware information in deciding between factorization or materialization. Finally, threshold-based estimators fall short in both accuracy and F1-score. The complexity ratio outperforms the tuple ratio suggested in earlier research across various threshold configurations. These results highlight our tree boosting model's effectiveness in determining whether to employ factorization or materialization.

Other metrics. Fig. 14 provides a confusion matrix for predictions from our tree boosting estimator, showing over 20% of test instances can benefit from factorized learning. We also analyze the average speedup for each confusion matrix category. Correctly predicted instances, on average, achieve a 1.42x speedup with factorization. Furthermore, when comparing the average speedups of False-Positive and True-Negative predictions, we find an interesting trend. Even when factorization is incorrectly applied due to mispredictions, the model training process can still achieve around 85% of the expected performance. This implies that our estimator is highly robust, maintaining reasonable performance even in the face of errors. Comparing the average speedups of True-Negative instances, our estimator shows a high positive correlation between accurate predictions and speedups.

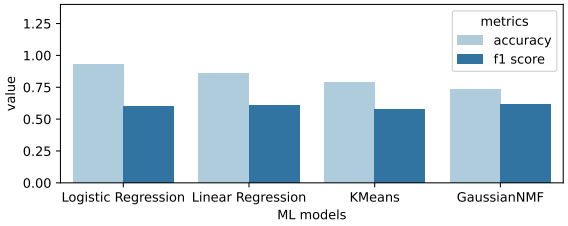

Figure 15: The predictive quality of our estimator under a "leave-one-out" scenario. Test set only contains the left-out model. For instance, the estimator for Logistic Regression has not been trained with any datapoints that we have gathered for logistic regression.

Table 6: Top 10 information gain of features in the estimator.

Top 10 features	Info. Gain
rowSums mem. write	30.96
LMM complexity with factorization	22.69
Dense scalar operation complexity	20.53
LMM complexity with materialization	8.68
colSums complexity with materialization	6.30
Complexity ratio	6.21
Total mem. write with materialization	5.88
Dense MM operation complexity	5.63
Dense MM mem. write	5.77
Feature Ratio	5.45

Ablation study for generalizability. To evaluate the generalizability of our performance estimator, we utilized a leave-one-out cross-validation method, specific to different ML models to be trained. We systematically exclude each ML model from the training set, one at a time. The excluded model is then used as the test set to evaluate the predictive quality of performance estimator trained without any information from that specific model. This way, we assess how well our estimator can generalize its predictions to unseen data related to the excluded model.

Fig. 15 indicates a decrease in both accuracy and F1-score when certain models are excluded from training data. This is due to missing operator features in the training set. For example, excluding the KMeans model, which heavily uses the dense scalar operator, creates a predictive bias due to an unbalanced feature distribution. Despite these constraints, the estimator remains functional with an approximate 80% accuracy rate. This suggests that even with certain predictive biases, our estimator is resilient and generalizable in optimizing ML workloads with new ML models, offering reasonably accurate predictions across different scenarios.

5.4.3 Feature Importance. Training the boosting tree model allows us to extract node information gain for ranking feature importance. Table 6 shows the top-10 features by information gain. Complexity features related to LMM, one of the most common operators in model training, have significantly high information gain. Operators like rowSums and colSums, which are not easily parallelized, are also important to determining speedup. Apart from complexity features, we observed a notable influence of memory I/O features, particularly those related to dense operators. Even though our raw data is stored in a sparse format, computations frequently generate dense intermediate results. This can significantly impact overall performance due to high I/O costs.

Table 7: Accuracy, F1-score and Overall Speedup of estimators over real-world data. Our proposed estimator maintains a performance similar to the evaluation conducted with the test set, while other estimators demonstrate varying degrees of performance degradation.

	Accuracy	F1-score	Overall Speedups
Tree Boosting (ours)	**0.905**	**0.821**	**1.22**
Logistic Regression	0.651	0.45	1.11
Complexity Ratio	0.409	0.387	1.09
Tuple Ratio	0.464	0.211	0.94
Selectivity	0.591	0.237	0.97

The complexity ratio, although not the most critical feature, is important in the estimator along with memory I/O features. Both materialization and factorization complexities are intrinsically linked to the complexity ratio. Conversely, the tuple ratio ranks lower, implying a lesser impact on performance estimation.

Our feature importance analysis offers valuable insights for designing a performance estimator for factorized learning, emphasizing the importance of considering hardware features. Incorporating these in the estimation process can improve prediction accuracy, given the computational demands of factorized learning and the influence of hardware features on its efficiency.

5.4.4 Validation on real-world data. We further evaluated our estimator using real-world data to validate its practical usability in realistic ML workloads. The predictive quality of our estimator is presented in Table 7. For a fair comparison, we use the same datasets employed in Morpheus [16].

Despite certain differences in data characteristics (e.g., target table sizes, sparsity) between the real-world and synthetic data, our estimator continues to demonstrate remarkable predictive quality in comparison to other estimators. Even with a slight drop in predictive quality, our estimator still manages to achieve an overall speedup of 1.24 when applied to model training tasks using real-world data sets. This performance is close to the theoretical peak value of 1.27. This outcome indicates that dataset variations do not significantly impact the effectiveness of our estimator.

In addition, we investigate the accuracy for each dataset in Fig. 16. The accuracy values do not show large variations across different datasets. Through evaluations conducted on real-world datasets, our estimator has demonstrated consistent effectiveness, underscoring its robustness and reliability across a range of data characteristics. In summary, by leveraging our performance estimator to make informed decisions on whether to employ factorization or materialization, we can enhance the overall speedup of ML training workloads across data sources.

Takeaways:
i) Our tree-boosting performance estimator outperforms other estimators in terms of accuracy, F1-score, and overall speedup on both synthetic and real-world datasets, thereby enhancing ML training workloads over source tables.
ii) Our estimator exhibits robust predictive power on unseen ML algorithms and datasets.

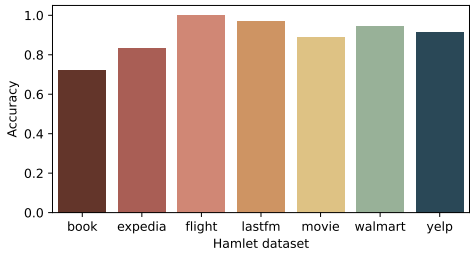

Figure 16: Accuracy w.r.t different real-world datasets.

6 RELATED WORK

Factorized learning. Early efforts required *manual* design for factorizing specific ML algorithms. Orion [51] supports linear and logistic regression, which are later extended to decision trees, feature ranking, and Naive Bayes in Santoku [52]. These two works mainly tackle inner joins via PK-FK relationship, while F [71] has extended factorization of linear regression to natural joins. As the successors of F, AC/DC [47] adds support to categorical variables, and LMFAO supports more ML algorithms such as decision trees. In a more *automated* manner, Morpheus [16] proposed a general factorization framework enabling the automation of linear algebra rewriting. By reordering of multiplication and exploiting pre-computed results, HADAD [3] creates rewrites for systems like Morpheus and further speeds up the computation. Later, Trinity [45] extends Morpheus in such a way that factorized LA logic is written *once*, but can be reused in multiple programming languages and LA tools. MorpheusFI [57] added support to non-linear feature interactions. Non-linear models such as Gaussian Mixture Models and Neural Networks are supported in [19].

Ilargi's main contribution is not on extending ML algorithms. We expand the dataset relationships to various integration scenarios, cf. Table 1. We take a more critical inspection of the speedup of factorization compared to materialization by utilizing DI metadata. We also discovered a third factor, hardware configuration, which has an impact on the decision boundary. We are currently expanding the implementation of Ilargi to more ML models, e.g., decision trees.

Tensor-based ML and data management systems. A large body of work has studied the unification of linear and relational algebra operations at the level of the execution engine [60, 10]. SystemDS [11] models data as tensors, and performs optimizations for end-to-end ML pipelines. TQP [41] executes relational operators such as joins and projections in tensor algebra, while multiple works leverage GPUs for relational joins [4, 23, 43]. Hummingbird [62, 50] compiles traditional ML models into tensor-based runtimes designed for deep learning models. Ilargi is complementary to these approaches as its input source datasets also come in matrix form. The key novelty of Ilargi lays in the modeling of data integration metadata (e.g., schema mappings traditionally expressed in first-order logic) in a matrix representation.

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, enabling factorization over sources. Moreover, Ilargi utilizes schema mappings as pruning rules, and employs a tree boosting based estimator, which leverages complexity ratios and hardware configuration to choose between materialization and factorization. Our experiments demonstrate the effectiveness and efficiency of our estimator, which covers more scenarios than state-of-the-art solutions and performs correct predictions with 70-96% accuracy depending on the dataset.

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.

$$T \rightarrow I_0 S_0 M_0^T + ... + I_k S_k M_k^T$$

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:

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

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 [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: T, 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: T, 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: T, 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.

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

E 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 E.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:

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

$f_2 : S_2.x_2 \rightarrow S_2.x_4 \ \Rightarrow\ S_2(b, d_1) \wedge S_2(b, d_2) \rightarrow d_1 = d_2$

Target egds over target table T:

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

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

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

To explain Example E.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.

F OPTIMIZATIONS OF MATRIX MULTIPLICATION ORDER

As mentioned in Sec.3.2, 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:

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

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.

G 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,

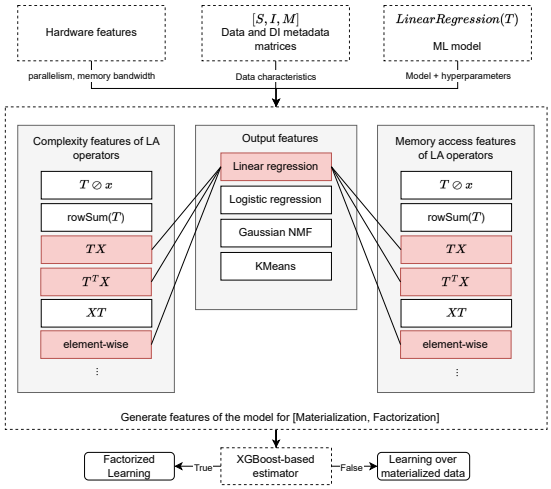

Figure 17: Workflow of the estimator.

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). The full list of features can be found in Sec. H.

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.

[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.

[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.

H FEATURE LIST

Table 8: The table displays the features of our tree-boosting estimator, which are divided by total complexity or total memory access depending on the feature category in both materialization and factorization scenarios. The features in complexity category are further divided by parallelism.

Category	Feature	Normalization
Memory	Materialized (or factorized) operators' memory read in bytes	yes
	Materialized (or factorized) operators' memory write in bytes	
	Memory read in bytes of scalar operators over dense matrices	
	Memory write in bytes of scalar operators over dense matrices	
	Memory read in bytes of matrix operators over dense matrices	
	Memory write in bytes of matrix operators over dense matrices	
Complexity	scalar operators' complexity with materialization or factorization	yes
	LMM operators' complexity with materialization or factorization	
	RMM operators' complexity with materialization or factorization	
	Scalar operators' complexity over dense matrices	
	Matrix operators' complexity over dense matrices	
Mixed	Complexity of materialized operators with column-major memory access	yes
	Memory read in bytes of the colSum operator with materialization or factorization	
	Memory write in bytes of the colSum operator with materialization or factorization	
	Memory read in bytes of the rowSum operator with materialization or factorization	
	Memory write in bytes of the rowSum operator with materialization or factorization	
	Complexity of the rowSum operator with materialization or factorization	
	Complexity of the cowSum operator with materialization or factorization	
Others	Selectivity	no
	Complexity ratio	
	Tuple ratio	
	Feature ratio	

I 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:

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

J DATA CHARACTERISTICS

As shown in Table 9 we use the same datasets from the state-of-the-art system [53, 72, 16, 57]. These datasets developed for Project Hamlet[9] [53, 72] 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.

Table 9: Data characteristics of real datasets.

Dataset	r_T	c_T	$nnz\ T$	k	$nnz\ S$	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
					6464	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

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

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[2] Andrew Adams et al. "Learning to optimize halide with tree search and random programs". In: *ACM Trans. Graph.* 38.4 (2019), 121:1–121:12.

[3] Rana Alotaibi et al. "HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21: International Conference on Management of Data. Virtual Event China: ACM, June 9, 2021, pp. 23–35.

[4] Rasmus Resen Amossen and Rasmus Pagh. "Faster Join-Projects and Sparse Matrix Multiplications". In: *ICDT 2009*. St. Petersburg, Russia: Association for Computing Machinery, 2009, 121–126.

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

[6] David Aumueller et al. "Schema and ontology matching with COMA++". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005, pp. 906–908.

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

[8] Luigi Bellomarini et al. "Exploiting the Power of Equality-Generating Dependencies in Ontological Reasoning". In: *Proc. VLDB Endow.* 15.13 (2022), 3976–3988.

[9] Felix Biessmann et al. "DataWig: Missing Value Imputation for Tables." In: *J. Mach. Learn. Res.* 20.175 (2019), pp. 1–6.

[10] Matthias Boehm, Matteo Interlandi, and Chris Jermaine. "Optimizing Tensor Computations: From Applications to Compilation and Runtime Techniques". In: *Companion of the 2023 International Conference on Management of Data*. SIGMOD '23. Seattle, WA, USA: Association for Computing Machinery, 2023, 53–59.

[11] Matthias Boehm et al. "SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle". In: *CIDR*. 2020.

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

[13] 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.

[14] Marco Calautti et al. "Exploiting Equality Generating Dependencies in Checking Chase Termination". In: *Proc. VLDB Endow.* 9.5 (2016), 396–407.

[15] Balder ten Cate and Phokion G. Kolaitis. "Structural characterizations of schema-mapping languages". In: *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*. Ed. by Ronald Fagin. Vol. 361. ACM International Conference Proceeding Series. ACM, 2009, pp. 63–72.

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

[17] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram et al. ACM, 2016, pp. 785–794.

[18] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 578–594.

[19] Zhaoyue Cheng et al. "Efficient Construction of Nonlinear Models over Normalized Data". In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1140–1151.

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

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

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

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

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

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

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

[27] *Efficient ML Model Training over Silos: to Factorize or to Materialize*. Technical report. https://raw.githubusercontent.com/amademicnoboday12/sigmod24-310/main/paper/technical_report.pdf. 2023.

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

[29] Jérôme Euzenat, Pavel Shvaiko, et al. *Ontology matching*. Vol. 18. Springer, 2007.

[30] 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.

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

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

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

[34] Ronald Fagin et al. "Data exchange: Semantics and query answering". In: *Database Theory—ICDT 2003: 9th International Conference Siena, Italy, January 8–10, 2003 Proceedings 9*. Springer. 2003, pp. 207–224.

[35] Grace Fan et al. "Table Discovery in Data Lakes: State-of-the-Art and Future Directions". In: *Companion of the 2023 International Conference on Management of Data*. SIGMOD '23. Seattle, WA, USA: Association for Computing Machinery, 2023, 69–75.

[36] 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.

[37] Isabelle Guyon et al., eds. *Feature Extraction - Foundations and Applications*. Vol. 207. Studies in Fuzziness and Soft Computing. Springer, 2006.

[38] Rihan Hai et al. "Data Lakes: A Survey of Functions and Systems". In: *IEEE Transactions on Knowledge and Data Engineering* (2023), pp. 1–20.

[39] Alon Halevy, Anand Rajaraman, and Joann Ordille. "Data Integration: The Teenage Years". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, 9–16.

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

[41] Dong He et al. "Query Processing on Tensor Computation Runtimes". In: vol. 15. 11. VLDB Endowment, 2022, 2811–2825.

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

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

[44] *Informatica*. https://www.informatica.com/es/products/data-integration/powercenter.html.

[45] 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.

[46] 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.

[47] Mahmoud Abo Khamis et al. "AC/DC: In-Database Learning Thunderstruck". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, June 15, 2018, pp. 1–10.

[48] Phokion G. Kolaitis et al. "Nested dependencies: structure and reasoning". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Richard Hull and Martin Grohe. ACM, 2014, pp. 176–187.

[49] 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.

[50] Dimitrios Koutsoukos et al. "Tensors: An Abstraction for General Data Processing". In: *Proc. VLDB Endow.* 14.10 (2021), 1797–1804.

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

[52] 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.

[53] 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.

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

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

[56] Peng Li et al. "CleanML: A Benchmark for Joint Data Cleaning and Machine Learning [Experiments and Analysis]". In: *CoRR* abs/1904.09483 (2019). arXiv: 1904.09483.

[57] 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.

[58] Jiabin Liu et al. "Feature augmentation with reinforcement learning". In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE. 2022, pp. 3360–3372.

[59] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[60] Nantia Makrynioti and Vasilis Vassalos. "Declarative Data Analytics: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 33.6 (June 2021), pp. 2392–2411.
[61] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.
[62] Supun Nakandala et al. "A Tensor Compiler for Unified Machine Learning Prediction Serving". In: *OSDI 2020*. USA: USENIX Association, 2020.
[63] Fatemeh Nargesian et al. "Table union search on open data". In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 813–825.
[64] *Pentaho*. https://www.hitachivantara.com/en-us/products/pentaho-platform/data-integration-analytics.html.
[65] Reinhard Pichler and Sebastian Skritek. "The Complexity of Evaluating Tuple Generating Dependencies". In: *Proceedings of the 14th International Conference on Database Theory*. ICDT '11. Uppsala, Sweden: Association for Computing Machinery, 2011, 244–255.
[66] Neoklis Polyzotis et al. "Data Lifecycle Challenges in Production Machine Learning: A Survey". In: *SIGMOD Rec.* 47.2 (2018), pp. 17–28.
[67] Erhard Rahm and Philip A Bernstein. "A survey of approaches to automatic schema matching". In: *the VLDB Journal* 10.4 (2001), pp. 334–350.
[68] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*. Vol. 3. McGraw-Hill New York, 2003.
[69] Theodoros Rekatsinas et al. "HoloClean: Holistic Data Repairs with Probabilistic Inference". In: *Proceedings of the VLDB Endowment* 10.11 (2017).
[70] Yuji Roh, Geon Heo, and Steven Euijong Whang. "A Survey on Data Collection for Machine Learning: A Big Data - AI Integration Perspective". In: *IEEE Transactions on Knowledge and Data Engineering* 33.4 (2021), pp. 1328–1347.
[71] 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. New York, NY, USA: Association for Computing Machinery, June 14, 2016, pp. 3–18.
[72] Vraj Shah, Arun Kumar, and Xiaojin Zhu. "Are key-foreign key joins safe to avoid when learning high-capacity classifiers?" In: *Proceedings of the VLDB Endowment* 11.3 (2017), pp. 366–379.
[73] *SQL Server Integration Services (SSIS)*. https://learn.microsoft.com/en-us/sql/integration-services/sql-server-integration-services?view=sql-server-ver16.
[74] Michael Stonebraker and Ihab F. Ilyas. "Data Integration: The Current Status and the Way Forward". In: *IEEE Data Eng. Bull.* 41.2 (2018), pp. 3–9.
[75] *Talend*. https://www.talend.com/products/integrate-data/.
[76] Lei Yu and Huan Liu. "Efficient feature selection via analysis of relevance and redundancy". In: *The Journal of Machine Learning Research* 5 (2004), pp. 1205–1224.