Bridging the Gap between Data Integration and ML Systems

Rihan Hai†  Yan Kang∗  Christos Koutras†  Andra Ionescu†  Asterios Katsifodimos†
†Delft University of Technology, ∗WeBank
{initial.lastname}@tudelft.nl, yankang@webank.com

ABSTRACT
The data needed for machine learning (ML) model training and inference, can reside in different separate sites often termed data silos. For data-intensive ML applications, data silos present a major challenge: the integration and transformation of data, demand a lot of manual work and computational resources. Sometimes, data cannot leave the local store, and the model has to be trained in a decentralized manner. In this work, we propose three matrix-based dataset relationship representations, which bridge the classical data integration (DI) techniques with the requirements of modern machine learning. We discuss how those matrices pave the path for utilizing DI formalisms and techniques for our vision of ML optimization and automation over data silos.

1 INTRODUCTION
Machine learning has become the backbone of information technology and is widely used in various domains such as retail, medical diagnosis, and transportation. The performance of an ML model heavily depends on its training data. In real world applications, often the data is not stored in a central database or file system, but spreads over different data silos. For example, for drug risk prediction, the features reside in datasets collected from clinics, hospitals, pharmacies, and laboratories [5]; to train a model for keyboard prediction, data from millions of phones needs to be accessed [21].

The data management systems that handle data silos, are data integration systems. A data integration system empowers the interoperability among multiple, heterogeneous sources, and provides a unified view for users. One of the key ingredients of data integration is to describe the sources and their relationships [11]. Such information is two-fold: mappings between different source schemata, i.e., schema matching and mapping [14, 42]; linkages between data instances, i.e., data matching (also known as record linkage or entity resolution) [6]. However, the end goal of a data integration system is to answer queries or transform data over silos. It does not directly support machine learning applications. Thus, today people solve the silos problem with the DI systems and ML tools separately, as shown in the following example.

Running example. Consider the feature augmentation example in Fig. 1. A data scientist tries to build a classification model to predict the Titanic passenger survival rate. She starts from a base table $S_1(s,n,se,a)$, which has the label column $s$ (survived), and feature columns $se$ (sex) and $a$ (age). To improve model accuracy, she discovered a second source table $S_2(s,n,se,p,d)$ in Fig. 1b. It brings a new feature column $p$ (parc), which is the number of companion parents/children traveling with the specific passenger. The label column and the selected feature columns constitute the schema of the table for downstream ML models, i.e. $T(s,se,a,p)$, which we refer to as the target table schema or meditated schema.

Figure 1: Integrate data in silos for ML: the traditional way

Problems with the separation of DI and ML. As shown in Fig. 1c-d, to use the data from the two tables $S_1$ and $S_2$, the data scientist needs to rely on a data integration system, or manually find the schema mapping and data matching between the two given tables. We elaborate on the explanation of schema mappings in Sec. 2.1. Then a data integration system can integrate these source tables by merging the mapped columns and linked entities (i.e., matched rows). Finally, it materializes the data instances of the target table $T$ and exports it to downstream ML applications. Such a process usually involves massive manual work and computational overhead, e.g., joining tables. Meanwhile, it demands the user equipped with the knowledge of data sources, principles of data integration, and/or familiarity with DI tools, e.g., Talend1, Informatica2. In other words, there is great potential to utilize DI techniques to reduce the human burden and automate ML pipelines.

1https://www.talend.com/products/integrate-data/
2https://www.informatica.com/products/data-integration.html
When we focus on ML applications with the data spread in silos, the difference, however, is that DI systems are designed to answer queries or transform data (e.g., Fig. 1d), not for model training. Thus, in this work, we first investigate data source reconciliation or even rebuild the logic-based DI theoretical framework. As the foundation of these systems, we might also need to federated learning [49], meta-learning [45], reinforcement learning [44]. As the foundation of these systems, we might also need to reconcile or even rebuild the logic-based DI theoretical framework. As the foundation of these systems, we might also need to reconcile or even rebuild the logic-based DI theoretical framework.

The contributions of this paper go as follows:

- **Formalized DI scenarios for ML.** Given ML applications, in particular, feature augmentation and federated learning, we categorize and formalize their typical dataset relationships with first-order schema mapping languages, cf. Table 1 (Sec. 2).

- **Matrix-based representations of dataset relationships.** To seamlessly support ML, we define three types of matrices, which capture column matching, row matching, and redundancies between data sources and the target table (Sec. 3).

- **New opportunities to automate and optimize ML.** With our matrix-based representations, we enlighten the new opportunities for linear algebra rewriting in model factorization, and feature engineering and model training in federated learning (Sec. 4).

- **System.** We demonstrate how our vision enables a next-generation data science platform Amalur for supporting an end-to-end, scalable machine learning pipeline over data silos (Sec. 5).

### Table 1: Four example data integration scenarios for feature augmentation and federated learning

| No. | Dataset Relationship | Schema mappings | Example use cases |
|-----|----------------------|-----------------|------------------|
| 1   | Full outer join      | $m_1 : \forall s, n, se, a, p, d \ (S_1(s, n, se, a) \land S_2(s, n, se, p, d) \rightarrow T(s, se, a, p))$ | Feature augmentation, Federated learning, … |
|     |                      | $m_2 : \forall s, n, se, a \ (S_1(s, n, se, a) \rightarrow \exists p \ T(s, se, a, p))$ | Feature augmentation, (Vertical) federated learning, … |
| 2   | Inner join           | $m_1 : \forall s, n, se, a, p, d \ (S_1(s, n, se, a) \land S_2(s, n, se, p, d) \rightarrow T(s, se, a, p))$ | Feature augmentation, (Vertical) federated learning, … |
|     |                      | $m_2 : \forall s, n, se, a \ (S_1(s, n, se, a) \rightarrow \exists p \ T(s, se, a, p))$ | Feature augmentation, (Vertical) federated learning, … |
| 3   | Left join            | $m_2 : \forall s, n, se, a, p, d \ (S_2(s, n, se, a, p, d) \rightarrow T(s, se, a, p))$ | Data sample augmentation, (Horizontal) federated learning, … |
| 4   | Union                | $m_3 : \forall s, n, se, a, p, d \ (S_2(s, n, se, a, p, d) \rightarrow T(s, se, a, p))$ | Data sample augmentation, (Horizontal) federated learning, … |

### 2 PROBLEM DEFINITION

We first introduce the mapping formalism to describe data integration scenarios in Sec. 2.1. In this work, we focus on two ML use cases, feature augmentation and federated learning. In Sec. 2.2 we explain how we capture the dataset relationships in these use cases with the mapping formalism. We analyze our goals in Sec. 2.3. In this work, by “dataset” we restrict ourselves to relational tables.

### 2.1 Preliminaries

**Schema mappings** lay at the heart of data integration and data exchange. Let $S$ and $T$ be a source relational schema and a target relational schema sharing no relation symbols. A schema mapping $M$ between $S$ and $T$ is a triple $M = (S, T, \Sigma)$, where $\Sigma$ is a set of dependencies over $(S, T)$. The dependencies $\Sigma$ can be expressed as logical formulas over source and target schemas. One of the most commonly used mapping languages is source-to-target tuple generating dependencies (s-t tgd) [3], which are also known as Global-Local-as-View (GLAV) assertions [35]. An s-t tgd is a first-order sentence in the form of $\forall x \ (\varphi(x) \rightarrow \exists y \ \psi(x, y))$, where $\varphi(x)$ is a conjunction of atomic formulas over the source schema $S$, and $\psi(x, y)$ is a conjunction of atoms over the target schema $T$.

**Example 2.1.** In Fig. 1c, $m_1$, $m_2$, and $m_3$ are all tgds. We represent mapped attributes with the same variable names, e.g., $S_1.s$ and $S_2.s$. The tgd $m_1$ specifies that the overlapped rows of $S_1$ and $S_2$ are added to $T$ ($\land$ 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 $S_2$.

**Federated learning (FL)** [49] studies how to build joint ML models over data silos (e.g., enterprise data warehouses, edge devices) without compromising privacy, which follows a decentralized learning paradigm. Similar to the problem setting of virtual data integration [11], FL assumes that the source data is not collected and stored at a central data store but stays at the local data stores. Notably, in this work we mainly study the dataset relationship in FL. There are other important yet orthogonal problems such as privacy-preserving, which we leave to future works. According to how the feature space and sample space are partitioned among the data sources, FL can be mainly categorized as Vertical FL (VFL) and Horizontal FL (HFL) [48]. For VFL, data sources share the overlapping data instances but the feature columns are partially overlapped or disjoint. For HFL, data sources share the overlapping feature columns, while the data instances may or may not overlap.
Feature augmentation is the exploratory process of adding new datasets and selecting features that help improve the ML model performance [9, 13, 31]. Fig. 1b shows an example: starting from a base table $S_1$, we augment the features by introducing the table $S_2$ and selecting the new feature parch.

2.2 DI scenarios for ML

In general, the aforementioned mapping formalism can describe ML applications whose features and samples are decentralized. We showcase two representative use cases, feature augmentation and federated learning. Table 1 shows four examples of DI scenarios, which are specified by tgd. We use them to explain typical dataset relationships of feature augmentation and federated learning.

Case 1 (full outer join) is explained for feature augmentation in Fig. 1 and Example 2.1. It can also be seen as a general case of federated learning, where sources have similar schemas, and data instances (entities) that may, or may not overlap with each other.

Case 2 (inner join) represents the DI scenario where only overlapped rows in two sources will be transformed, i.e., an inner join between $S_1$ and $S_2$ then a projection on columns $s$, $e$, $a$, $p$. It can be used to describe the feature augmentation processes where fewer missing values are preferred. Such dataset relationships also reside in a VFL use case, where data sources share the sample space (overlapped rows) but not necessarily the feature space (overlapped feature columns).

Case 3 (left join) shows a left join between $S_1$ and $S_2$. Compared to Case 1, we slightly change the schema of $S_2$ by dropping the label column $s$. Case 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. Case 3 can be used to describe the VFL cases where not all but specific sources hold the labels for supervised model training.

Case 4 (union) is a special case of Case 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 $T$. Case 4 can represent the scenario when a new table is selected to bring more data samples. Alternatively, it can describe the HFL scenario where data sources share feature columns but not data samples.

2.3 Problem analysis

Now we revisit the question $Q$ raised in Sec. 1. Table 1 partially answers $Q$. That is, for the use cases of feature augmentation and federated learning (and possibly many more), the dataset relationships between source tables and desired target table, can be captured by a class of well-studied data dependencies, i.e., tgd, which are the commonly used formalism in data integration studies.

However, there is still a gap. Tgd is first-order sentences. A DI system generates such schema mappings in executable data transformation scripts, e.g., SQL, which transform the source data instances and materialize the target table $T$. In contrast, one of the fundamental mathematical disciplines of machine learning is linear algebra, i.e., computations among scalars, vectors, matrices, and tensors. Such a fundamental difference in their natures, hinders us from effectively using DI techniques for data science and machine learning tasks. In the next section, we introduce three simple representations, which weld the DI and ML worlds.

### 3 A TALE OF THREE MATRICES

In this section, we define three dataset relationship representations: mapping matrix for preserving the column mapping (Sec. 3.1), indicator matrix for row matching (Sec. 3.2), and redundancy matrix for data redundancy (Sec. 3.3). In Table 2 we summarize the notations used in this paper. We abuse the notation a little and refer to both a target/source table name and its schema with $T/S_k$.

#### 3.1 Mapping matrix

Schema mappings contain the information about the mapped columns between source and target tables. We define the mapping matrix to preserve such column mappings. As a preparation step, we add ID numbers to mapped columns as shown in Fig. 2a.

**Definition 3.1 (Mapping matrix).** Mapping matrices between source tables $S_1, S_2, ..., S_n$ and target table $T$ are a set of binary matrices $M = \{M_1, ..., M_n\}, M_k (k \in [1, n])$ is a matrix with the shape $c_T \times c_{S_k}$, where

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

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. 2a shows the mapping matrices $M_1$ for $S_1$, and $M_2$ for $S_2$ of the running example.

It is easy to see that the binary mapping matrices are often sparse. Because each attribute in the source table $S_k$ is mapped to only one attribute in $T$. Thus, in each row of $M_k$ at most one element is 1, while the rest are 0. Moreover, if an attribute of $T$ does not have a mapped attribute in $S_k$, the corresponding row of the mapping matrix has only values of 0. For example, $T.p$ (column ID: 3) does not have a mapped column in $S_k$, thus, the last row of $M_k$ has only zeros, i.e., $M_k[3] = [0,0,0,0]$. To solve the sparsity problem we apply a more compressed form of mapping matrices as follows.

**Definition 3.2 (Compressed mapping matrix).** Compressed mapping matrices between source tables $S_1, S_2, ..., S_n$ and target table $T$ are a set of row vectors $CM = \{CM_1, ..., CM_n\}, CM_k (k \in [1, n])$ is a row vector of size $c_T$, where

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

### Table 2: Notations used in the paper

| Notation   | Description                        |
|------------|------------------------------------|
| $T/S_k$    | Target table/the $k$-th source table |
| $D_k$      | Processed $k$-th source table in matrix form |
| $cT/C_k$   | Number of mapped columns in $T/S_k$ |
| $rT/C_k$   | Number of mapped rows in $T/S_k$   |
| $M_k/C_k$  | Full/compressed mapping matrix for $S_k$ |
| $I_k/C_k$  | Full/compressed indicator matrix for $S_k$ |
| $R_k$      | Redundancy matrix for $S_k$        |
We continue with the running example. Fig. 2a illustrates the compressed mapping matrices \( CM = \{CM_1, CM_2\} \). They can be directly generated from schema mappings without the generation of the mapping matrices \( M = \{M_1, M_2\} \).

### 3.2 Indicator matrix

We use the indicator matrix (denoted as \( I_k \)) to preserve the row matching between each source table \( S_k \) and the target table \( T \). Similar to the mapping matrix, a binary indicator matrix could be very sparse and its compressed form is preferred. Due to space restriction, we directly define the compressed indicator matrix.

**Definition 3.3 (Compressed indicator matrix).** Compressed indicator matrices between source tables \( S_1, S_2, \ldots, S_n \) and target table \( T \) are a set of row vectors \( CI = \{CI_1, \ldots, CI_n\}, CI_k (k \in \{1, n\}) \) is a row vector of size \( r_T \), where

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

Notably, for the downstream ML algorithms, not all but only partial data of an original source table will participate in the computation as features or labels. Thus, we transform the original tables \( S_1 \) and \( S_2 \) in Fig. 1a-b to their matrix forms \( D_1 \) and \( D_2 \) in Fig. 2b, which only include the mapped columns. The transformation also includes data preprocessing operations, e.g., one-hot encoding (female \( \rightarrow 0 \), male \( \rightarrow 1 \)). Fig. 2b shows the row matching of the running example and the compressed indicator matrices, \( CI_1 \) for \( S_1 \) and \( CI_2 \) for \( S_2 \).

### 3.3 Redundancy matrix

Data integration systems often need to handle data redundancy when multiple sources have overlapping values. Consider the example in Fig. 1, when a user query asks how many male passengers are in \( S_1 \) and \( S_2 \), the correct answer is four instead of five. That is, the overlapped row of Winchester should be counted only once. Such redundancy resides in the projection of shared rows on the overlapped columns. Similarly, to support ML models we also need to detect redundancy to avoid repeated computation, which might lead to erroneous results. Thus, we propose a declarative representation to capture redundancy, i.e., redundancy matrix. To prepare for its definition, we first discuss how each source table contributes to the target table materialization. With the mapping matrix \( M_k \) and indicator matrix \( I_k \), we can transform a source table \( D_k \) to an intermediate matrix with the same shape as \( T \), denoted as \( T_k \).

\[
T_k = I_k D_k M_k^T
\]

Fig. 2c shows \( T_1 \) and \( T_2 \) of the running example. The red values in \( T_2 \) are the repeated values that already appeared in \( T_1 \). It is easy to see that \( T_k \) indicates the contribution from each source \( S_k \). However, due to the aforementioned redundancy issue (red values in \( T_2 \)), we cannot make a simple matrix addition to obtain the target table. For instance, \( T_1 + T_2 \neq T \) in Fig. 2b-c. This is why we need the redundancy matrix, which is defined below.

**Definition 3.4 (Redundancy matrix).** A redundancy matrix \( R_k \) of source table \( S_k \) is a binary matrix with the shape of \( r_T \times c_T \), where

\[
R_k[i,j] = \begin{cases} 
0, & \text{if } T_k[i,j] \text{ is redundant} \\
1, & \text{otherwise}
\end{cases}
\]

Note that before we say the data of a source table is redundant, we first need to specify which source table is the base table. For example, in Case 1-3 of Table 1, if we specify \( S_1 \) as the base table, then we consider the overlapped values in \( S_2 \) are redundant, and only need to generate a redundancy matrix for \( S_2 \). For completeness we can consider that the redundancy matrix for the base table is an all-ones matrix, which has all the element values equal to one. Fig. 2c shows \( R_2 \) for \( S_2 \) given the running example, which is computed based on mapping and indicator matrices of \( S_1 \) and \( S_2 \).

**Discussion.** With the three proposed types of matrices, we are not trying to provide a final solution. Instead, we offer a new perspective of representing DI information, i.e., the relationships between data sources and the target table, in an ML-compatible form. The definition and use cases of these matrices, could be extended beyond what has been discussed here. As the core of our vision, they bridge the logic-based DI scenarios with ML use cases, and answer the question \( Q \) in Sec. 1. In the next section, we explain two research opportunities on how these matrices are useful to optimize and automate ML applications.

### 4 RESEARCH OPPORTUNITIES

In this section, we dive into the promising research opportunities brought by the three proposed matrices. In Sec. 4.1, we show that the three types of matrices bring enhancement and new challenges to in-database ML factorization [30, 39, 43], which studies factorizing and pushing ML models to joinable source tables. In Sec. 4.2, for federated learning, our proposed matrices lead to new automation opportunities for reducing manual dataset transformation effort and reducing data traffic between silos.
4.1 Generalization for ML factorization

Recent advances in in-databases ML factorization [1, 8] have proposed to factorize linear algebra (LA) operators and execute them in each source table. In this way, the overhead of joining database tables is avoided. The existing works in this line of research, assume that tables are joinable via Key-foreign Key (PK-FK) relationships or equi-joins within a database.

To enable LA factorization in scenarios of Table 1, besides the three types of matrices, we also need to extend the existing LA rewrite rules. Next, we briefly discuss how to apply our indicator/mapping/redundancy matrices, together with new research opportunities. Here we use the example of LA operator left matrix multiplication (LMM) and its rewrite rule from [8]. Given a matrix $X$ with the size $c_T \times c_X$, the LMM of $T$ and $X$ is denoted as $TX$. For better understanding, below we use full mapping/indicator matrices $M_k$, although in the implementation we generate and utilize their compressed forms $CM_k$ and $CL_k$. Below are the existing LMM rewrite rule [8] and our proposed rule, which has two steps.

\[
TX \rightarrow I_1(D_1X[1 : c_S,]) + I_2(D_2X[c_S + 1 : c_T]) \quad [8]
\]

\[
TX \rightarrow I_1D_1M_1^TX + ((I_2D_2M_2^T) \circ R_2)X \quad [Ours]
\]

1. **Local result generation.** We first compute $I_kD_kM_k^T$ for each source table. In this step, to reduce computation overhead, we reorder the matrix multiplication sequence, similar to the optimization of join ordering in databases.

2. **Local result assembling.** The main task is to detect and remove computation redundancy by applying the redundancy matrices. For instance, we continue with the running example. Consider $D_1$ as the base table while $D_2$ is redundant. To obtain the correct final LMM result, here we perform a Hadamard Product $\circ$ (element-wise multiplication) between $I_2D_2M_2^T$ and the redundancy matrix $R_2$. In this way, we drop the redundant intermediate results indicated by the redundancy matrix $R_2$. Fig. 2c shows the results of $TX$ and $(I_2 \circ R_2)X$. It is easy to verify that their addition is the same as $TX$.

**Enhancement over existing work.** First, to compute the local LMM result, in the above rule (1) [8], $X$ is partitioned as $X[1 : c_S,]$ and $X[c_S + 1 : c_T,]$. Because the columns of $T$ are assumed to be two disjoint sets from $D_1$ and $D_2$. As shown in Table 1, it is common that data sources share overlapping columns. To meet this end, the mapping matrix comes in handy. In our modified rule (2), with the mapping matrix $M_k$, we can have more flexibility in choosing the columns of $S_k$. Second, to compute the final result, in the original rule (1) the two local LMM results (i.e., $D_1X[1 : c_D,]$ and $D_2X[c_D + 1 : c_T,]$) are simply added up via indicator matrices $I_1$ and $I_2$. However, as we have shown, we need to handle redundancy when generalizing the LA factorization problem.

**Future challenges.** Although not elaborated on in this work, typical DI questions such as the soundness, completeness, and efficiency might deserve more discussion when designing and implementing LA rewrite rules over all factorizable LA operators and ML models listed in the existing factorization works.

4.2 Automating data transformation for FL

In federated learning, a crucial prerequisite is establishing alignments among data silos, i.e., obtaining their column and row matching. This typically requires ML engineers to prepare a subset of local data by adding or removing feature and instance candidates from different data silos. It costs massive manpower, or programming efforts to collect, prepare and transform data from the sources, which also involves tiresome re-engineering. With our proposed mapping and indicator matrices, the subsets of local data can be represented and embedded in the federated models, which has great potential to automate the whole process. In what follows, we explain our intuition with the vertical federated linear regression (FLR) algorithm from [48] and Case 2 in Table 1. The FLR training objective is:

\[
\min_{\Theta_A, \Theta_B} \sum_i \left( \Theta_A X_A^{(i)} + \Theta_B X_B^{(i)} - y^{(i)} \right)^2 \quad ,\text{where}
\]

$X_A$ and $X_B$ are feature spaces of $S_1$ and $S_2$ respectively, $Y$ is the label space of $S_1$, $\Theta_A$ and $\Theta_B$ are the local FLR model parameters of $S_1$ and $S_2$. $i$ denotes the row index of data instances in the matrix.

The performance of trained FLR models depends on the quality of $X_A$ and $X_B$, which are prepared before training and fixed during training. Refining the performance of FLR models typically requires regenerating $X_A$ and $X_B$. With our mapping and indicator matrices, we can integrate the generation of $X_A$ and $X_B$ into the FLR training as an end-to-end optimization procedure. By denoting $X_A$ as $I_1D_1M_1^TX$ and $X_B$ as $I_2D_2M_2^T$, we rewrite the FLR objective as:

\[
\min_{\Theta_A, \Theta_B, I_1, I_2, M_1, M_2} \sum_i \left( \Theta_A(I_1D_1M_1^T)^{(i)} + \Theta_B(I_2D_2M_2^T)^{(i)} - y^{(i)} \right)^2 \quad .\text{Optimization and automation opportunities.}
\]

The new FLR objective can be optimized by alternatively training $\Theta_A, \Theta_B$ and $I_1, M_1, I_2, M_2$. While $\Theta_A, \Theta_B$ can be trained by following the secure federated learning procedure [48], efficiently and effectively training $I_1, M_1, I_2, M_2$ are challenging problems. For one thing, the conventionally centralized data selection approaches cannot be directly applied to the decentralized federated training because they impose heavy communication burdens. Therefore, communication-efficient data selection solutions are required to favor a fast search for optimal $I_1, M_1, I_2, M_2$. For another, $I_1, M_1, I_2, M_2$ contain meta-data of different data sources, and therefore they should be trained in a privacy-preserving manner. These challenges open up new research directions that incorporate multiple disciplines involving data integration, federated learning, and cryptography.

5 AMALUR: A DATA SCIENCE PLATFORM

We are currently developing a data science platform Amalur. It features an end-to-end, scalable training of ML models, and reduces the manual work of integrating data. Fig. 3 shows Amalur’s software stack and key components relevant to this paper. For the sake of brevity we omit details of other components that we have implemented in past works [17, 26], such as metadata extraction and profiling [27, 41], schema mapping [18] and matching [27], as well as query rewriting and data exchange [19].

**Code-less data preparation.** To ease the data engineering burden of users (e.g., complex SQL queries or Python scripts), data scientists can choose datasets, select features, and transform data (e.g., null
which will be extended from our earlier work [34].

API-compatible with Pandas and NumPy respectively. These two Amalur implements its own Dataframe and Array library, which are value replacement, one-hot encoding, normalization, training/test dataset split) for the downstream ML pipelines. We support these via a code-less UI. The UI’s backend then generates the corresponding python code and executes them over DataFrames which are built from the selected columns of source tables.

**Programming model &Intermediate representation.** Amalur follows the paradigm of “lazy” Dataframe libraries to enable data scientists to express their pipelines. Similar to [40] and Grizzly [16], Amalur implements its own Dataframe and Array library, which are API-compatible with Pandas and NumPy respectively. These two libraries enable data scientists to author end-to-end data science pipelines. Then they are lifted into an intermediate representation, which will be extended from our earlier work [34].

**Federation metadata** includes the mapping, indicator and redundancy matrices (Sec. 3). For each data source used in the ML pipeline, its matrices are automatically generated as explained in Sec. 3. The federation metadata also includes the size (data types, number of columns and rows), the dataframe/matrix sparsity as well as the location of a given dataset (the data silo in which it resides, alongside access information), etc. The metadata enables various cost-based optimizations and rewritings.

**Rewriting & Optimization.** Amalur’s optimizer and query rewriting module receive the lifted data science pipeline, federation metadata (virtual tables, data sizes, etc.), and the three types of matrices. It then reasons about the data relationships and provenance, and can perform optimizations across linear and relational algebra operations [32].

**Compilation, Orchestration and Distribution.** Finally, the optimized lifted program is compiled into concrete programs that can be executed on top of existing platforms such as TensorFlow or Spark. The compiled binaries comprise linear and relational algebra operations as well as control-flow that is executed on an orchestrator/driver program. The executables can be shipped to the different sites that contain the data, and return results such as data extracted from remote sites, model parameters that need to be merged, and matrices that take part in the final computations of the orchestrator program.

6 RELATED WORK

Machine learning has been applied to improve key operations of data integration such as schema matching [2, 7], data matching [7, 10, 25, 47]. However, except for data cleaning [28, 29, 36], little has been discussed in terms of using the key DI operations to facilitate machine learning [12], which is the goal of this paper. The group of works most related to our work, is in-DB machine learning.

**In-DB machine learning.** Integration of machine learning in relational databases has become one of the most important database research topics [38]. A common solution is to link ML algorithms with databases via carefully crafted user-defined functions (UDFs). Another way is to extend the data types for vector/matrix and SQL queries for ML models [22, 37]. Recent systems propose new intermediate data presentations [23, 33] or even novel tensor-based data models [4]. A more fundamental line of research is to support the formal languages of both worlds, i.e., relational algebra and linear algebra [1, 8, 24, 30, 46]. Notably, the above approaches mostly propose a representation as the data abstraction. In contrast, in data integration it is important to understand data sources and their relationships. Thus, although we have also applied matrix-based data abstraction (\(D_1, D_2\) in Fig. 2b), our main contribution, i.e., three proposed matrices, are representations of *dataset relationships*. Those capture the matching and redundancy for columns and rows from different data sources.

**Representations of dataset relationship.** The term *indicator matrix* has been proposed in [8, 30] for primary PK-FK-joinable or equi-joinable tables. They share similar ideas to ours in terms of using binary matrices to indicate the matches between rows from different tables. In our vision, the semantics of indicator matrices are richer. Besides row-level matching of inner joins as discussed in [8, 30], we have also covered how users choose to combine the datasets, i.e., the table-level relationships such as outer join, left join, and union as shown in Table 1. Moreover, the declarative representations in our vision include also mapping matrix and redundancy matrix. They preserve another two fundamental aspects of relationships among sources, i.e., source-to-target schema-level mapping and data redundancy. It then leads to new research challenges of LA rewriting, as discussed in Sec. 4.1, and also broadens future research directions, e.g., federated learning as discussed in Sec. 4.2.

7 CONCLUSION

In this work, we focus on data integration challenges over different sources for machine learning tasks. We proposed three types of matrices for schema mapping, data matching, and data redundancy. With these simple, easy-to-use representations, we have shown the new opportunities for model factorization and federated learning, which enable new systems such as our data science platform Amalur. Our vision is that the proposed representations become one of the first steps towards bridging the recent advances in machine learning with the well-studied classical data integration field. We hope that our proposed solution and outlined challenges would inspire more interesting future work.
