The VLDB Journal manuscript No.
(will be inserted by the editor)

TablePuppet: Towards a Generic Framework for Learning over Relational Tables

Lijie Xu · Chulin Xie · Yiran Guo · Haiyang Lu · Gustavo Alonso · Bo Li · Guoliang Li · Wei Wang · Wentao Wu · Ce Zhang

Received: date / Accepted: date

Abstract Learning over (distributed) relational tables (LRT) requires applying SQL queries that involve costly operations such as *joins* and *unions* to compose the training dataset, followed by model training atop the query results. This paradigm often introduces considerable computation, storage, and communication overhead that cannot be addressed by existing approaches. In this paper, we propose `TablePuppet`, a generic framework that can significantly reduce the overhead of LRT. We first formalize the LRT problem as *learning over union of conjunctive queries* (UCQ). We then decompose the learning process into two steps: (1) *learning over join* (LoJ), followed by (2) *learning over union* (LoU). In essence, LoJ pushes learning down to the individual tables being joined, while LoU further pushes learning down to the horizontal partitions/shards of each table. This two-step decomposition approach enables efficient distributed training without raw table sharing while preserving model accuracy. `TablePuppet` supports two standard ML optimization strategies, *stochastic gradient descent* (SGD) and *alternating direction method of multipliers* (ADMM), and can accommodate both centralized and distributed environments. In addition, `TablePuppet` introduces computation and communication optimizations to handle duplicate tuples introduced by joins, while further offering privacy guarantees for federated learning (FL) scenarios. Experimental evaluation results show that `TablePuppet` achieves comparable model accuracy to centralized baselines running directly on top of the SQL query results. Moreover, the SGD and ADMM algorithms implemented atop `TablePuppet` take less communication/training time to converge compared to the state-of-the-art approaches.

Keywords Machine learning, relational tables, SGD, ADMM, federated learning

1 Introduction

Relational databases are the cornerstone of enterprise data management, accounting for ~72% of the overall popularity ranking of all data systems [6]. A recent report [64] also indicates that about 66-86% of the data in practical data science tasks, such as retail, insurance, and marketing, is stored in relational formats. To extract valuable insights for tasks like financial and medical analysis, data analysts need to first perform SQL queries and then train ML models on the query result, a process known as *learning over relational data* [76,50,74,16,87,75,73]. Since most relational datasets are typically normalized and stored across multiple tables [68], these SQL queries often require joins to compose the training data. Moreover, in distributed environments, such as distributed/cloud databases, union operations are necessary alongside joins to combine horizontally sharded data into a unified dataset before ML training [77,49,13,9].

Compared to machine learning (ML) over centralized data such as a single table, learning over (distributed) relational data introduces new challenges. First, join operations are both computationally expensive and storage-intensive. The redundancy introduced by joins—both in primary key-foreign key (PK-FK) joins and many-to-many (M:N) joins—can dramatically increase the size of the resulting dataset, often by an order of magnitude or more. This leads to substantial storage overhead when storing or materializing the joined tables using centralized approaches. Second, joins and unions introduce significant communication overhead in distributed database environments. When tables are partitioned across

· Lijie Xu, Yiran Guo, Haiyang Lu, Wei Wang,
Institute of Software, Chinese Academy of Sciences
· Chulin Xie, Bo Li, UIUC, Urbana-Champaign, USA
· Gustavo Alonso, ETH Zürich, Zürich, Switzerland
· Guoliang Li, Tsinghua University, Beijing, China
· Wentao Wu, Microsoft Research, Washington, USA
· Ce Zhang, University of Chicago, Chicago, USA

multiple nodes, geographic regions, or even federated organizations, the cross-node joins/unions will introduce high communication costs and privacy concerns.

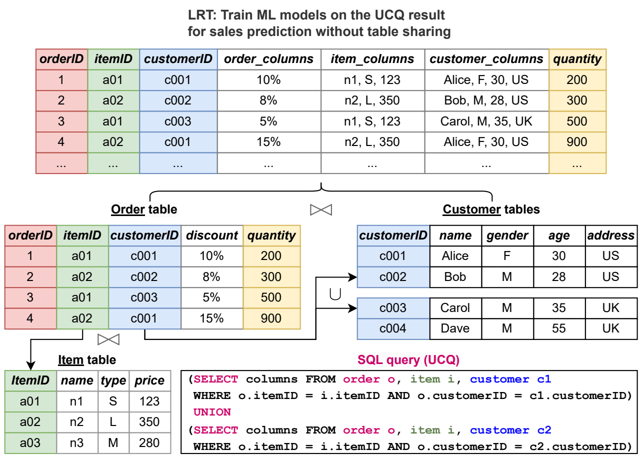
To address these challenges, both the database and ML research communities have proposed several approaches. The database community proposed factorized approaches for ML over joins, namely *factorized learning* or *learning over factorized joins* [76,50,74,16,75,73]. These methods decompose specific ML algorithms, such as generalized linear models and linear algebra, into sub-computations on individual tables. However, these approaches rely on the *gradient descent* (GD) algorithm for the decomposition and ML training, which converges slower than the widely-used *stochastic gradient descent* (SGD). Factorized approaches also focus on centralized (single-machine) environments. In contrast, the ML community has proposed federated learning approaches [47, 48,45,56], such as *horizontal/vertical federated learning* (HFL/VFL), which are better suited for distributed environments. For example, Teradata adopted HFL techniques to reduce data movement in their parallel clusters of a shared-nothing architecture [71]. However, both HFL and VFL are limited to single-table scenarios (a single table with either horizontal or vertical partitions), and do not support joins [56, 52]. Recent studies in both VFL and HFL [56,52] highlight the importance of integrating SQL queries with FL, such as supporting one-to-many data alignment and federated databases, as a key future direction.

To tackle the challenges of learning over relational tables (LRT) in centralized and especially distributed environments, we formalize the LRT problem as *learning over the union of conjunctive queries* (UCQ), a well-established concept in database theory [79]. We refer to relational tables that are to be joined as *vertical tables*, as they contain distinct (feature) columns. The horizontal partitions of these vertical tables, which we call *horizontal tables*, can be distributed across multiple nodes or regions. To construct a complete dataset for model training, we must first union the horizontal tables and then join the vertical tables. This process is conceptually equivalent to performing a UCQ across all involved tables. For ML training algorithms, we aim to support both SGD and *alternating direction method of multipliers* (ADMM). SGD is a widely-used algorithm, while ADMM is particularly well-suited for distributed environments, though its convergence properties for non-convex models are less extensively studied [40,85]. Our problem formulation can be seen as a generalization of existing ML over joins and federated learning problems, but introducing new challenges: how to efficiently support complex SQL operations (joins/unions) and diverse training algorithm (SGD/ADMM) simultaneously for LRT, especially in a distributed environment.

In this paper, we propose `TablePuppet`, a generic framework for LRT. `TablePuppet` decomposes the learning process into two steps: (1) *learning over join* (LoJ), followed by (2) *learning over union* (LoU). Essentially, LoJ pushes learning on the joined table down to the vertical tables being joined, and LoU further pushes learning on each vertical table down to the horizontal partitions/shards of each table. This two-step decomposition process facilitates `TablePuppet` to break down the global optimization problem of SGD and ADMM into independent sub-optimization problems on top of individual tables. In addition, `TablePuppet` optimizes and minimizes the computation and communication overhead caused by duplicate tuples generated from joins. Moreover, `TablePuppet` incorporates privacy-preserving mechanisms to provide formal *differential privacy* (DP) guarantees, making it well-suited for geo-distributed and federated scenarios, such as federated learning (FL), where protecting data privacy is crucial. For the storage overhead,

The implementation of `TablePuppet` adopts a server-client architecture, where each client represents a node that holds an individual table or its horizontal partition. The server, which can be either an independent node or one of the clients, coordinates the model training process. The global ML model over the joined tables is decoupled into local models maintained by the clients, and the server and clients collaboratively and iteratively train these local models. To unify the implementation of SGD and ADMM training processes, `TablePuppet` introduces three physical operators that abstract their computation and communication: (1) a *LoJ operator*, (2) a *LoU operator*, and (3) a *client operator* for the model updates inside clients. This abstraction facilitates the implementation of SGD/ADMM algorithms not only for LRT (**TP-UCQ-SGD** and **TP-UCQ-ADMM**) but also for simpler join-only scenarios, i.e., without horizontal partitions (**TP-Join-SGD** and **TP-Join-ADMM**). Specifically, in the LoJ step, the server uses a table-mapping mechanism to build a logical joined table of the UCQ result, which records the correspondence between rows of the joined table and the original tables. Since this mapping is significantly smaller than the full joined table and can fit in memory, `TablePuppet` does not incur on-disk storage overhead in contrast to centralized methods.

We analyze the computation and communication complexity of `TablePuppet`. We further study the effectiveness and efficiency of `TablePuppet` by evaluating model accuracy and performance of SGD/ADMM atop `TablePuppet`. For model accuracy, we consider directly training centralized ML models on the fully joined table as the baseline approach. Our experiments show that `TablePuppet` can achieve model accuracy comparable to this (strongest) baseline. Our approach also converges faster than factorized approaches which use GD-based algorithms. For performance, we focus on studying the communication overhead as it is the primary bottleneck in distributed environments [71,60,48]. Experimental results show that our optimized SGD/ADMM atop `TablePuppet` takes less communication time and less training time to con-


LRT: Train ML models on the UCQ result
for sales prediction without table sharing
Order table
⋈
Customer tables
∪
⋈
Item table
SQL query (UCQ)

orderID	itemID	customerID	order_columns	item_columns	customer_columns	quantity
1	a01	c001	10%	n1, S, 123	Alice, F, 30, US	200
2	a02	c002	8%	n2, L, 350	Bob, M, 28, US	300
3	a01	c003	5%	n1, S, 123	Carol, M, 35, UK	500
4	a02	c001	15%	n2, L, 350	Alice, F, 30, US	900
...	...	...	...	...	...	...

orderID	itemID	customerID	discount	quantity
1	a01	c001	10%	200
2	a02	c002	8%	300
3	a01	c003	5%	500
4	a02	c001	15%	900

customerID	name	gender	age	address
c001	Alice	F	30	US
c002	Bob	M	28	US

c003	Carol	M	35	UK
c004	Dave	M	55	UK

ItemID	name	type	price
a01	n1	S	123
a02	n2	L	350
a03	n3	M	280

```
(SELECT columns FROM order o, item i, customer c1
 WHERE o.itemID = i.itemID AND o.customerID = c1.customerID)
 UNION
(SELECT columns FROM order o, item i, customer c2
 WHERE o.itemID = i.itemID AND o.customerID = c2.customerID)
```

Fig. 1 A LRT example, using UCQ (with SQL syntax) to express the joins and unions over three tables.

verge (to similar accuracy) compared to the state-of-the-art SGD/ADMM implementations.

In summary, this paper makes the following contributions:

– We formalize the problem of learning over relational tables (LRT) as *learning over the UCQ result*, which can accommodate the centralized and distributed database environments.
– We present `TablePuppet`, a generic framework that can push learning down to individual tables without table sharing. It supports both SGD and ADMM, with computation and communication optimization as well as privacy guarantees.
– We implement `TablePuppet` using a server-client architecture with three high-level operators, which unifies the design and implementation of SGD/ADMM algorithms.
– We provide a theoretical analysis of the computation and communication complexity of `TablePuppet`.
– We evaluate `TablePuppet` on real-world datasets with multiple ML models in diverse scenarios. Our evaluation results demonstrate its effectiveness and efficiency.

2 Learning over Relational Tables (LRT)

We start by an example scenario of LRT, followed by a formal problem statement. We further discuss limitations of existing approaches when facing the challenges brought in by the LRT problem.

2.1 Example Scenario of LRT

Assume there is a large international store with branches in US and UK. As shown in Figure 1, the store owns an **order** table, an **item** table, and a **customer** table. The store wants to use these three tables to train a user purchase model for sales prediction. The label column is *quantity*, while the other columns of the three tables can be viewed as feature columns. To obtain the complete training data for this data science task, one needs to *join* the three tables **order**, **item**, and **customer** with the following join predicates:

$$\textbf{order}.itemID = \textbf{item}.itemID \quad \textbf{and}$$

$$\textbf{order}.customerID = \textbf{customer}.customerID.$$

The join columns can contain duplicates. For example, as shown in Figure 1, the **order** table contains multiple tuples with the same *itemID* or the same *customerID*. Consequently, the join result can be substantially larger than each individual table.

Each table involved in the join can be *horizontally* partitioned into multiple tables, and each horizontal table can be owned by a different node/region/organization. In the example shown in Figure 1, the **customer** table is partitioned into two tables, **customer**$_1$ and **customer**$_2$, that are managed by two branches of the store in US and UK. This necessitates an additional *union* over the join results. In database literature, the above operations can be naturally expressed by *union of conjunctive queries* (UCQ) [79].

2.2 Problem Formalization

Suppose that there are M relational tables $\{T_i\}_{i\in[M]}$. The i-th table T_i consists of Q_i horizontal partitions/shards as $T_i = \cup[T_i^1, \ldots, T_i^{Q_i}]$, where each horizontal shard/table T_i^q is owned by a client c_i^q. In practice, the client can be a node in a local cluster, a cloud region, or a federated organization. As shown in Figure 2, each T_i^q contains n_i^q tuples and d_i features, and therefore the full table T_i contains $n_i = \sum_{q=1}^{Q} n_i^q$ tuples and d_i features (i.e., $T_i \in \mathbb{R}^{n_i \times d_i}$). The total client count is Q, which is the sum of Q_i. We can perform SQL (UCQ) queries on these tables $\{T_i^q\}_{q\in[Q_i],i\in[M]}$. We assume that one of the $\{T_i\}_{i\in[M]}$ tables contains the *label* column.

We denote the fully *joined table* (i.e., the UCQ result) as

$$X = \bowtie [T_1, \ldots, T_M] = [X_1, \ldots, X_M],$$

which contains N tuples, $d = \sum_{i=1}^{M} d_i$ feature columns, and a label column $y \in \mathbb{R}^N$. A single server is responsible for coordinating these clients to jointly train machine learning models on X. We can now define the optimization problem associated with LRT as Eq. 1:

$$\min_{\Theta} \frac{1}{N} \sum_{j=1}^{N} \ell(\Theta; X_{i,j}, y_j) + \beta \mathcal{R}(\Theta), \quad \text{where} \tag{1}$$

$$X_{i,j} = \{\bowtie [T_1, \ldots, T_M]\}_{i,j}, T_i = \cup[T_i^1, \ldots, T_i^{Q_i}], i \in [M].$$

Here, we assume that the server has obtained and stored the UCQ results as $X = [X_1, \ldots, X_M] = \{X_{i,j}\}_{i\in[M],j\in[N]}$, i.e., $X_{i,j}$ is the j-th tuple of X_i. In this case, we can directly train the

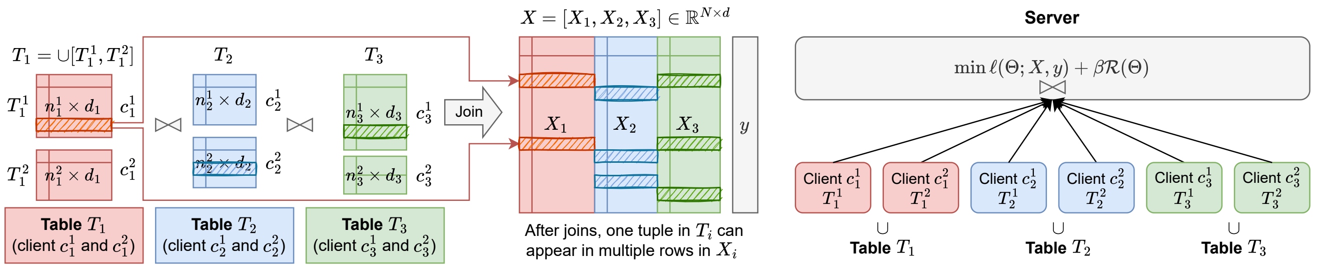


Fig. 2 Illustration of LRT where the table X with column y is the UCQ result. Each horizontal table T_i^q is owned by a client c_i^q.

ML models with parameters Θ on X in the server. Moreover, ℓ represents the *loss function* (e.g., cross entropy loss) and $\mathcal{R}$ represents the *regularization function* (e.g., L^2-norm) for Θ (with constant β). However, it is often too expensive or even infeasible (e.g., for data compliance reasons like GDPR regulation [2]) to perform UCQ queries on the tables, so the server cannot directly obtain X. As a result, Eq. 1 *cannot be directly solved as it is.*

2.3 Limitations of Existing Work

We present a brief overview of existing work and discuss its limitations in addressing the challenges raised by the LRT problem.

2.3.1 ML over Joins

The database community has proposed factorized approaches, such as *factorized learning* and *learning over factorized joins* [76,75,50,16,74] for the problem of ML over joins. However, these approaches have several limitations: (1) *centralized focus*—they are primarily designed for centralized environments, focusing only on join operations, and do not address the communication challenges in distributed environments with union operations; (2) *model specificity*—they are tailored to specific models, particularly linear models and matrix factorization models, limiting their applicability to broader ML tasks, like non-linear neural networks; and (3) *inefficiency of convergence*—they typically employ gradient descent (GD), which converges slower than more efficient methods such as stochastic gradient descent (SGD). Moreover, existing work on ML over joins cannot support ADMM, which is popular and well-suited for distributed environments.

2.3.2 Federated Learning (FL)

The machine learning community proposed horizontal/vertical FL (HFL/VFL) for learning over federated tables. HFL is a special case of LRT when there is only one table with multiple horizontal partitions (i.e., $M = 1$). Existing work on HFL, such as FedAvg [60], usually trains local models on individual horizontal partitions and then periodically synchronizes (e.g., by taking the average of) the model parameters or gradients. VFL is another special case of LRT. A single table is shared among multiple clients, where each client owns a *vertical partition*, i.e., a set of feature columns. Moreover, it assumes that the tuples in each vertical partition can be one-to-one perfectly aligned *without joins*. VFL has been extensively studied [56,19,82,35,38,89,92,25,41,55,80,17,42,43]. It typically leverages SGD [56,42,17] or ADMM [83,41] to train local models in distributed clients.

LRT is more complex than HFL and VFL, which involves a combination of vertical/horizontal tables, as well as one-to-many and/or many-to-many relationships introduced by joins. HFL/VFL cannot be directly run on the horizontal tables shown in Figure 2, because the objective function for model training is defined over the joined table and cannot be directly defined on these horizontal tables (e.g., T_1^1 and T_1^2) for HFL/VFL.

3 Overview of `TablePuppet`

The key idea of `TablePuppet` is to *decompose* the learning process involved in LRT into two steps: (1) *learning over join* (LoJ) followed by (2) *learning over union* (LoU). Specifically, the LoJ step pushes learning on the fully joined table to each vertical table T_i. It involves a *table mapping mechanism* to avoid actual joins and unions. It also employs optimization strategies that significantly reduce the computation and communication complexities from $O(N)$ to $O(n_i)$, where N is the length of the joined table and n_i is the length of each T_i. For each learning problem on the vertical table T_i resulted from the LoJ step, the LoU step further pushes computation down to each horizontal table/shard T_i^q. This (LoJ-LoU) two-step decomposition enables `TablePuppet` to flexibly integrate and optimize various types of learning algorithms. We demonstrate how to effectively implement two widely-used algorithms, SGD and ADMM.

`TablePuppet` coordinates the LoJ and LoU computation on vertical/horizontal tables using a server-client architecture (detailed in Section 4), where the server can be one of the clients or an independent instance. The global ML model with parameters Θ is partitioned into local models that are stored within individual clients. The server performs global server-side computation and coordinates the computation across clients, while clients perform client-side computation with local model updates.

3.1 Learning over Join on Vertical Tables (LoJ)

LoJ first uses a table mapping mechanism to build a *logical* joined table X to represent the UCQ result without data sharing among clients. It then decomposes/pushes learning over the joined table to each vertical table T_i. It finally performs computation and communication reduction for duplicate tuples introduced by joins.

3.1.1 ***Table-Mapping Mechanism***

To represent the global table of UCQ results, our key idea is to join the $\langle join\_key, row\_id\rangle$ columns of each table to get an index mapping between the logical joined table X and each vertical table T_i as $X_{i,j} \mapsto T_{i,p_i(j)}$, i.e., the j-th tuple of X_i comes from the $p_i(j)$-th tuple of T_i as shown in Figure 3(a), and p_i denotes the mapping function for T_i.

Step 1: To obtain this mapping, each client first extracts the $\langle join\_key, row\_id\rangle$ columns from its table and then sends these columns to the server. The clients that own the labels need to send the labels to the server as well. For federated scenarios, we describe the related privacy guarantees on $\langle join\_key, row\_id\rangle$ in Section 4.3.

Step 2: After collecting $\langle join\_key, row\_id\rangle$ columns from all the clients, the server first merges the collected $\langle join\_key, row\_id\rangle$ of horizontal tables to be $\langle join\_key, row\_id\rangle$ of vertical table T_i. It then joins the *join_key* columns specified in the UCQ and obtains the mapping $p_i = \left[X_{i,j} \mapsto T_{i,p_i(j)}\right]$ that maps X_i to T_i.

After that, the computation on each tuple of the joined table X, i.e., $X_{i,j}$, can be transferred to the computation on the corresponding tuple of the vertical table T_i, i.e., $T_{i,p_i(j)}$. The server also aggregates the received labels as y based on the table mapping.

3.1.2 ***Problem Decomposition and Push Down***

LoJ aims to push ML training on the joined table X down to each vertical table T_i. Following the vertical FL paradigm [56, 42], `TablePuppet` first decomposes the global model into local models by partitioning the global model parameters $\Theta = [\theta_1, \ldots, \theta_M]$, where θ_i denotes the parameters of the local model f_i associated with X_i. Then, `TablePuppet` transforms the LRT problem of Eq. 1 into an optimization problem on the predictions of local models (Eq. 2), where $h_{i,j}$ denotes the model prediction of f_i on the tuple $X_{i,j}, \forall j \in [N]$:

$$\min_{\{\theta_i\}_{i=1}^M} \frac{1}{N}\sum_{j=1}^{N} \ell\left(\sum_{i=1}^{M} h_{i,j}; y_j\right) + \beta \sum_{i=1}^{M} \mathcal{R}_i(\theta_i), \quad \text{where} \qquad (2)$$

$$h_{i,j} = f_i(\theta_i; X_{i,j}) = f_i(\theta_i; T_{i,p_i(j)}), T_i = \cup[T_i^1, \ldots, T_i^{Q_i}].$$

By default, the outputs $h_{i,j}$ from all local models are first aggregated at the server (e.g., summed) and then passed through a non-linear activation function (e.g., sigmoid or softmax) to form a unified prediction. The cross-entropy loss $\ell(\cdot; y_j)$ is then computed on this activated, aggregated output. This aggregation–activation–loss pipeline is the key step where cross-feature patterns are captured: although raw features from different vertical tables are never explicitly joined, their contributions are fused in the aggregated prediction and jointly optimized through the loss. Each local model can be either a linear model or a neural network, making `TablePuppet` applicable to *both* linear and nonlinear cases. For a linear model, $\theta_i \in \mathbb{R}^{d_i \times d_c}$ is the weight matrix with d_c classes; for a neural network, θ_i denotes its parameters.

After that, `TablePuppet` leverages the table-mapping mechanism to push the learning problem on top of X_i down to each vertical table T_i, enabling the computation of $h_{i,j} = f_i(\theta_i; T_{i,p_i(j)})$ for all $j \in [N]$ without requiring X_i explicitly.

However, we cannot directly solve the optimization problem of Eq. 2, since the server cannot directly compute its gradient $\nabla_{\theta_i}\ell_j = \frac{\partial \ell_j}{\partial \theta_i}$ where $\ell_j = \ell(\sum_{i=1}^M h_{i,j}; y_j)$, which requires the local tables that only reside in individual clients. To address this table sharing issue, we decompose the optimization problem of Eq. 2 into sub-problems and push each sub-problem down to the corresponding vertical table T_i.

Specifically, we discuss below how to push SGD and ADMM down to vertical tables, alongside necessary computation on the server side. For SGD, we focus on how to break down the gradient computation across vertical tables as well as the server. For ADMM, we focus on how to decompose the global optimization problem into separate and independent optimization problems, each associated with a vertical table. Compared to decomposed gradient computation in SGD, these independent optimization problems in ADMM enable more computation on the client side, thereby reducing the communication rounds with the server:

(1) SGD. We can decompose the optimization problem of Eq. 2 into two sub-problems, by the chain rule $\nabla_{\theta_i}\ell_j = \frac{\partial \ell_j}{\partial h_{i,j}}\frac{\partial h_{i,j}}{\partial \theta_i}$. The first sub-problem is to compute the partial derivative of ℓ_j w.r.t. $h_{i,j}$ as $\frac{\partial \ell_j}{\partial h_{i,j}}$ on the server side, since the server can obtain all the model predictions $h_{i,j}$ from clients. The second sub-problem is to compute the partial derivative of $h_{i,j}$ w.r.t. θ_i as $\frac{\partial h_{i,j}}{\partial \theta_i}$ on the client side. Consequently, the client with T_i can compute $\nabla_{\theta_i}\ell_j$ after receiving the $\frac{\partial \ell_j}{\partial h_{i,j}}$ from the server, and further compute the gradient of Eq. 2 as $\nabla_{\theta_i}F(T_i)$ using Eq. 5 shown in Table 2. Finally, the client

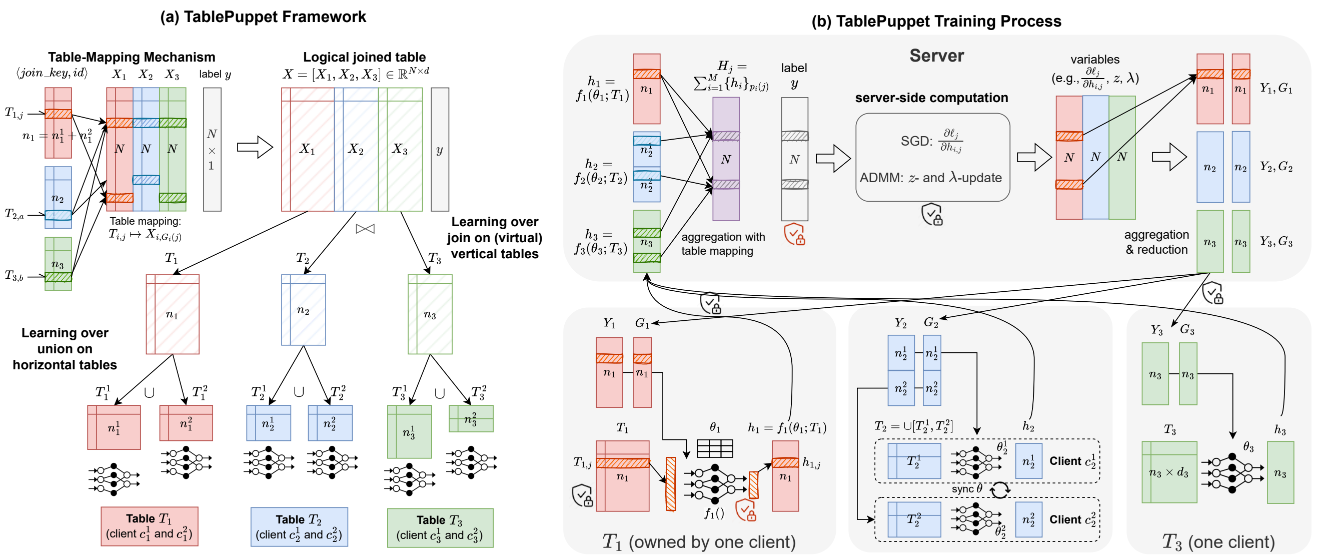

(a) TablePuppet Framework
Table-Mapping Mechanism
Logical joined table
Table mapping:
Learning over join on (virtual) vertical tables
Learning over union on horizontal tables
Table T_1 (client c_1^1 and c_1^2)
Table T_2 (client c_2^1 and c_2^2)
Table T_3 (client c_3^1 and c_3^2)
(b) TablePuppet Training Process
Server
server-side computation
SGD: $\frac{\partial \ell_j}{\partial h_{i,j}}$
ADMM: z- and λ-update
aggregation with table mapping
aggregation & reduction
variables
Client c_2^1
Client c_2^2
sync θ
T_1 (owned by one client)
T_3 (one client)

Fig. 3 `TablePuppet` framework and its training process. The red (black) locks indicate locations where DP is (can be) applied.

Table 1 Table of Notations.

Notation	Definition
T_i, T_i^q	Given a table T_i, the q-th horizontal partition/shard of T_i is T_i^q.
X_i	X_i refers to the i-th vertical slice of the joined table X.
$X_{i,j} \mapsto T_{i,p_i(j)}$	After the UCQ, the j-th tuple of X_i is from the $p_i(j)$-th tuple of T_i.
$h_{i,j} = f_i(\theta_i; x)$	θ_i refers to the parameters of the local model f_i, while $h_{i,j}$ is the prediction of model $f_i(\theta_i)$ on tuple x.
$\ell_j = \ell(x; y_j)$	The loss function ℓ for the element x with label y_j.
z_j, λ_j	z_j, λ_j are auxiliary/dual variables for ADMM in LoJ.
ρ	$\rho \in \mathbb{R}$ is a penalty parameter that can be tuned in ADMM.
$T_{i,j} \mapsto X_{i,G_i(j)}$	After the UCQ, $T_{i,j}$ appears multiple times in X_i and the corresponding set of row indices is denoted as $G_i(j)$.
$g \in G_i(j)$	Each $g \in G_i(j)$ denotes a specific tuple index in X_i, i.e., $X_{i,g}$ comes from $T_{i,j}$.
$T_{i,j}^q, \theta_i^q$	$T_{i,j}^q$ is the j-th tuple of T_i^q, and θ_i^q denotes model parameters w.r.t. T_i^q.
w_i, u_i	w_i, u_i are auxiliary/dual variables for ADMM in LoU.

with T_i can update its local θ_i using Eq. 6 with learning rate η. Now we have pushed the LoJ problem down to each vertical table T_i associated with client-side computation of Eq. 5 and Eq. 6, alongside server-side computation of Eq. 4. We can also use mini-batch SGD to compute these equations, by randomly subsampling a batch B from N and change N to B in each equation. However, the client-side computation complexity remains $O(N)$.

(2) ADMM. Unlike SGD, ADMM aims to decompose the global optimization problem of Eq. 2 into multiple independent optimization sub-problems. To achieve this, we follow the *sharing ADMM* paradigm [12] to rewrite Eq. 2 into the Eq. 3 below, by introducing auxiliary variables $z = \{z_j\}_{j=1}^N$ where $z_j \in \mathbb{R}^{d_c}$:

$$\begin{array}{ll}\text{minimize} & \frac{1}{N}\sum_{j=1}^N \ell(z_j; y_j) + \beta \sum_{i=1}^M \mathcal{R}_i(\theta_i), \\ \text{subject to} & \sum_{i=1}^M h_{i,j} - z_j = 0,\ h_{i,j} = f_i(\theta_i; T_{i,p_i(j)}).\end{array} \tag{3}$$

We then add a quadratic term to the Lagrangian of Eq. 3, known as *augmented Lagrangian* [12]. After that, we can solve the optimization problem using ADMM with three update steps, including the **server-side** z-update and λ-update (Eq. 7 and 8), as well as the **client-side** θ-update (Eq. 9), as shown in Table 2. In these equations, a^t refers to the value of a in the t-th epoch, whereas $\rho \in \mathbb{R}$ is a tunable penalty parameter. $\{\lambda_j\}_{j=1}^N$ are dual variables and $\lambda_j \in \mathbb{R}^{d_c}$. The $\{s_{i,j}\}_{j=1}^N$ are residual variables for each table T_i, where $s_{i,j} = \sum_{k=1,k\neq i}^M f_i(\theta_k; T_{k,p_k(j)}) - z_j$. This decomposition pushes independent optimization problems, i.e., θ_i-updates to each T_i, which requires less communication with the server, i.e., only after the θ_i-update finishes, compared to (mini-batch) SGD.

3.1.3 ***Reduction of Computation and Communication Cost***

While ADMM can reduce the number of communication rounds compared to SGD, its computation and communication complexity remains as $O(N)$. Both SGD and ADMM need to perform $\sum_{j=1}^N(\cdot)$ during client-side computation using Eq. 5 and 9. Here, N refers to the length of the joined table X, which can be orders of magnitude larger than the length n_i of each vertical table T_i, due to duplicate tuples introduced by joins. For example, if both $X_{i,a}$ and $X_{i,b}$ are generated by the

Table 2 The problem decomposition for LoJ, using SGD and ADMM with computation/communication reduction.

Algorithm	Server	Client with T_i
SGD	$\frac{\partial \ell_j}{\partial h_{i,j}}, \quad \forall j \in [N]. (4)$	$\nabla_{\theta_i} F(T_i) = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{\partial \ell_j}{\partial \theta_i} + \beta \frac{\partial \mathcal{R}_i}{\partial \theta_i}\right), (5)$ $\theta_i := \theta_i - \eta \nabla_{\theta_i} F(T_i). (6)$
ADMM	$z_j^t = \operatorname{argmin}_{z_j}\left(\ell(z_j; y_j) - \left(\lambda_j^{t-1}\right)^\top z_j + \frac{\rho}{2}\left\|\sum_{i=1}^{M} f_i(\theta_i^t; T_{i,p_i(j)}) - z_j\right\|^2\right), (7)$ $\lambda_j^t = \lambda_j^{t-1} + \rho\left(\sum_{i=1}^{M} f_i(\theta_i^t; T_{i,p_i(j)}) - z_j^t\right). (8)$	$\theta_i^{t+1} = \operatorname{argmin}_{\theta_i}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N} \lambda_j^{t\top} f_i(\theta_i; T_{i,p_i(j)}) + \frac{1}{N}\sum_{j=1}^{N}\frac{\rho}{2}\left\|s_{i,j}^t + f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right). (9)$
SGD (optimized)	$Y_{i,j} = \sum_{g \in G_i(j)} \frac{\partial \ell_g}{\partial h_{i,g}}, \quad \forall j \in [n_i], (10)$ $G_{i,j} = \lvert G_i(j)\rvert, \quad \forall j \in [n_i]. (11)$	$\nabla_{\theta_i} F(T_i) = \frac{1}{N}\sum_{j=1}^{n_i}\left(Y_{i,j}\frac{\partial h_{i,j}}{\partial \theta_i} + \beta G_{i,j}\frac{\partial \mathcal{R}_i}{\partial \theta_i}\right), (12)$ $\theta_i := \theta_i - \eta \nabla_{\theta_i} F(T_i). (13)$
ADMM (optimized)	Apart from Eq. 7 and 8, the server also performs: $Y_{i,j}^t = \sum_{g \in G_i(j)} (\lambda_g^t + \rho s_{i,g}^t), \quad \forall j \in [n_i], (14)$ $G_{i,j} = \lvert G_i(j)\rvert, \quad \forall j \in [n_i]. (15)$	$\theta_i^{t+1} = \operatorname{argmin}_{\theta_i}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{n_i}(Y_{i,j}^t)^\top f_i(\theta_i; T_{i,j}) + \frac{1}{N}\sum_{j=1}^{n_i}\frac{\rho G_{i,j}}{2}\left\|f_i(\theta_i; T_{i,j})\right\|^2\right). (16)$

Table 3 The problem decomposition for LoU, using optimized SGD and ADMM.

Algorithm	Coordinator (Server)	Client with T_i^q
SGD (optimized)	$\nabla_{\theta_i} F(T_i) = \frac{1}{N}\sum_{q=1}^{Q_i}\nabla_{\theta_i} F(T_i^q). (17)$	$\nabla_{\theta_i} F(T_i^q) = \sum_{j=1}^{n_i^q}\left(Y_{i,j}\frac{\partial h_{i,j}}{\partial \theta_i} + \beta G_{i,j}\frac{\partial \mathcal{R}_i}{\partial \theta_i}\right), (18)$ $\theta_i := \theta_i - \eta \nabla_{\theta_i} F(T_i). (19)$
ADMM (optimized)	$w_i^\tau = \operatorname{argmin}_{w_i}\left(\beta\mathcal{R}_i(w_i) + \frac{M\rho}{2}\left\|w_i - \overline{\theta_i^\tau} - \overline{u_i^{\tau-1}}\right\|^2\right), (20)$ $u_i^{q,\tau} = u_i^{q,\tau-1} + \theta_i^{q,\tau} - w_i^\tau. (21)$	$\theta_i^{q,\tau+1} = \operatorname{argmin}_{\theta_i^q}\left(\frac{1}{N} l(\theta_i^q; T_i^q) + \frac{\rho}{2}\left\|\theta_i^q - w_i^\tau + u_i^{q,\tau}\right\|^2\right). (22)$

same tuple $T_{i,k}$ after the join, $T_{i,k}$ will be visited twice during the client-side computation of SGD and ADMM. Moreover, the server needs to send its computation results to each client with T_i in each epoch, such as the partial gradient $\frac{\partial \ell_j}{\partial h_{i,j}}$ of SGD and the $s_{i,j}, \lambda_j$ of ADMM. These variables are of length N, which attributes to the $O(N)$ communication complexity.

To minimize the computation and communication overhead, we develop two optimization strategies for the LoJ step: (1) reduce the client-side computation on duplicate tuples by aggregating the variables on the server-side; and (2) communicate the aggregated variables instead of the original ones between the server and clients. Below we detail their implementations for SGD and ADMM:

(1) SGD. We figure out that the second part $\frac{\partial h_{i,j}}{\partial \theta_i}$ of the chain rule $\nabla_{\theta_i}\ell_j = \frac{\partial \ell_j}{\partial h_{i,j}}\frac{\partial h_{i,j}}{\partial \theta_i}$ is the same for all duplicate tuples from T_i, as it is only determined by the model function f_i and each tuple of T_i. Therefore, we can aggregate the part $\frac{\partial \ell_j}{\partial h_{i,j}}$ for duplicate tuples on the server side, and then send the aggregated variables to the clients for reducing the client-side computation.

To facilitate the aggregation on the server side, we first construct a *reverse* table mapping $T_{i,j} \mapsto X_{i,G_i(j)}$, which means $T_{i,j}$, i.e., the j-th tuple of T_i, is mapped to multiple tuples in X_i, denoted as $\{X_{i,g}\}_{g \in G_i(j)}$. Here, $X_{i,g}$ is the g-th tuple in X_i and $G_i(j)$ refers to the index set of the mapped multiple tuples. $|G_i(j)|$ denotes the total tuple number in the $G_i(j)$. We then aggregate the $\frac{\partial \ell_j}{\partial h_{i,j}}$ of duplicate tuples as shown in Eq. 10 and Eq. 11. Using the aggregated variables $Y_{i,j}$ and $G_{i,j}$, we can rewrite the client-side computation of Eq. 5 into Eq. 12, where G_i remains the same in each epoch.

Note that Eq. 12 has transformed the computation of $\sum_{j=1}^{N}(\cdot)$ to $\sum_{j=1}^{n_i}(\cdot)$ and therefore the computation overhead is reduced from $O(N)$ to $O(n_i)$. Moreover, the server can send $Y_i \in \mathbb{R}^{n_i \times d_c}$ and $G_i \in \mathbb{R}^{n_i}$ to the client that owns T_i, instead of sending $\frac{\partial \ell_j}{\partial h_{i,j}} \in \mathbb{R}^{N \times d_c}$ and the table-mapping that are in $O(N)$. Therefore, the communication overhead between the server and the clients is reduced from $O(N)$ to $O(n_i)$ as well. However, for mini-batch SGD, the server needs to aggregate for the duplicate tuples inside each batch, so the reduced computation and communication is between $O(n_i)$ and $O(N)$.

(2) ADMM. For the client-side computation of Eq. 9, duplicate tuples lead to duplicate computation of $f_i(\theta_i; T_{i,p_i(j)})$. We can perform an optimization strategy similar to SGD. The server combines and aggregates the variables (e.g., λ_j and $s_{i,j}$) of duplicate tuples as shown in Eq. 14 and Eq. 15. Using $Y_{i,j}$ and $G_{i,j}$, we can rewrite the θ_i-update as Eq. 16, which transforms the computation of $\sum_{j=1}^{N}(\cdot)$ to $\sum_{j=1}^{n_i}(\cdot)$ and thus reduces the computation overhead from $O(N)$ to $O(n_i)$. Moreover, the server can send Y_i and G_i to the client that owns T_i, instead of sending λ, s_i and the table-mapping that are in $O(N)$ as shown by Eq. 9. Therefore, the communication overhead is also reduced from $O(N)$ to $O(n_i)$.

3.2 Learning over Union on Horizontal Tables

LoJ has pushed ML training down to each vertical table T_i. The next question is how to push the computation down to each horizontal shard/table T_i^q. For SGD, we can decompose the gradient computation and synchronize it for local model updates, as the gradient computation on each tuple is independent. For ADMM, to decompose the optimization problem of θ_i-update, we can also use SGD. However, to reduce the number of communication rounds, we use horizontal ADMM instead to decompose it to independent optimization problems on horizontal tables:

(1) SGD. Our goal is to push the computation on T_i, i.e., Eq. 12 and Eq. 13, down to the clients that own $\{T_i^q\}_{q\in[Q_i]}$. Our key idea is to decouple the gradient computation while performing the same model update in each client as that of Eq. 13. First, we allocate the same model with the same θ_i for each client with T_i^q. Second, we decompose the gradient computation of Eq. 12 into sub-problems as follows: (1) each client first performs partial gradient computation on T_i^q as Eq. 18; (2) a coordinator of the clients then aggregates the partial gradients as Eq. 17 and sends the aggregate back to the clients; (3) each client updates the model using Eq. 19. In particular, the server can act as the coordinator as well.

(2) ADMM. For ADMM, we decompose an independent optimization problem, i.e., the θ_i-update, into sub-problems w.r.t. the T_i^q's, by leveraging *consensus ADMM* [12]. Conceptually, this decomposition "pushes" ADMM through the *union* operation down to the horizontal tables. Specifically, we first rewrite Eq. 16 as Eq. 23:

$$\begin{gathered}\text{minimize} \tfrac{1}{N}\sum_{q=1}^{Q_i} l\left(\theta_i^q; T_i^q\right) + \beta\mathcal{R}_i(\theta_i),\\ l\left(\theta_i^q; T_i^q\right) = \sum_{j=1}^{n_i^q}\left[(Y_{i,j}^{q,t})^\top f_i(\theta_i^q; T_{i,j}^q) + \frac{\rho G_{i,j}^q}{2}\left\|f_i(\theta_i^q; T_{i,j}^q)\right\|^2\right].\end{gathered} \tag{23}$$

Here, θ_i^q refers to the model parameters w.r.t. T_i^q. $T_{i,j}^q$ denotes the j-th tuple in T_i^q, and $Y_{i,j}^{q,t}$ refers to the j-th element of the q-th part of Y_i in the t-th epoch. We then rewrite Eq. 23 as Eq. 24 by introducing auxiliary variables w_i to approximate each θ_i^q:

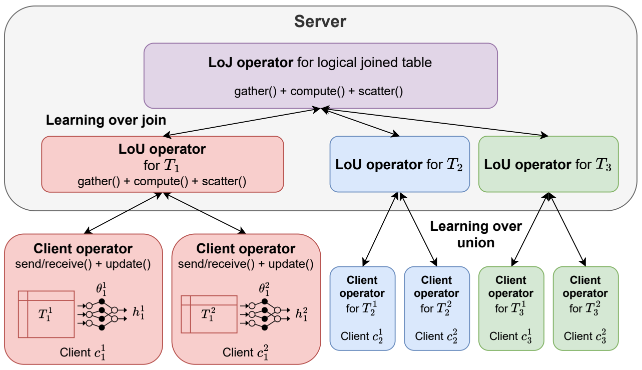


Fig. 4 The architecture of `TablePuppet`.

$$\begin{aligned}&\text{minimize} \ \sum_{q=1}^{Q_i}\tfrac{1}{N} l\left(\theta_i^q; T_i^q\right) + \beta\mathcal{R}_i(w_i),\\ &\text{subject to} \quad \theta_i^q - w_i = 0, \quad \forall q \in [Q_i].\end{aligned} \tag{24}$$

We can now use consensus ADMM to solve this optimization problem with three update steps (Table 3). Here, $\overline{\theta_i^\tau}$ is the average of $\{\theta_i^{q,\tau}\}_{q\in[Q_i]}$ and $\overline{u_i^{\tau-1}}$ is the average of $\{u_i^{q,\tau-1}\}_{q\in[Q_i]}$, where u_i is the scaled dual variable and τ denotes the τ-th epoch of the horizontal ADMM. The coordinator performs the w-update and u-update, while each client q that owns T_i^q can perform the θ_i^q-update. During each epoch of the horizontal ADMM, each client q sends its updated θ_i^q to the coordinator and the coordinator returns w_i and u_i^q to the client q for model update with one round of communication.

4 Implementation of `TablePuppet`

`TablePuppet` adopts a server-client architecture (Figure 4), where the global ML model with parameters Θ is partitioned into local models with $\{\theta_i^q\}_{i\in[M],q\in[Q_i]}$ stored by the clients. The server and clients collaboratively and iteratively train these local models. We abstract the training process as an *execution plan* of three *physical operators* (Section 4.1), which unifies the computation and communication of SGD/ADMM (Section 4.2). `TablePuppet` also ensures differential privacy for *both* features and labels (Section 4.3).

4.1 Training Process with Physical Operators

We illustrate the training process in Algorithm 1, which mainly consists of two loops: (1) the outer loop that conducts *learning over join* (LoJ) and (2) the inner loop that performs *learning over union* (LoU). To implement and unify the training processes of SGD and ADMM, we design three physical

Algorithm 1: `TablePuppet` training process.

```
Input: Suppose $X = \bowtie [T_1, \ldots, T_M]$ and
    $T_i = \cup[T_i^1, T_i^2, \ldots, T_i^{Q_i}]$. There are one server and $Q$
    clients with $Q = \sum_{i=1}^{M} Q_i$. Each client owns a local
    model and a table as $T_i^q$. The number of
    communication rounds for outer loop is $\mathcal{T}$ and inner
    loop is $\mathcal{T}'$. For SGD, $\mathcal{T} = KN/B$ and $\mathcal{T}' = 1$, where
    $K$ is the epoch number, $N$ is the tuple number of $X$,
    and $B$ is the batch size. For ADMM, $\mathcal{T} = K$.
1  Server performs table mapping (Section 3.1.1).
2  for communication round $t \in [\mathcal{T}]$ do    // Outer loop
3      for each client with $T_i^q$, $i \in [M]$, $q \in [Q_i]$ do
4          computes model predictions and send them to the
             server.
5      Server gathers and aggregates model predictions.
6      Server does server-side computation (SGD-/ADMM-Opt
         in Table 2).
7      Server scatters computation results to the clients.
8      for $i \in [M]$ do    // Inner loop
9          for communication round $\tau \in [\mathcal{T}']$ do
             // $\mathcal{T}' = 1$ if $Q_i = 1$
10             if $Q_i > 1$ then
11                 for each client with $T_i^q$, $q \in [Q_i]$ do
12                     computes and sends variables (SGD:
                         Eq. 18 with DP, ADMM: local model
                         parameters) to the server.
13                 Server performs coordinator-side
                     computation (SGD: Eq. 17, ADMM:
                     Eq. 20 and 21).
14                 Server scatters computation results to the
                     clients.
15             for each client with $T_i^q$, $q \in [Q_i]$ do
16                 performs client-side computation and
                     model update (SGD: Eq. 19, ADMM:
                     Eq. 22 with DP).
```

operators that abstract the computation and communication of `TablePuppet`. The three operators include a *LoJ operator*, a *LoU operator*, and a *client operator* for client-side model updates. As shown in Table 4, each operator contains three functions, including a *compute*() for computation, as well as *gather/scatter*() or *send/receive*() for communication.

(1) LoJ operator: It works on the server for LoJ, with the assumption that each LoU operator represents a vertical table T_i. It first uses *gather*() to collect model predictions from each LoU operator, and then aggregates these model predictions based on the table mapping. After that, it performs server-side computation via *compute*(). Finally, it performs *scatter*() to distribute the computation results to each LoU operator, as if the computation results were scattered to each T_i. As shown in Algorithm 1, the LoJ operator is responsible for the steps 5-7 of the training process.

(2) LoU operator: It works on the server for coordinating the clients with horizontal partitions of the same table. In the outer loop, it uses *gather*() to collect model predictions from clients, and combines them as model predictions on a vertical T_i, which are sent to the LoJ operator. It also applies *scatter*() to distribute computation results from the LoJ operator to the clients. In the inner loop, it gathers partial gradients or variables from clients using *gather*(), performs coordinator-side computation using *compute*(), and distributes computation results back to clients with *scatter*(). As shown in Algorithm 1, the LoU operator is responsible for the steps 13-14 of the training process.

(3) Client operator: It works on each client with horizontal table T_i^q. It applies *send*() to transmit model predictions, partial gradients, or model parameters to the LoU operator (coordinator) for aggregation and computation. It gets computation results, such as aggregated gradients and variables, from the LoU operator using *receive*(). Finally, it performs *compute*() for model update. As shown in Algorithm 1, the client operator is responsible for the steps 4, 12, and 16 of the training process.

Our communication operators are built on top of Ray [62], a distributed execution framework that uses gRPC over TCP for inter-node communication, rather than MPI or NCCL. This design aligns with existing FL frameworks such as Flower [7].

The primary reason for leveraging Ray's gRPC-based communication is that the VFL/UCQ setting differs fundamentally from traditional data-parallel distributed ML training within a local cluster. In VFL/UCQ, clients typically reside in different organizations or geographically distributed locations and often lack direct peer-to-peer connectivity. Our design only requires each client to establish a secure and firewall-friendly connection with a central coordinating server via TCP. This approach is robust and widely supported in real-world deployment environments [7].

Furthermore, clients hold different sets of features and in the UCQ scenario, they also process different samples. As a result, traditional data-parallel synchronization primitives such as All-Reduce are not applicable. These primitives are designed for homogeneous workloads where data is partitioned across samples with identical feature structures. In contrast, our framework follows a parameter-server-style architecture. Clients train local models and exchange intermediate results such as logits, gradients, or ADMM auxiliary variables with the server. This communication pattern is naturally supported by the task-actor model of Ray as well as its built-in networking capabilities.

4.2 SGD/ADMM with `TablePuppet`

By combining the three physical operators in various ways and adjusting the implementations of their inner functions, we can implement SGD/ADMM not only for LRT (i.e., join-union scenario) but also for a simpler join-only scenario (i.e. without horizontal partitions). As shown in Table 5, for the join-only scenario where each table has one client, we can just utilize the LoJ operator and the client operator to implement **TP-Join-SGD** and **TP-Join-ADMM**. For LRT, we use all

Table 4 The physical operators with computation and communication functions, which cover both SGD and ADMM.

Oper.	Function	SGD	ADMM
LoJ oper.	*gather*()	model predictions	model predictions
	compute()	Eq. 10, 11	Eq. 7, 8, 14, 15
	scatter()	partial derivatives	auxiliary variables
LoU oper.	*gather*()	model predictions, partial gradients	model predictions, model parameters
	compute()	Eq. 17	Eq. 20 and Eq. 21
	scatter()	aggregated gradients	auxiliary variables
Client oper.	*send*()	model predictions, partial gradients	model predictions, model parameters
	receive()	aggregated gradients	auxiliary variables
	compute()	Eq. 18 and 19	Eq. 22

Table 5 The SGD and ADMM algorithms atop `TablePuppet`.

Scenario	Algorithm	Operator implementation
join-only	**TP-Join-SGD**	LoJ operator: Eq. 10, 11, Client operator: Eq. 12, 13
	TP-Join-ADMM	LoJ operator: Eq. 7, 8, 14, 15, Client operator: Eq. 16
join-union (UCQ)	**TP-UCQ-SGD**	As that in Table 4
	TP-UCQ-ADMM	As that in Table 4

three operators with functions shown in Table 4 to implement **TP-UCQ-SGD** and **TP-UCQ-ADMM**.

4.3 Privacy Guarantees

For geo-distributed and federated scenarios, we need to defend the tables and communicated variables from attacks. To address this, `TablePuppet` provides additional *differential privacy* (DP) [23] to ensure data privacy for both the table features and labels. We adapt the DP definition for LRT as follows.

Definition 1 (**(ϵ, δ)-DP [23]**) A randomized algorithm $\mathcal{M} : \mathcal{X}^n \mapsto \Theta$ satisfies (ϵ, δ)-DP if, for every pair of neighboring datasets (i.e., the joined tables) $X, X' \in \mathcal{X}^n$ that differ by one single tuple, and every possible (measurable) output set $E \subseteq \Theta$, the following inequality holds: $\Pr[\mathcal{M}(X) \in E] \leq e^{\epsilon} \Pr[\mathcal{M}(X') \in E] + \delta$.

We focus on addressing the key problem of *where* to inject DP noises. Possible options for noise injections include the individual tables, local model parameters, or the diverse variables communicated between the server and clients, shown as black/red locks in Figure 3(b). The added noises should ensure the privacy of labels/features in SGD/ADMM optimization algorithms. Our approach uses separate differential privacy (DP) mechanisms to protect both the labels and the features. This separation is motivated by the structure of the data: the labels are stored on the server, while the table features reside on the client side. Our method is flexible and can be adapted to other scenarios. For example, if the server is a client that owns the label (and the features), there is no need to add noise to the label itself. In such cases, we can apply a one-time noise injection locally to protect both the label and the features together on that client side. To protect labels, we **add noises to the raw labels**, namely label-level DP (Section 4.3.1). To protect features, we **perturb local model parameters** on the client side (Section 4.3.2) instead of perturbing the communicated variables. Our rationale for making this design decision is that SGD and ADMM have different communicated variables so it is challenging to add noises to these variables in a uniform way. Adding noises to the local model parameters can automatically ensure DP of the communicated variables, due to the post-processing property of DP [23].

4.3.1 Privacy Guarantee for Labels

In `TablePuppet`, clients need to add noises to the labels before sending them to the server. Specifically, as shown by Eq. 25, `TablePuppet` first adds Laplace noise with per-coordinate standard deviation λ to the labels y, to satisfy the DP guarantee, which results in a perturbed label vector. Then, we identify the class that retains the maximum value in the perturbed label vector as the new class label, with N_{class} representing the total number of classes:

$$\hat{y} \leftarrow \text{OneHot}(y) + \text{Laplace}(\lambda), \quad \hat{y} \leftarrow \arg \max_{j \in [N_{\text{class}}]} \hat{y}_j. \quad (25)$$

For continuous/numeric labels, $N_{\text{class}} = 1$.

In addition to the labels, clients can use cryptographic methods such as SHA-256 hashing to encrypt the *join_key*. The server then uses the encrypted *join_key* to perform joins and unions, assuming that hash collision is rare. Since the *row_id* normally does not contain sensitive information, clients can just send the raw *row_id* to the server. We leave how to use other cryptographic methods, such as *private set intersection* (PSI) [95,57,61], to perform table mapping as future work.

4.3.2 Privacy Guarantee for Features

To protect the communicated variables as well as each client's local data, we leverage DP-SGD (i.e., clipping and perturbing) [8] when updating each local model so that it satisfies (ϵ, δ)-DP. Since DP holds for any post-processing on top of the data, the communicated variables based on client's local model, such as model predictions, partial derivatives/gradients, and model parameters, also satisfy (ϵ, δ)-DP.

Specifically, when updating the local model of each client, we first clip per-sample gradient $\mathcal{G}_j$ with L^2-norm threshold C, and then add Gaussian noise sampled from $\mathcal{N}(0, \sigma^2 C^2)$ to the averaged batch gradient as

$$\mathcal{G} \leftarrow \frac{1}{B} \left(\sum_{j=1}^{B} \texttt{Clip}\left(\mathcal{G}_j, C\right) + \mathcal{N}\left(0, \sigma^2 C^2\right) \right). \quad (26)$$

Table 6 The computation and communication complexity comparison for each epoch. Here, "Comp. (Server)" represents the computation complexity on the server side, while "Comp. (Client)" refers to the computation complexity for each client c_i^q. "Comm. cost" denotes the communication cost between the server and all clients per epoch. B represents the *batch size*, and $\alpha_i \in [n_i, N]$.

Complexity	**VFL-SGD**	**VFL-ADMM**	**TP-Join-SGD**	**TP-Join-ADMM**	**TP-UCQ-SGD**	**TP-UCQ-ADMM**
Comp. (Server)	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N) + \mathcal{O}(MN/B)$	$\mathcal{O}(N) + \mathcal{O}(M\mathcal{T}')$
Comp. (Client)	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(\alpha_i)$	$\mathcal{O}(n_i)$	$\mathcal{O}(\alpha_i/\lvert Q_i\rvert)$	$\mathcal{O}(\mathcal{T}' n_i^q)$
Comm. rounds	$\mathcal{O}(N/B)$	$\mathcal{O}(1)$	$\mathcal{O}(N/B)$	$\mathcal{O}(1)$	$\mathcal{O}(N/B)$	$\mathcal{O}(\mathcal{T}')$
Comm. cost	$\mathcal{O}(MN)$	$\mathcal{O}(MN)$	$\mathcal{O}\left(\sum_{i=1}^M \alpha_i\right)$	$\mathcal{O}\left(\sum_{i=1}^M n_i\right)$	$\mathcal{O}\left(\sum_{i=1}^M \alpha_i\right) + \mathcal{O}(QN/B)$	$\mathcal{O}\left(\sum_{i=1}^M n_i + \mathcal{T}' Q\right)$

Here, $\mathcal{G}_j$ is the gradient of the j-th sample/tuple, and $\mathcal{G}$ is the averaged gradient over a batch. B refers to the batch size. For SGD, we perturb $\nabla_{\theta_i} F(T_i)$, by performing per-sample gradient `Clip` in Eq. 18 with $\mathcal{G}_j = \frac{\partial \ell}{\partial \theta_i} + \beta \frac{\partial \mathcal{R}_i}{\partial \theta_i}$ and adding noise $\mathcal{N}$. For ADMM, we directly perform the formula of $\mathcal{G}$ while computing the gradient of the θ_i^q-update optimization problem (Eq. 22).

4.4 Feature Transformations

In `TablePuppet`, feature preprocessing depends on how the data is partitioned across clients: for join-only (VFL-like) scenarios where each client holds a disjoint set of features, transformations such as binning, encoding, and scaling can be applied independently and thus locally in each client, following standard centralized ML methods. For UCQ scenarios where clients hold different tuples of the same features, we employ a server-coordinated federated preprocessing approach as follows:

Feature binning. Each client computes local histogram or quantile statistics for its data and sends encrypted summaries to the server. The server aggregates these to derive global bin boundaries, which are then broadcast back to all clients. Clients apply these unified thresholds to discretize their local data consistently.

One-hot encoding. Clients identify local categorical values and submit encrypted or hashed versions to the server. The server constructs a global category-to-index mapping (e.g., `"US": 0, "UK": 1`) by merging all reported values, and distributes this mapping to clients. Each client then performs one-hot encoding using this shared schema.

Feature scaling. Clients compute local summary statistics (e.g., mean, std, min, max) and securely send them to the server. The server computes global statistics (e.g., via weighted averaging based on sample counts), and returns the unified scaling parameters. Clients apply these globally consistent rules for normalization or standardization.

This preprocessing phase ensures that all clients operate on aligned feature spaces, enabling correct gradient computation and model convergence during distributed training.

5 Complexity Analysis

Table 6 summarizes the computation and communication complexity of the SGD/ADMM algorithms atop `TablePuppet`, in comparison with VFL methods (**VFL-SGD** and **VFL-ADMM**), which are hypothetically assumed to run on the vertical partitions of the fully joined table. For simplicity, we omit the complexity analysis of the table-mapping mechanism in Table 6. We also represent the computation/communication complexity in terms of the tuple number (e.g., N and n_i), since other parameters, such as feature dimension (e.g., d_i), are correlated with the tuple number.

5.1 Join-Only Scenario

In this scenario, each table is owned by one client without horizontal partitions. Below we analyze the complexity of **VFL-SGD**, **VFL-ADMM**, as well as our **TP-Join-SGD** and **TP-Join-ADMM**.

Computation complexity. The server-side computation complexity is $\mathcal{O}(N)$ per epoch for the four algorithms, since they all perform computation on the mapped model predictions with N tuples. The computation complexity of each client is $\mathcal{O}(N)$ for **VFL-SGD** and **VFL-ADMM**, since they compute on vertical partitions of the joined table. Our **TP-Join-SGD** and **TP-Join-ADMM** reduce the computation complexity to $\mathcal{O}(\alpha_i)$ and $\mathcal{O}(n_i)$, respectively, where $\alpha_i \in [n_i, N]$ (Section 3.1.3).

Communication complexity. **VFL-SGD** and **TP-Join-SGD** split the joined table into multiple batches with batch size of B and gather/scatter model predictions/gradients for every batch. Therefore, the number of communication rounds is N/B per epoch. The corresponding cost between the server and all clients is $\mathcal{O}(MN)$ per epoch for **VFL-SGD** and is reduced to $\mathcal{O}(\sum_{i=1}^M \alpha_i)$ for **TP-Join-SGD**. In contrast, **VFL-ADMM** and **TP-Join-ADMM** decouple the ML training problem into sub-problems and solve each sub-problem in each client. Thus, they only need to gather/scatter model predictions/variables *once per epoch* between the server and clients, introducing $\mathcal{O}(1)$ communication rounds. However, **VFL-ADMM** still suffers from $\mathcal{O}(MN)$ communication cost per epoch, while our **TP-Join-ADMM** reduces it to $O(\sum_{i=1}^{M} n_i)$ (Section 3.1.3).

5.2 Join-Union (UCQ) Scenario

Each table can have multiple horizontal tables and each client owns a horizontal table. Below we only analyze the complexity of our **TP-UCQ-SGD** and **TP-UCQ-ADMM**, as existing state-of-the-art approaches do not support this scenario.

Computation complexity. In the outer loop of the training process, the computation complexity of the server is $O(N)$, as it computes on the logical joined table. In the inner loop, the server acts as the coordinator of the clients to perform coordinator-side SGD/ADMM computation. For **TP-UCQ-SGD**, the coordinator-side computation complexity is $O(1)$, which is repeated N/B times per epoch, leading to computation complexity of $O(MN/B)$ for all the M tables. On average, the client-side computation complexity is $O\left(\alpha_i/|Q_i|\right)$. For **TP-UCQ-ADMM**, the coordinator-side computation complexity is also $O(1)$ but is repeated $\mathcal{T}'$ times per epoch, leading to $O\left(M\mathcal{T}'\right)$ for all the M tables. The corresponding client-side computation complexity is $O\left(\mathcal{T}' n_i^q\right)$.

Communication complexity. In the outer loop of the training process, the server has the same number of communication rounds and cost as that in the join-only scenario; in the inner loop, the server (coordinator) needs to gather/scatter variables from clients. For **TP-UCQ-SGD**, there is only one communication round for each inner loop, leading to $O\left(Q\right)$ communication cost between the server and the Q clients. Since there are N/B outer loops, the total communication cost of **TP-UCQ-SGD** is $O(\sum_{i=1}^{M} \alpha_i) + O\left(QN/B\right)$. For **TP-UCQ-ADMM**, the number of communication rounds is $\mathcal{T}'$ for each inner loop, leading to $O\left(\mathcal{T}'Q\right)$ communication cost between the server and the Q clients. Since there is only one outer loop per epoch, its total communication cost is $O\left(\sum_{i=1}^{M} n_i + \mathcal{T}'Q\right)$.

6 Evaluation

We study the effectiveness and efficiency of `TablePuppet`, by evaluating model accuracy and performance of SGD/ADMM implemented atop `TablePuppet` and comparing against state-of-the-art approaches in diverse scenarios. These scenarios include join-only/join-union, non-DP/DP, as well as different ML models on a variety of datasets. Below we summarize our evaluation methodology and main results:

(1) Effectiveness (model accuracy). We regard directly training centralized ML models on the fully joined table as baselines, and compare SGD/ADMM atop `TablePuppet` with these baselines in terms of model accuracy. The experiments show that SGD/ADMM atop `TablePuppet` can achieve comparable model accuracy to the baselines in both join-only and join-union scenarios. Additionally, our SGD/ADMM algorithms converge faster than existing GD-based algorithms employed in factorized approaches. With privacy guarantees, SGD/ADMM atop `TablePuppet` suffers from up to 4.5% lower model accuracy than the baselines due to the noises injected into data labels and model training.

Table 7 The datasets and ML models used in the evaluation, where 15% of each dataset is used for model testing.

Dataset (ML Model)	#Table	#Tuple	#Feature
MIMIC-III (NN)	5	[35K, 2.9M]	[2, 717]
Yelp (BERT-Softmax)	3	[35K, 3.2M]	[3, 775]
MovieLens-1M (Linear)	3	[6K, 0.9M]	[4, 52]
MovieLens-1M (NN)	3	[6K, 0.9M]	[4, 52]

(2) Efficiency (communication/computation/training time). Since communication is the primary bottleneck in distributed environments [71,60], our evaluation mainly focuses on the trade-off between model accuracy and communication time for the SGD/ADMM algorithms atop `TablePuppet`. We also report the model accuracy vs. computation/training time. Compared to the state-of-the-art VFL algorithms (i.e., **VFL-SGD/ADMM**) in the join-only scenario, our SGD/ADMM algorithms require less communication/computation/training time to converge while achieving similar accuracy. In the join-union scenario, we evaluate the efficiency of our SGD/ADMM algorithms. We observe that ADMM is more communication-efficient than SGD when implemented on top of `TablePuppet`, converging to similar accuracy in less time.

6.1 Experimental Setup

6.1.1 Datasets and Models

We train four linear/NN (Neural Network) ML models on three real-world datasets for both classification and regression tasks, as summarized in Table 7.

(1) MIMIC-III: This is a healthcare dataset with 46K patients admitted to ICUs at the BIDMC between 2001 and 2012 [3]. We leverage the scripts in MIMIC-III Benchmarks [39,4] to extract five tables `Patients`, `Admissions`, `Stays`, `Diagnoses`, and `Events`. We perform the decompensation prediction task, which uses a *neural network* (NN) model with 10 hidden layers of 16 neurons to predict whether the patient's health will rapidly deteriorate in the next 24 hours (with 0/1 label).

(2) Yelp: This Yelp dataset contains three core tables `business`, `review`, and `user` [5]. The label column is `stars` in the `review` table, denoting the user ratings of 1 to 5 for businesses. To be simple, we only use the tuples of restaurants in the `business` table, and we obtain 3.2M reviews and

Table 8 Latency and bandwidth among our AWS cloud instances

Regions	Latency	Bandwidth
US-East ↔ US-West	63 ms	60.0 MB/s
US-West ↔ EU	153 ms	26.2 MB/s
US-East ↔ EU	110 ms	33.8 MB/s

ratings from 1.2M users. We first use BERT [21] to extract 768-dimensional embeddings from review text and then use *softmax regression* for classification.

(3) MovieLens-1M: This dataset contains 0.9M user ratings on about 4K movies given by 6K users [1], with three tables `movies`, `ratings`, and `users`. The label column is `rating`, which denotes the user ratings of 1 to 5. We use both *linear regression* (Linear) and *neural network* (NN) model with a hidden layer of 16 neurons to predict the review score. If a movie has multiple genres, there are multiple tuples for this movie in the `movies` table.

All table schemas are available at our code repository.[1] In the join-only scenario, each client owns a whole table. In the join-union scenario, each table is further partitioned into two horizontal shards, i.e., two clients for each table.

6.1.2 Experimental Settings

To analyze the performance of `TablePuppet` in a real-world cloud environment, we implemented the framework on top of Ray and evaluated it using real-world AWS cloud instances in different regions, including US-East (Virginia), US-West (Oregon), and EU (Stockholm). We launched one g5.8xlarge instance in each region and distributed the tables among these three instances. Each g5.8xlarge instance contains 32 vCPUs, 128GB RAM, and 1 NVIDIA A10G Tensor Core GPU. The measured latency and bandwidth between these regions are shown in Table 8.

6.1.3 Hyperparameters

We use grid search to tune hyperparameters, including the learning rate η for SGD and the penalty parameter ρ for ADMM. For each dataset, we run 10 epochs for SGD/ADMM. We vary the η of SGD from 0.01 to 0.5, and vary the ρ of ADMM from 0.1 to 2. We set the batch size for SGD as 10K, as small batch size can lead to many communication rounds. In DP scenario, for model training with DP-SGD, we set the (ϵ, δ)-DP by using $\epsilon = 1$ and $\delta = 1e\text{-}5$ [8] with clipping threshold $C = 1$. For label DP, we set the Laplace noise $\lambda = 0.5$, which results in $\epsilon = 5.6$. This label DP level is consistent with existing work that typically sets ϵ between 3 and 8 [59,31]. We leverage PyTorch's DP tool, Opacus [90], to implement the DP and calculate the privacy budget.

[1] Our code is available at https://github.com/JerryLead/TablePuppet.

6.2 Effectiveness of `TablePuppet`

6.2.1 Baselines

As LRT is to train ML models on the UCQ result, we regard directly training ML models on the fully joined table as baselines. To obtain the baseline accuracy, we use SGD to train ML models on the joined table without privacy guarantees (shown as *centralized* in Figures 5 and 6). The model accuracy refers to test accuracy (higher is better) for classification models and test root mean square error (RMSE, lower is better) for regression models. We compare the model accuracy of SGD/ADMM atop `TablePuppet` with the baselines and state-of-the-art approaches, including GD-based algorithms and VFL algorithms (**VFL-SGD/ADMM**) that are forced to run directly on the vertical partitions of the joined table.

For the join-only scenario, we adopt two representative VFL algorithms as baselines. **(1) VFL-SGD**: This follows the method used in FDML [42] and VAFL [17], two representative VFL frameworks [56]. **(2) VFL-ADMM**: This follows the method proposed by VIMADMM [83], another representative VFL framework [56].

6.2.2 Results without Privacy Guarantee

Figure 5 shows the convergence rates of different SGD/ADMM algorithms without privacy guarantees. In this non-DP scenario, all algorithms atop `TablePuppet` can converge to model accuracy comparable to the baselines, which demonstrates the effectiveness of `TablePuppet`. Our SGD/ADMM algorithms also converge faster than GD-based algorithms used by factorized approaches for ML over joins.

For SGD algorithms, the convergence curve of **TP-UCQ-SGD** is similar to that of **TP-Join-SGD**, because they share the same gradient computation—the computation results of Eq. 17 and 18 of **TP-UCQ-SGD** are the same as Eq. 12 of **TP-Join-SGD**. Moreover, both of them exhibit comparable convergence rate to **VFL-SGD**, showcasing the effectiveness of our computational/communication optimization on SGD. Compared to ADMM, SGD converges faster in most cases, because SGD updates the local models much more frequently than ADMM in each epoch. For example, on the *Yelp* dataset, SGD updates local models 320 times (i.e., the number of batches) per epoch, whereas ADMM only updates the local model once per epoch (by completely solving the local optimization problem in each client). However, with more model updates, SGD requires more communication rounds than ADMM, leading to longer communication time and thus longer training time per epoch (Section 6.3).

Regarding ADMM algorithms, **TP-Join-ADMM** achieves a similar convergence rate to **VFL-ADMM**. This reveals that the computation/communication optimization used by **TP-Join-ADMM** is effective and does not affect the convergence

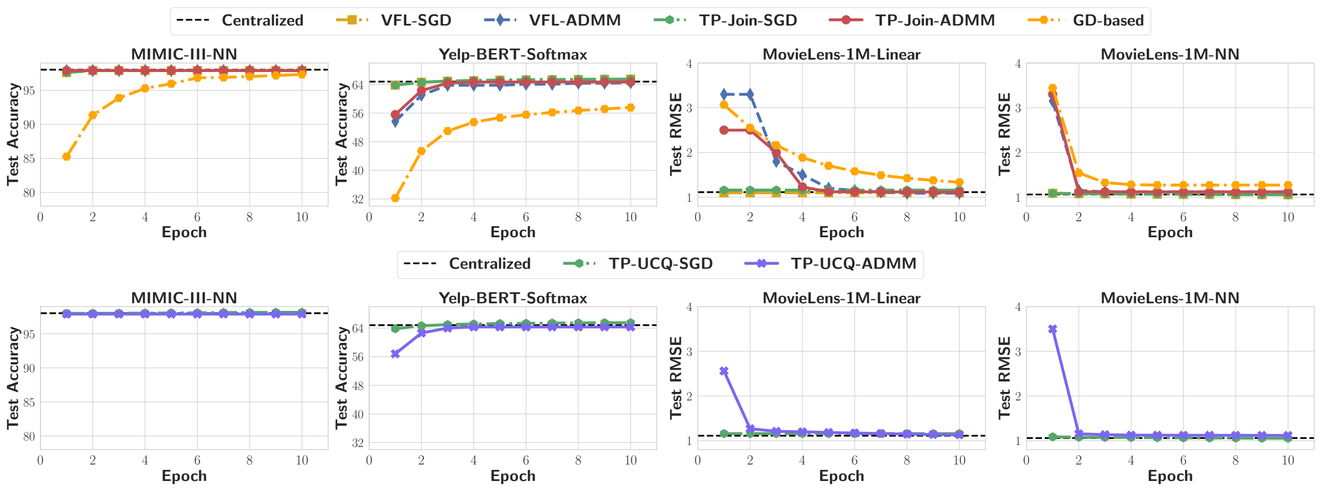


Fig. 5 The convergence rates of different algorithms for join-only and join-union scenarios without privacy guarantees.

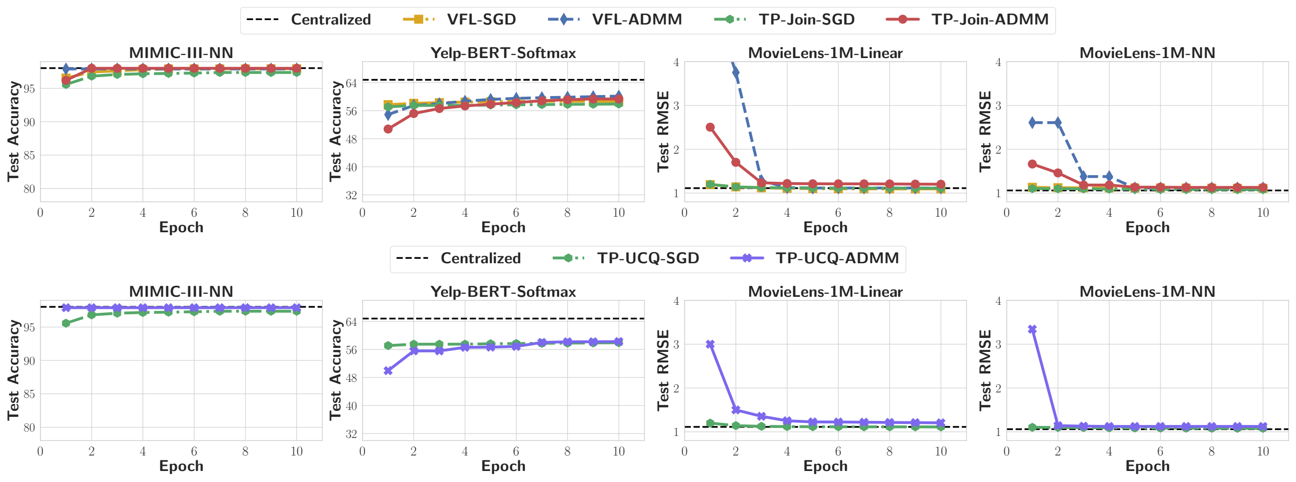


Fig. 6 The convergence rates of different algorithms for join-only and join-union scenarios with privacy guarantees. Here, centralized refers to the non-DP baselines for measuring the accuracy gap between non-DP and DP results.

of ADMM. Moreover, **TP-UCQ-ADMM** also demonstrated comparable model accuracy to **TP-Join-ADMM**, indicating that our paradigm of learning over unions is also effective.

6.2.3 Results with Privacy Guarantee

By introducing DP to both labels and model training in `TablePuppet`, the model accuracy of SGD/ADMM drops compared to the non-DP centralized baselines (Figure 6). Taking **TP-Join-SGD** for example, the test accuracy drops by 7.2 points for Yelp-BERT-Softmax, while the test RMSE slightly increases by 0.05 for MovieLens-Linear and 0.02 for MovieLens-NN. However, while model accuracy drops, these algorithms gain on privacy protection against feature and label leakages. In this DP scenario, we can still observe that all algorithms atop `TablePuppet` can converge to similar model accuracy. Specifically, SGD algorithms exhibit the fastest convergence in most cases due to the largest number of local model updates per epoch.

6.3 Efficiency of `TablePuppet`

6.3.1 Baselines

We consider VFL algorithms (**VFL-SGD** and **VFL-ADMM**) as the baselines, because they are well-suited for distributed environments and converge faster than GD-based algorithms. We first compare our SGD/ADMM algorithms (**TP-Join-SGD** and **TP-Join-ADMM**) with the baselines in the join-only scenario. We next evaluate our SGD/ADMM algorithms (**TP-UCQ-SGD** and **TP-UCQ-ADMM**) in the join-union scenario, since only `TablePuppet` supports this use case. The efficiency is measured using *model accuracy vs. communication/computation/training time*. The server is located in US-East, and clients are distributed across US-West, US-East, and EU. We next report experimental results for the non-DP scenario, as we observe similar results in the DP setting.

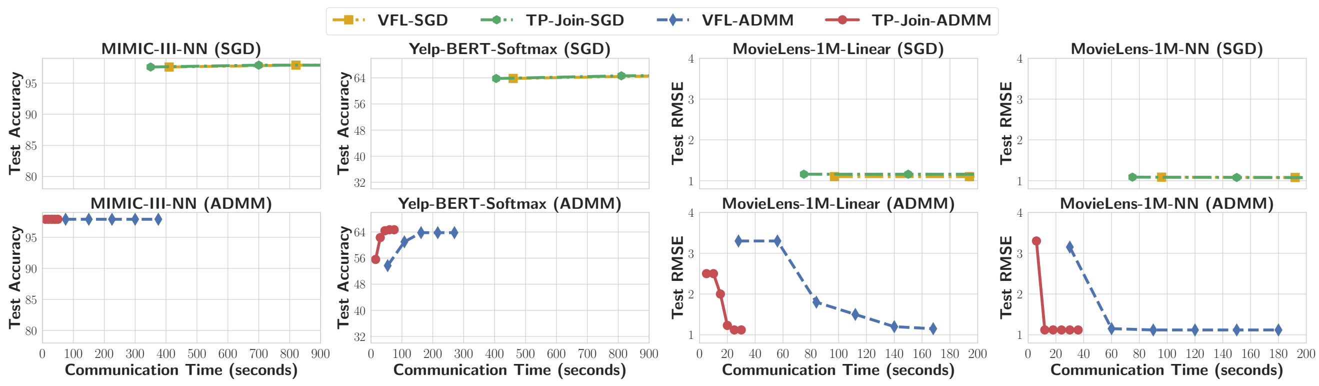


Fig. 7 The model accuracy vs. communication time for join-only scenario.

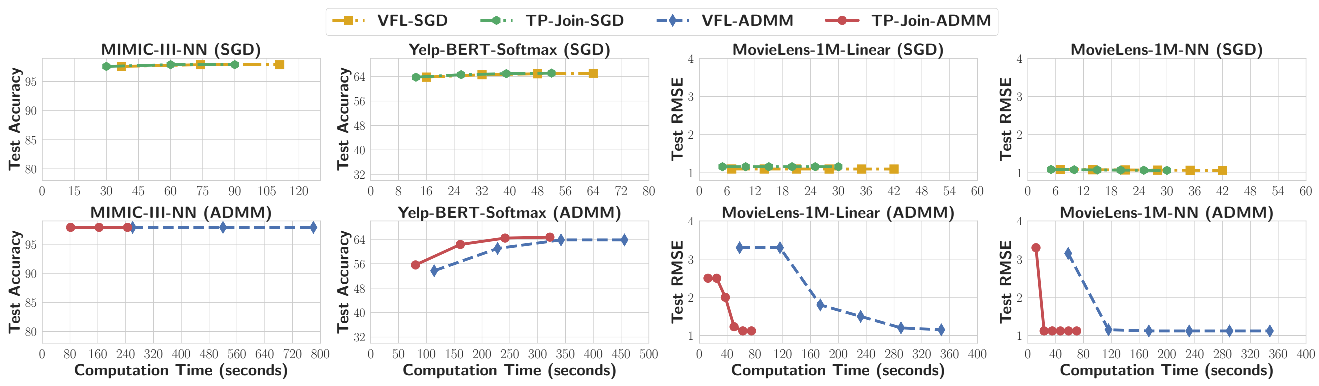


Fig. 8 The model accuracy vs. computation time for join-only scenario.

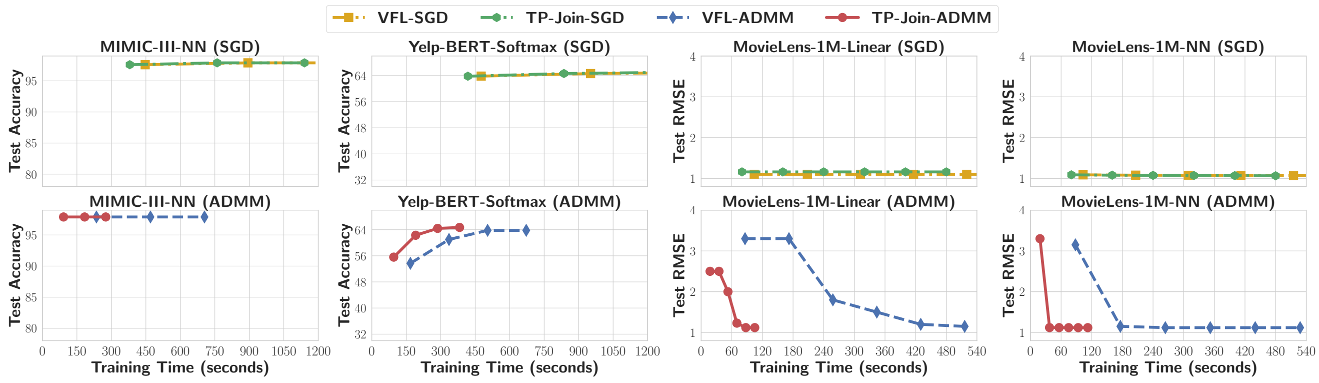


Fig. 9 The model accuracy vs. training time for join-only scenario.

6.3.2 Results of Join-Only Scenario

Figure 7, 8, and 9 illustrate the *model accuracy vs. communication/computation/training time*, respectively. The training time refers to the per-epoch training time, including both communication and computation time. Each point in the figure represents the test accuracy/RMSE after one epoch. Note that **VFL-SGD** and **TP-Join-SGD** results are not fully plotted in the figure, due to the long communication/training time caused by too many communication rounds per epoch. For example, for the *MIMIC-III* dataset with 2.9M tuples, **VFL-SGD** suffers from 290 communication rounds per epoch, leading to extremely long communication time (>1,000 seconds for just three epochs) that exceeds the boundary of the horizontal axis range. The high communication time arises not only from data volume but also from the multiplicative effect of network latency and the number of rounds: even small per-round latency accumulates significantly when multiplied by hundreds of rounds. Our **TP-Join-SGD** requires less communication time than **VFL-SGD**, but still longer time than ADMM algorithms due to more communication rounds. In more detail, **VFL-ADMM** requires ∼75s communication time per epoch, while our **TP-Join-ADMM** only requires 10s. As another example of MovieLens-1M, our

TP-Join-ADMM converges with 4-5× less communication time than **VFL-ADMM**, owing to the communication reduction on duplicate tuples as described in Section 3.1.3. The above results are consistent with the complexity analysis in Section 5, where **VFL-ADMM** outperforms **VFL-SGD** due to fewer communication rounds and our **TP-Join-ADMM** further outperforms **VFL-ADMM** due to less communication cost.

Similar trends are observed in *model accuracy vs. computation/training time*, which demonstrates that our optimizations in `TablePuppet` are effective not only in reducing communication cost but also in improving computational efficiency. Despite having shorter per-epoch computation time than ADMM, SGD-based methods exhibit longer per-epoch training time overall, because the high communication rounds and associated latency dominate the total time. Thus, even with faster local computation, the communication bottleneck prevents SGD from converging efficiently.

6.3.3 Results of Join-Union Scenario

Figure 10 presents the *model accuracy vs. communication time* and Figure 11 presents the *model accuracy vs. training time*. Although **TP-UCQ-SGD** exhibits a faster convergence rate in terms of the number of epochs (Section 6.2), **TP-UCQ-ADMM** still takes less communication/training time than **TP-UCQ-SGD** to converge. The reason is similar to that of the join-only scenario, i.e., **TP-UCQ-SGD** requires more communication rounds due to per-batch communication.

In comparison with the results in the join-only scenario, SGD/ADMM algorithms for the join-union scenario require more communication time per epoch to communicate inner-loop computation results from LoU, such as the partial gradients of SGD and the auxiliary variables of ADMM. Although for LoJ the size of the inner-loop computation results is normally smaller than that of the outer-loop computation results, the extra inner-loop computation increases both the number of communication rounds and the communication cost per epoch. The corresponding complexity has been analyzed in Section 5, where **TP-UCQ-SGD** and **TP-UCQ-ADMM** require extra $O(QN/B)$ and $O(\mathcal{T}'Q)$ communication cost in the inner loop. In summary, **TP-Join-ADMM** and **TP-UCQ-ADMM** outperform the counterparts in terms of per-epoch communication/training time.

7 Related Work

ML over Joins. Given the central role of relational data in enterprise data management, ML over joins has been extensively studied for years in the database community [76, 75, 50, 16]. The primary solution has centered around factorized approaches, namely *factorized learning* or *learning over factorized joins* [74, 75, 50, 16]. However, as discussed in Section 2.3.1, these approaches have three limitations. First, these methods primarily focus on joins in a single database and overlook the communication overhead in distributed environments. The LRT problem extends the scope of ML over joins to accommodate distributed settings. Second, factorized approaches are typically restricted to specific models, such as generalized linear models or matrix factorization, and lack support for non-linear models such as deep neural networks. In contrast, our work is not limited to specific model types. Finally, most existing approaches rely on gradient descent (GD), while our framework supports (mini-batch) SGD and ADMM, which converge faster than GD and are well-suited for distributed environments.

Federated Learning (FL). FL is a privacy-preserving technique for collaboratively training ML models across decentralized clients. Based on different types of data partitioning, existing surveys [88, 56] classify FL into horizontal FL, vertical FL, and federated transfer learning (FTL). Section 2.3.2 discussed horizontal and vertical FL, while FTL focuses on model transfer between clients using overlapping data samples and features [54, 70, 18]. A recent addition is hybrid FL [93, 65] where data can be partitioned *both* horizontally and vertically; however, existing hybrid FL work either requires overlapping samples/features like FTL [93] or does not support non-convex models like neural network [65]. Moreover, current FL solutions assume that the decentralized data tables can be simply one-to-one aligned, while the LRT problem studied in this paper targets (distributed) relational tables that require joins and unions. FL traditionally employs one specific training algorithm such as SGD, ADMM, *block coordinate descent* (BCD) [55], or *gradient-boosted decision trees* (GBDT) for tree-based models [28, 82, 37]. Our `TablePuppet` currently supports both SGD and ADMM, and we plan to extend it for more types of training algorithms in future work. In Appendix C, we detail the comparison with tree-based models and further discuss the potential extensions of `TablePuppet`.

While `TablePuppet` can handle joins and unions effectively, addressing complex scenarios, such as evolving data distributions or schema heterogeneity, could benefit from insights in existing FL work. For example, FedNova [81], SCAFFOLD [44], and pFedMe [22] propose strategies to address heterogeneity in horizontal FL by normalizing updates, correcting client-drift, or personalizing models, offering promising directions for future optimization. Additionally, to improve communication efficiency, we can also explore how to leverage asynchronous or other communication-efficient model update protocols from existing FL research [27, 84, 53, 36, 58].

Privacy Guarantee. Unlike traditional HFL/VFL, where only features or labels are distributed across clients, in LRT both features and labels are distributed, requiring more advanced differential privacy (DP) methods to protect them

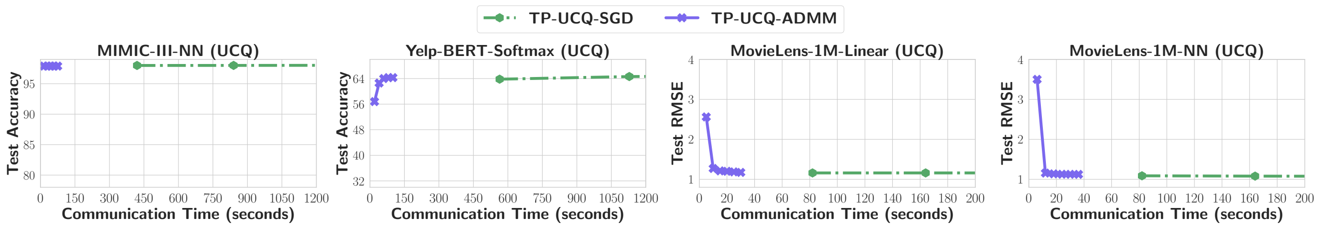


Fig. 10 The model accuracy vs. communication time for join-union scenario.

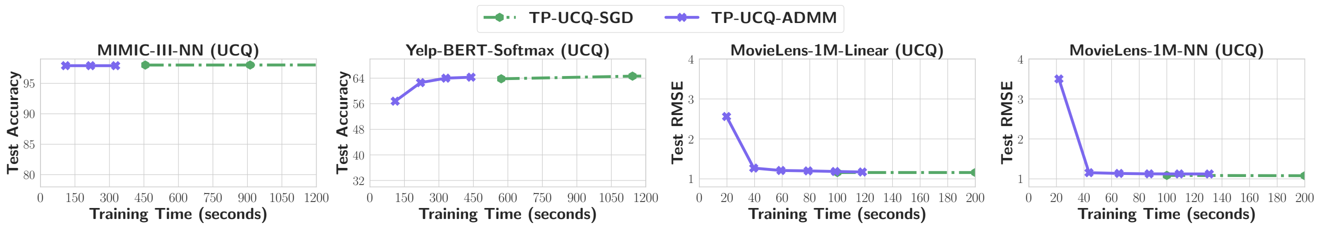


Fig. 11 The model accuracy vs. training time for join-union scenario.

simultaneously. To address this, `TablePuppet` currently uses separate label-level DP and DP-SGD to flexibly safeguard features and labels to accommodate different scenarios as described in Section 4.3. It remains interesting to incorporate other advanced label-level DP mechanisms [59, 31], such as post-processing the model predictions through Bayesian inference, into `TablePuppet`. Moreover, cryptographic techniques like homomorphic encryption [69, 32] and secure multiparty computation [10, 11] can also be potentially integrated into `TablePuppet` to enhance its privacy guarantee, albeit with increased computation overhead.

Relation Learning. Several works have explored learning rules or relations from relational data, rather than training traditional machine learning models. QuickFOIL [91] learns first-order rules (e.g., Datalog rules) through an optimized inductive logic programming (ILP) approach. It also leverages in-RDBMS computation to minimize join overhead and scale to large datasets. Schema Independent Relation Learning [66] introduces Castor, a framework that ensures consistent rule-learning outcomes across schema transformations by utilizing data dependencies. While these systems focus on rule induction, they address challenges in handling complex joins and schema variations. Castor's schema independence approach provides insights into building systems resilient to schema changes, while QuickFOIL's efficient join handling highlights the benefits of pushing computation closer to the data source. These strategies could inspire `TablePuppet` to further enhance its robustness to dynamic schemas and optimize join computations.

Data Discovery Approaches. Data discovery systems [72, 26, 96, 33, 86, 14, 24, 63] aim to identify and integrate relevant datasets from large repositories, often through join or union paths, to enrich a base dataset for downstream analytical tasks such as machine learning. Among these, goal-oriented approaches like METAM [29] enables more targeted and efficient data composition by explicitly modeling the utility of each potential augmentation with respect to a specific analytical objective. While `TablePuppet` focuses on optimizing ML training over a given relational schema, we see data discovery techniques as a promising direction for future integration. In complex environments such as data lakes, where tables and relationships are numerous and heterogeneous, data discovery methods can serve as an intelligent upstream module to automatically recommend relevant tables and join/union paths that form the underlying Union of Conjunctive Queries (UCQ). This would allow `TablePuppet` to operate on high-quality, task-specific data compositions without requiring users to manually specify the schema. We believe combining these two lines of work could enable a holistic pipeline from intelligent data identification to optimized and privacy-preserving model training, extending the applicability of `TablePuppet` to real-world and large-scale relational settings.

8 Discussion and Future Work

Join and Union Schemas. Although our experiments primarily utilize the common star schema, `TablePuppet` is designed to be agnostic to other join schemas, including chain and snowflake schemas. Regardless of the schema, the UCQ result is represented as a large joined table X, composed of vertical tables X_i. `TablePuppet` employs a table-mapping mechanism to construct this virtual joined table and efficiently push ML computation down to the individual vertical tables. While different schemas may influence the number of duplicate tuples in each table, they do not affect the framework's functionality, such as processes of LoJ and LoU. For union schemas, we assume that all horizontal tables share the same schema, a typical scenario in distributed environments (e.g., across cloud instances or regions). Handling horizontal tables with different schemas (e.g., tables from different banks) presents a more complex scenario and is left for future work. To address this problem in the current framework, these tables can be combined under an "umbrella" schema that includes all columns, resulting in sparse features for ML models. While feasible, this approach may impact model accuracy, highlighting the need for more advanced solutions in future research.

Non-IID Data Distributions. One issue with joining multiple tables is that some entities (e.g. orders) may appear multiple times in the joined table, causing them to be overrepresented and potentially introducing bias into the model. This problem is well recognized in statistical relational learning [30,67]. In our settings, a simple way to address this is to reduce the weights of tuples linked to frequently occurring entities. Another option is to reduce their influence during training by dividing their gradients by the number of times they appear (e.g., $|G_i(j)|$). Whether such adjustments are necessary depends on the situation. For instance, if an entity appears frequently because it is genuinely more important (e.g., a high-value customer with many orders), it makes sense for the model to give it more weight. However, in cases where frequent repetition skews the results, these adjustments can help reduce bias. Although `TablePuppet` has the flexibility to apply these techniques during the LoJ or LoU stages, a thorough evaluation of their effectiveness and integration with advanced relational learning methods remains an area of future work.

Full-Join ML vs. Non-Join ML. While recent work such as Kumar et al. [51] identify specific scenarios where joins can be "safely avoided" for efficiency, namely "non-join ML", they also acknowledge that avoiding joins is not always safe and can lead to a decrease in model accuracy. Moreover, non-join ML only applies to key-foreign-key joins and cannot handle more complex patterns such as many-to-many joins. For a generic framework such as `TablePuppet`, it is essential to support all scenarios where the full set of features from the joined tables is indispensable for optimal model accuracy. Furthermore, complex models like Neural Networks benefit significantly from rich, high-dimensional feature spaces (more features from joined tables), which may not be adequately captured by simple foreign key embeddings used in [51]. Additionally, metrics such as Risk of Representation (ROR) and tuple ratios (TR) used to determine whether joins can be safely avoided in [51], require global knowledge of the joined tables. In distributed and federated learning settings, obtaining this global knowledge is often infeasible or incurs prohibitive communication and computational costs due to privacy constraints and data distribution issues. Despite these challenges, investigating how to incorporate non-join ML concepts into `TablePuppet` offers a promising direction for future work. Specifically, integrating rule-based strategies into the table-mapping stage could enable intelligent decisions about which tables truly require joining, potentially reducing computation/communication overhead in certain scenarios.

Join with Sub-queries. In addition to join between two base tables, `TablePuppet` also supports join of a base table with an aggregated sub-query, as clients can locally aggregate data before sending the join keys for table mapping. We leave the exploration of join with other types of sub-queries [46] as one interesting direction for future work.

Sampling-Based Training. In large-scale data analytics, sampling over joins has become an essential technique for efficiently handling massive datasets. Rather than computing the full result of the join, sampling methods create a representative subset of the data, which is particularly useful for analytical queries. Techniques such as uniform sampling [15, 94], reservoir sampling [20], and weighted sampling [78] have shown significant efficiency improvements, especially for acyclic and cyclic joins, by leveraging advanced indexing and dynamic updates to simplify the sampling process. Although `TablePuppet` currently trains ML models directly on the full UCQ result to ensure accuracy, exploring sampling-based methods in the future can speed up training for large-scale datasets. This can achieve a balance between higher performance and maintaining model accuracy.

Performance Bottleneck. `TablePuppet` adopts a client-server architecture, where the server may become a performance bottleneck under heavy workloads. To address this concern, we outline several practical strategies to mitigate server-side overhead. First, we could explore adopting closed-form or second-order methods for ADMM's z-update to reduce server-side computational overhead. Second, we can leverage multi-threading technique to enable parallel aggregation and distribution of communicated variables. Third, in multi-user scenarios, deploying multiple server instances for independent tasks would effectively distribute the load. Additionally, as mentioned in related work, sampling techniques can help reduce both communication and computation costs. These strategies collectively represent promising directions to alleviate performance bottlenecks in practical deployments.

Fine-Tuning Approaches. We provide practical guidelines for configuring and tuning hyperparameters in `TablePuppet` to balance convergence rate, communication cost, and computational efficiency. For SGD, the number of communication rounds per epoch is equal to the ratio of the joined table size to the batch size. Thus, a larger batch size reduces communication rounds/time but may slow down the convergence. We recommend using a medium batch size such as 1/100 of the joined table size, as that in the VFL framework of VAFL [17]. In future work, we can explore dynamically adjusting the batch size based on runtime communication time to better balance the convergence rate and communication time. For

ADMM, the communication per epoch is fixed at one round in the join-only scenario, so tuning for communication is unnecessary. In the join-union scenario, ADMM involves inner rounds of communication among clients holding horizontal partitions of the same table. We set the maximum of the inner rounds to 10, as ADMM usually converges within 10 epochs, also verified in the VFL framework of VIMADMM [83]. However, users can lower this value to reduce communication time if the trained model is simple.

9 Conclusion

In this paper, we formalized the problem of learning over relational tables (LRT) as learning over unions of conjunctive queries (UCQ). Our formulation generalizes the problem of ML over joins and federated learning. To address the LRT problem, we presented `TablePuppet`, a two-step decomposition framework that can push two widely-used ML training algorithms, SGD and ADMM, down to individual relational tables, with both performance optimization and privacy guarantees. Our implementation of `TablePuppet` leverages a server-client architecture with three abstracted high-level operators, unifying the design and implementation of SGD and ADMM algorithms. The SGD/ADMM algorithms implemented atop `TablePuppet` can achieve model accuracy comparable to the strongest centralized baselines while outperforming existing state-of-the-art methods in terms of communication/training time.

References

1. MovieLens 1M dataset. `https://grouplens.org/datasets/movielens/1m/` (2003)
2. General Data Protection Regulation (GDPR). `https://gdpr-info.eu/` (2016)
3. MIMIC-III Clinical Database. `https://physionet.org/content/mimiciii/1.4/` (2016)
4. MIMIC-III Benchmarks. `https://github.com/YerevaNN/mimic3-benchmarks` (2023)
5. Yelp Open Dataset: An all-purpose dataset for learning. `https://www.yelp.com/dataset` (2023)
6. DB-Engines. Ranking scores per category in percent. `https://db-engines.com/en/ranking_categories?ref=timescale-blog` (2024)
7. Flower: A Friendly Federated AI Framework. `https://github.com/adap/flower` (2025)
8. Abadi, M., Chu, A., Goodfellow, I., McMahan, H.B., Mironov, I., Talwar, K., Zhang, L.: Deep learning with differential privacy. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 308–318 (2016)
9. Arnold, J., Glavic, B., Raicu, I.: A high-performance distributed relational database system for scalable OLAP processing. In: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019, pp. 738–748. IEEE (2019)
10. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Proceedings of the twentieth annual ACM symposium on Theory of computing, pp. 1–10 (1988)
11. Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H.B., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for privacy-preserving machine learning. In: proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1175–1191 (2017)
12. Boyd, S.P., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Found. Trends Mach. Learn. **3**(1), 1–122 (2011)
13. Bruno, N., Kwon, Y., Wu, M.: Advanced join strategies for large-scale distributed computation. Proc. VLDB Endow. **7**(13), 1484–1495 (2014)
14. Castelo, S., Rampin, R., Santos, A.S.R., Bessa, A., Chirigati, F., Freire, J.: Auctus: A dataset search engine for data discovery and augmentation. Proc. VLDB Endow. **14**(12), 2791–2794 (2021)
15. Chaudhuri, S., Motwani, R., Narasayya, V.R.: On random sampling over joins. In: SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA, pp. 263–274. ACM Press (1999)
16. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)
17. Chen, T., Jin, X., Sun, Y., Yin, W.: Vafl: A method of vertical asynchronous federated learning. arXiv preprint arXiv:2007.06081 (2020)
18. Chen, Y., Qin, X., Wang, J., Yu, C., Gao, W.: Fedhealth: A federated transfer learning framework for wearable healthcare. IEEE Intelligent Systems **35**(4), 83–93 (2020)
19. Cheng, K., Fan, T., Jin, Y., Liu, Y., Chen, T., Papadopoulos, D., Yang, Q.: Secureboost: A lossless federated learning framework. IEEE Intelligent Systems **36**(6), 87–98 (2021)
20. Dai, B., Hu, X., Yi, K.: Reservoir sampling over joins. Proc. ACM Manag. Data **2**(3), 118 (2024)
21. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, pp. 4171–4186 (2019)
22. Dinh, C.T., Tran, N.H., Nguyen, T.D.: Personalized federated learning with moreau envelopes. In: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
23. Dwork, C., Roth, A., et al.: The algorithmic foundations of differential privacy, vol. 9. Now Publishers, Inc. (2014)
24. Esmailoghli, M., Quiané-Ruiz, J., Abedjan, Z.: COCOA: correlation coefficient-aware data augmentation. In: Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021, pp. 331–336. OpenProceedings.org (2021)
25. Feng, S., Yu, H.: Multi-participant multi-class vertical federated learning. arXiv preprint arXiv:2001.11154 (2020)
26. Fernandez, R.C., Min, J., Nava, D., Madden, S.: Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, pp. 1190–1201. IEEE (2019)
27. Fu, F., Miao, X., Jiang, J., Xue, H., Cui, B.: Towards communication-efficient vertical federated learning training via cache-enabled local update. Proc. VLDB Endow. **15**(10), 2111–2120 (2022)
28. Fu, F., Shao, Y., Yu, L., Jiang, J., Xue, H., Tao, Y., Cui, B.: Vf^{2}boost: Very fast vertical federated gradient boosting for cross-enterprise learning. In: SIGMOD '21: International Conference on Management of Data, 2021, pp. 563–576. ACM (2021)

29. Galhotra, S., Gong, Y., Fernandez, R.C.: Metam: Goal-oriented data discovery. In: 39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023, pp. 2780–2793. IEEE (2023)
30. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). The MIT Press (2007)
31. Ghazi, B., Golowich, N., Kumar, R., Manurangsi, P., Zhang, C.: Deep learning with label differential privacy. Advances in neural information processing systems **34**, 27,131–27,145 (2021)
32. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning, pp. 201–210. PMLR (2016)
33. Gong, Y., Zhu, Z., Galhotra, S., Fernandez, R.C.: Ver: View discovery in the wild. In: 39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023, pp. 503–516. IEEE (2023)
34. Grinsztajn, L., Oyallon, E., Varoquaux, G.: Why do tree-based models still outperform deep learning on typical tabular data? In: Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 (2022)
35. Gu, B., Dang, Z., Li, X., Huang, H.: Federated doubly stochastic kernel learning for vertically partitioned data. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2483–2493 (2020)
36. Gu, B., Xu, A., Huo, Z., Deng, C., Huang, H.: Privacy-preserving asynchronous vertical federated learning algorithms for multiparty collaborative learning. IEEE Trans. Neural Networks Learn. Syst. **33**(11), 6103–6115 (2022)
37. Han, Y., Du, P., Yang, K.: Fedgbf: An efficient vertical federated learning framework via gradient boosting and bagging. CoRR **abs/2204.00976** (2022). DOI 10.48550/arXiv.2204.00976. URL `https://doi.org/10.48550/arXiv.2204.00976`
38. Hardy, S., Henecka, W., Ivey-Law, H., Nock, R., Patrini, G., Smith, G., Thorne, B.: Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. arXiv preprint arXiv:1711.10677 (2017)
39. Harutyunyan, H., Khachatrian, H., Kale, D.C., Ver Steeg, G., Galstyan, A.: Multitask learning and benchmarking with clinical time series data. Scientific Data **6**(1), 96 (2019)
40. Hong, M., Luo, Z.Q., Razaviyayn, M.: Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. SIAM Journal on Optimization **26**(1), 337–364 (2016)
41. Hu, Y., Liu, P., Kong, L., Niu, D.: Learning privately over distributed features: An admm sharing approach. arXiv preprint arXiv:1907.07735 (2019)
42. Hu, Y., Niu, D., Yang, J., Zhou, S.: Fdml: A collaborative machine learning framework for distributed features. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2232–2240 (2019)
43. Jin, X., Chen, P.Y., Hsu, C.Y., Yu, C.M., Chen, T.: Catastrophic data leakage in vertical federated learning. Advances in Neural Information Processing Systems **34** (2021)
44. Karimireddy, S.P., Kale, S., Mohri, M., Reddi, S.J., Stich, S.U., Suresh, A.T.: SCAFFOLD: stochastic controlled averaging for federated learning. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, Proceedings of Machine Learning Research, vol. 119, pp. 5132–5143. PMLR (2020)
45. Khan, A., ten Thij, M., Wilbik, A.: Vertical federated learning: A structured literature review. CoRR **abs/2212.00622** (2022). URL `https://doi.org/10.48550/arXiv.2212.00622`
46. Kim, W.: On optimizing an sql-like nested query. ACM Trans. Database Syst. **7**(3), 443–469 (1982)
47. Konečný, J., McMahan, H.B., Ramage, D., Richtárik, P.: Federated optimization: Distributed machine learning for on-device intelligence. CoRR **abs/1610.02527** (2016). URL `http://arxiv.org/abs/1610.02527`
48. Konečný, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated learning: Strategies for improving communication efficiency. CoRR **abs/1610.05492** (2016). URL `http://arxiv.org/abs/1610.05492`
49. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)
50. Kumar, A., Naughton, J.F., Patel, J.M.: Learning generalized linear models over normalized data. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1969–1984. ACM (2015)
51. Kumar, A., Naughton, J.F., Patel, J.M., Zhu, X.: To join or not to join?: Thinking twice about joins before feature selection. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pp. 19–34. ACM (2016)
52. Li, Q., Diao, Y., Chen, Q., He, B.: Federated learning on non-iid data silos: An experimental study. In: 38th IEEE International Conference on Data Engineering, ICDE 2022, pp. 965–978. IEEE (2022)
53. Liu, J., Lou, J., Xiong, L., Liu, J., Meng, X.: Projected federated averaging with heterogeneous differential privacy. Proc. VLDB Endow. **15**(4), 828–840 (2021)
54. Liu, Y., Kang, Y., Xing, C., Chen, T., Yang, Q.: A secure federated transfer learning framework. IEEE Intelligent Systems **35**(4), 70–82 (2020)
55. Liu, Y., Kang, Y., Zhang, X., Li, L., Cheng, Y., Chen, T., Hong, M., Yang, Q.: A communication efficient collaborative learning framework for distributed features. arXiv preprint arXiv:1912.11187 (2019)
56. Liu, Y., Kang, Y., Zou, T., Pu, Y., He, Y., Ye, X., Ouyang, Y., Zhang, Y., Yang, Q.: Vertical federated learning: Concepts, advances and challenges. CoRR **abs/2211.12814** (2023). DOI 10.48550/arXiv.2211.12814. URL `https://doi.org/10.48550/arXiv.2211.12814`
57. Lu, L., Ding, N.: Multi-party private set intersection in vertical federated learning. In: 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020, pp. 707–714. IEEE (2020)
58. Ma, J., Zhang, Q., Lou, J., Ho, J.C., Xiong, L., Jiang, X.: Privacy-preserving tensor factorization for collaborative health data analysis. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019, pp. 1291–1300. ACM (2019)
59. Malek Esmaeili, M., Mironov, I., Prasad, K., Shilov, I., Tramer, F.: Antipodes of label differential privacy: Pate and alibi. Advances in Neural Information Processing Systems **34**, 6934–6945 (2021)
60. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, vol. 54, pp. 1273–1282. PMLR (2017)
61. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins and PSI for secret shared data. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1271–1287. ACM (2020)
62. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: A distributed framework for emerging AI applications. In: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018, pp. 561–577. USENIX Association (2018)

63. Nargesian, F., Pu, K.Q., Zhu, E., Bashardoost, B.G., Miller, R.J.: Organizing data lakes for navigation. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pp. 1939–1950. ACM (2020)
64. Olteanu, D.: The relational data borg is learning. Proc. VLDB Endow. **13**(12), 3502–3515 (2020)
65. Overman, T., Blum, G., Klabjan, D.: A primal-dual algorithm for hybrid federated learning. CoRR **abs/2210.08106** (2022). DOI 10.48550/arXiv.2210.08106. URL `https://doi.org/10.48550/arXiv.2210.08106`
66. Picado, J., Termehchy, A., Fern, A., Ataei, P.: Schema independent relational learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, pp. 929–944. ACM (2017)
67. Raedt, L.D., Kersting, K., Natarajan, S., Poole, D.: Statistical Relational Artificial Intelligence: Logic, Probability, and Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2016)
68. Ramakrishnan, R., Gehrke, J.: Database management systems (3. ed.). McGraw-Hill (2003)
69. Rouhani, B.D., Riazi, M.S., Koushanfar, F.: DeepSecure: Scalable provably-secure deep learning. In: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6 (2018)
70. Saha, S., Ahmad, T.: Federated transfer learning: Concept and applications. Intelligenza Artificiale **15**(1), 35–44 (2021)
71. Sandha, S.S., Cabrera, W., Al-Kateb, M., Nair, S., Srivastava, M.B.: In-database distributed machine learning: Demonstration using teradata SQL engine. Proc. VLDB Endow. **12**(12), 1854–1857 (2019)
72. Santos, A.S.R., Bessa, A., Chirigati, F., Musco, C., Freire, J.: Correlation sketches for approximate join-correlation queries. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pp. 1531–1544. ACM (2021)
73. Schleich, M.: Structure-aware machine learning over multi-relational databases. Ph.D. thesis, University of Oxford, UK (2020). URL `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.808366`
74. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, pp. 3–18. ACM (2016)
75. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, pp. 1642–1659. ACM (2019)
76. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: Learning models over relational data: A brief tutorial. In: Scalable Uncertainty Management - 13th International Conference, SUM 2019, Compiègne, France, December 16-18, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11940, pp. 423–432. Springer (2019)
77. Segev, A.: Optimization of join operations in horizontally partitioned database systems. ACM Trans. Database Syst. **11**(1), 48–80 (1986)
78. Shekelyan, M., Cormode, G., Ma, Q., Shanghooshabad, A.M., Triantafillou, P.: Streaming weighted sampling over join queries. In: Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023, pp. 298–310 (2023)
79. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)
80. Vepakomma, P., Gupta, O., Swedish, T., Raskar, R.: Split learning for health: Distributed deep learning without sharing raw patient data. arXiv preprint arXiv:1812.00564 (2018)
81. Wang, J., Liu, Q., Liang, H., Joshi, G., Poor, H.V.: A novel framework for the analysis and design of heterogeneous federated learning. IEEE Trans. Signal Process. **69**, 5234–5249 (2021)
82. Wu, Y., Cai, S., Xiao, X., Chen, G., Ooi, B.C.: Privacy preserving vertical federated learning for tree-based models. Proceedings of the VLDB Endowment **13**(12), 2090–2103 (2020)
83. Xie, C., Chen, P., Zhang, C., Li, B.: Improving privacy-preserving vertical federated learning by efficient communication with ADMM. CoRR **abs/2207.10226** (2022). DOI 10.48550/arXiv.2207.10226. URL `https://doi.org/10.48550/arXiv.2207.10226`
84. Xie, Y., Wang, Z., Gao, D., Chen, D., Yao, L., Kuang, W., Li, Y., Ding, B., Zhou, J.: Federatedscope: A flexible federated learning platform for heterogeneity. Proc. VLDB Endow. **16**(5), 1059–1072 (2023)
85. Xu, Z., De, S., Figueiredo, M.A.T., Studer, C., Goldstein, T.: An empirical study of ADMM for nonconvex problems. CoRR **abs/1612.03349** (2016)
86. Yakout, M., Ganjam, K., Chakrabarti, K., Chaudhuri, S.: Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, pp. 97–108. ACM (2012)
87. Yang, K., Gao, Y., Liang, L., Yao, B., Wen, S., Chen, G.: Towards factorized SVM with gaussian kernels over normalized data. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, pp. 1453–1464. IEEE (2020)
88. Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol. **10**(2), 12:1–12:19 (2019). DOI 10.1145/3298981. URL `https://doi.org/10.1145/3298981`
89. Yang, S., Ren, B., Zhou, X., Liu, L.: Parallel distributed logistic regression for vertical federated learning without third-party coordinator. arXiv preprint arXiv:1911.09824 (2019)
90. Yousefpour, A., Shilov, I., Sablayrolles, A., Testuggine, D., Prasad, K., Malek, M., Nguyen, J., Ghosh, S., Bharadwaj, A., Zhao, J., Cormode, G., Mironov, I.: Opacus: User-friendly differential privacy library in pytorch. CoRR **abs/2109.12298** (2021). URL `https://arxiv.org/abs/2109.12298`
91. Zeng, Q., Patel, J.M., Page, D.: Quickfoil: Scalable inductive logic programming. Proc. VLDB Endow. **8**(3), 197–208 (2014)
92. Zhang, Q., Gu, B., Deng, C., Huang, H.: Secure bilevel asynchronous vertical federated learning with backward updating. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 10,896–10,904 (2021)
93. Zhang, X., Yin, W., Hong, M., Chen, T.: Hybrid federated learning: Algorithms and implementation. CoRR **abs/2012.12420** (2020). URL `https://arxiv.org/abs/2012.12420`
94. Zhao, Z., Christensen, R., Li, F., Hu, X., Yi, K.: Random sampling over joins revisited. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pp. 1525–1539. ACM (2018)
95. Zhou, Z., Tian, Y., Peng, C.: Privacy-preserving federated learning framework with general aggregation and multiparty entity matching. Wirel. Commun. Mob. Comput. **2021**, 6692,061:1–6692,061:14 (2021)
96. Zhu, E., Nargesian, F., Pu, K.Q., Miller, R.J.: LSH ensemble: Internet-scale domain search. Proc. VLDB Endow. **9**(12), 1185–1196 (2016)

A ADMM: Problem decomposition details

Different from SGD, ADMM aims to decompose the global optimization problem of Eq. 2 into multiple sub-problems, as independent optimization problems. To achieve this, we follow the *sharing ADMM* paradigm to rewrite Eq. 2 to the following Eq. 27, by introducing auxiliary variables $z = \{z_j\}_{j=1}^N$ where $z_j \in \mathbb{R}^{d_c}$:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{N}\sum_{j=1}^{N} \ell\left(z_j; y_j\right) + \beta \sum_{i=1}^{M} \mathcal{R}_i(\theta_i), \\ \text{subject to} \quad & \sum_{i=1}^{M} h_{i,j} - z_j = 0,\ \forall j \in [N],\ h_{i,j} = f_i(\theta_i; T_{i,p_i(j)}). \end{aligned} \tag{27}$$

We then add a quadratic term to the Lagrangian of Eq. 27, which results in Eq. 28 and is known as *augmented Lagrangian*. Here, $\{\lambda_j\}_{j=1}^N$ are dual variables and $\lambda_j \in \mathbb{R}^{d_c}$.

$$\begin{aligned} \min \mathcal{L}\left(\theta_i, z_j, \lambda_j\right) = & \frac{1}{N}\sum_{j=1}^{N} \ell\left(z_j; y_j\right) + \beta \sum_{i=1}^{M} \mathcal{R}_i(\theta_i) \\ & + \frac{1}{N}\sum_{j=1}^{N} \lambda_j^\top \left(\sum_{i=1}^{M} f_i(\theta_i; T_{i,p_i(j)}) - z_j\right) \\ & + \frac{\rho}{2N}\sum_{j=1}^{N} \left\|\sum_{i=1}^{M} f_i(\theta_i; T_{i,p_i(j)}) - z_j\right\|^2. \end{aligned} \tag{28}$$

To simplify notation, we define residual variables $\{s_{i,j}\}_{i\in[M],j\in[N]}$ for each table T_i as follows, where $s_{i,j} \in \mathbb{R}^{d_c}$:

$$s_{i,j} = \sum_{k=1,k\neq i}^{M} f_i(\theta_k; T_{k,p_k(j)}) - z_j. \tag{29}$$

Given the optimization problem of Eq. 28 as follows, we next detail how to leverage *Alternating Direction Method of Multipliers* (ADMM) to decompose this problem to sub-problems.

$$\min \mathcal{L}\left(\theta_i, z_j, \lambda_j\right) = \frac{1}{N}\sum_{j=1}^{N} \ell\left(z_j; y_j\right) + \beta \sum_{i=1}^{M} \mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N} \lambda_j^\top \left(\sum_{i=1}^{M} f_i(\theta_i; T_{i,p_i(j)}) - z_j\right) + \frac{\rho}{2N}\sum_{j=1}^{N} \left\|\sum_{i=1}^{M} f_i(\theta_i; T_{i,p_i(j)}) - z_j\right\|^2$$

To be simple, we first define residual variables $\{s_{i,j}\}_{i\in[M],j\in[N]}$ for each table T_i as $s_{i,j}^t = \sum_{k=1,k\neq i}^{M} f_i(\theta_k^t; T_{k,p_k(j)}) - z_j^t$ and $s_{i,j}^t \in \mathbb{R}^{d_c}$. We then apply ADMM and obtain following sub-problems (three updates), including client-side θ-update and server-side z-update and λ-update. Here, a^t refers to the value of a in the t-th epoch, while $z_j \in \mathbb{R}^{d_c}$ and $\lambda_j \in \mathbb{R}^{d_c}$.

$$\theta_i^{t+1} := \underset{\theta_i}{\operatorname{argmin}} \left(\beta \mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left[\lambda_j^{t\top} f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|s_{i,j}^t + f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right]\right) \tag{30}$$

$$z_j^{t+1} := \underset{z_j}{\operatorname{argmin}} \left(\ell\left(z_j; y_j\right) - \lambda_j^{t\top} z_j + \frac{\rho}{2}\left\|\sum_{i=1}^{M} f_i(\theta_i^{t+1}; T_{i,p_i(j)}) - z_j\right\|^2\right) \tag{31}$$

$$\lambda_j^{t+1} := \lambda_j^t + \rho\left(\sum_{i=1}^{M} f_i(\theta_i^{t+1}; T_{i,p_i(j)}) - z_j^{t+1}\right) \tag{32}$$

To simplify the notations and algorithm description, we use z_j^t-update and λ_j^t-update instead of z_j^{t+1}-update and λ_j^{t+1}-update, and move them before θ_i^{t+1}-update as they are executed in the t-th epoch. The resulting equations are as follows and are equivalent to the above equations.

$$z_j^{t} := \underset{z_j}{\operatorname{argmin}} \left(\ell\left(z_j; y_j\right) - \left(\lambda_j^{t-1}\right)^\top z_j + \frac{\rho}{2}\left\|\sum_{i=1}^{M} f_i(\theta_i^{t}; T_{i,p_i(j)}) - z_j\right\|^2\right) \tag{33}$$

$$\lambda_j^{t} := \lambda_j^{t-1} + \rho\left(\sum_{i=1}^{M} f_i(\theta_i^{t}; T_{i,p_i(j)}) - z_j^{t}\right) \tag{34}$$

$$\theta_i^{t+1} := \underset{\theta_i}{\operatorname{argmin}} \left(\beta \mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left[\lambda_j^{t\top} f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|s_{i,j}^t + f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right]\right) \tag{35}$$

B ADMM: Details of computation and communication reduction

B.1 Computation reduction

For our table mapping, recall that the tuple number of X_i is N, the tuple number of T_i is n_i, and $X_{i,j}$ (the j-th tuple of X_i) comes from $T_{i,p_i(j)}$. In reverse, $T_{i,j}$ (the j-th tuple of T_i) can be mapped to multiple tuples in X_i, and we refer to the index set of these tuples as $G_i(j)$. $|G_i(j)|$ denotes the total tuple number in the $G_i(j)$. Using $G_i(j)$, we can aggregate the weights of duplicated tuples, and then rewrite the θ_i-update of Eq. 9 as follows, where $h_{i,j} = f_i(\theta_i; T_{i,j})$.

$$\begin{aligned}
\theta_i^{t+1} &:= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left[\lambda_j^{t\top} f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|s_{i,j}^t + f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right]\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left[\lambda_j^{t\top} f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|s_{i,j}^t\right\|^2 + \rho(s_{i,j}^t)^\top f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right]\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left[\left(\lambda_j^t + \rho s_{i,j}^t\right)^\top f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2}\left\|f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right]\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{N}\left(\lambda_j^t + \rho s_{i,j}^t\right)^\top f_i(\theta_i; T_{i,p_i(j)}) + \frac{\rho}{2N}\sum_{j=1}^{N}\left\|f_i(\theta_i; T_{i,p_i(j)})\right\|^2\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{n_i}\left(\sum_{g\in G_i(j)}(\lambda_g^t + \rho s_{i,g}^t)\right)^\top f_i(\theta_i; T_{i,j}) + \frac{\rho}{2N}\sum_{j=1}^{n_i}|G_i(j)|\left\|f_i(\theta_i; T_{i,j})\right\|^2\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{n_i}\left[\left(\sum_{g\in G_i(j)}(\lambda_g^t + \rho s_{i,g}^t)\right)^\top f_i(\theta_i; T_{i,j}) + \frac{\rho|G_i(j)|}{2}\left\|f_i(\theta_i; T_{i,j})\right\|^2\right]\right)
\end{aligned} \tag{36}$$

Now, for the θ_i-update of each table T_i, we have reduced the computation complexity from $O(N)$ (i.e., $\sum_{j=1}^{N}$) to $O(n_i)$ (i.e., $\sum_{j=1}^{n_i}$). We can use SGD to solve the θ_i-update problem of Eq. 36.

B.2 Communication reduction

Currently, to perform θ_i-update of Eq. 36 in the client, the server needs to send $\lambda \in \mathbb{R}^{N\times d_c}$, $s_i \in \mathbb{R}^{N\times d_c}$, and $\{G_i(j)\}_{j=1}^{n_i}$ variables to the client that owns T_i. Here, suppose T_i is not horizontally split, the communication complexity is $O(N)$ between the server and each client. To reduce the communication, we can aggregate these variables to be $Y_i \in \mathbb{R}^{n_i\times d_c}$ and $G_i \in \mathbb{R}^{n_i}$ in the sever using the Eq. 37 and Eq. 38 as follows, and then send them to the client owns T_i. Recall that $G_i(j)$ denotes $T_{i,j}$ appears in multiple positions (an index set) in X_i after joins. Therefore, for each $T_{i,j}$, $G_{i,j} = |G_i(j)|$ denotes how many times $T_{i,j}$ appears in X_i after joins, while $Y_{i,j}$ denotes the j-th element of the aggregation of λ and s_i. Thus, the server also does not need to send the table mapping information (i.e., $p_i(j)$) to the clients.

$$Y_{i,j}^t = \sum_{g\in G_i(j)}(\lambda_g^t + \rho s_{i,g}^t) \qquad j = 1 \to n_i \tag{37}$$

$$G_{i,j} = |G_i(j)| \qquad j = 1 \to n_i \tag{38}$$

After that, we can rewrite the θ_i-update of Eq. 36 as follows.

$$\begin{aligned}
\theta_i^{t+1} &:= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{n_i}\left[\left(\sum_{g\in G_i(j)}(\lambda_g^t + \rho s_{i,g}^t)\right)^\top f_i(\theta_i; T_{i,j}) + \frac{\rho|G_i(j)|}{2}\left\|f_i(\theta_i; T_{i,j})\right\|^2\right]\right)\\
&= \underset{\theta_i}{\operatorname{argmin}}\left(\beta\mathcal{R}_i(\theta_i) + \frac{1}{N}\sum_{j=1}^{n_i}\left[(Y_{i,j}^t)^\top f_i(\theta_i; T_{i,j}) + \frac{\rho G_{i,j}}{2}\left\|f_i(\theta_i; T_{i,j})\right\|^2\right]\right)
\end{aligned} \tag{39}$$

After this communication reduction, the communication complexity between the server and the client drops from $O(N)$ to $O(n_i)$.

C Comparison with tree-based models

We have tested tree-based models on our datasets using XGBoost, which is the accuracy baseline of tree-based VFL algorithms such as SecureBoost [19]. We use grid search to tune the hyperparameters of tree-based models such as tree number, iteration number, and learning rate. As shown in Table 9, our approach achieves higher accuracy on MIMIC-III and is comparable to XGBoost on Yelp and MovieLens-1M. Note that we use much larger datasets (exceeding 1 million rows) than that (maximum 50K rows) used in [34]. This result indicates that linear models and neural networks can be more competitive with tree-based models on large datasets. In addition, neural networks is generally well-suited for scenarios involving unstructured data, such as text stored as embeddings in relational tables.

As mentioned by the reviewer, extending TablePuppet to support tree-based models is a valuable direction. However, this introduces new architectural and algorithmic challenges:

Table 9 The accuracy comparison between tree-models and linear/NN models in `TablePuppet`.

Dataset	**Tree-models**	**TablePuppet (Linear/NN)**
MIMIC-III	95.1%	**98.0%**
Yelp	**64.9%**	64.7%
MovieLens-1M	1.10 (RMSE)	**1.09 (RMSE)**

(1) **Effective distributed tree structure:** We must devise an effective method to build and split trees across tables that are partitioned both vertically and horizontally within a single framework. This is a non-trivial problem that goes beyond the scope of existing vertical or horizontal federated tree-based models.

(2) **Algorithmic incompatibility:** Tree-based models such as GBDT rely on histogram-based split-finding, which is different from the gradient-based aggregation used by our current SGD/ADMM optimization. To integrate tree-based models, we need to rethink our core operators to support secure and efficient histogram aggregation over the results of complex join/union queries.

We will leave the solutions to these challenges as future work.