Finding Joinable Tables in Data Lakes with Multi-Dimensional Vectors

Yuyang Dong† Kunihiro Takeoka† Masafumi Oyamada†
† NEC Corporation, Japan
{dongyuyang,k_takeoka,oyamada}@nec.com

ABSTRACT

Finding joinable tables in data lakes is an important technique for many applications such as data integration, data (feature) augmentation, data analysis, and data market. Existing works have a drawback that only focuses on finding joinable tables with equi-join. However, since different tables in data lakes do not have common rules in terms of notation, and equi-join can not capture the semantics of strings, so simply evaluating the relevance of tables with equi-joins will cause many potentially valuable tables to be lost.

In this paper, we define a novel search problem to find joinable tables with multi-dimensional vectors. We solve the above drawback by representing original columns in sets of multi-dimensional vectors. To solve this problem efficiently, we develop PEXESO, a general framework that can handle arbitrary threshold values and a large space of similarity functions. PEXESO reduces computation with a block-and-verify method with pivot-filtering and matching and efficiently finds joinable tables. We also test the effectiveness of our method with ML tasks. The experiment confirmed that the retrieved tables from our method can enhance ML models and have better performance than the retrieved tables from the existing equi-join method.

PVLDB Reference Format:
Yuyang Dong† Kunihiro Takeoka† Masafumi Oyamada† . Finding Joinable Tables in Data Lakes with Multi-Dimensional Vectors. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Join is a fundamental and essential operation that connects two or more tables, and it is also a significant technique applied to relational database management systems and business intelligence tools for data analysis. The benefits of joining tables are not only in the database fields (data integration, data mining, data indexing) but also in the machine learning (ML) fields such as feature augmentation and data enrichment [16, 19, 20, 31].

With the trends of open data movements by governments and the dissemination of data lake solutions in industries, we have more opportunities to obtain a huge number of tables from data lakes (e.g., WDC Web Table Corpus [32]) and make use of them to improve our local tables. Many researchers studied the problem of searching for joinable tables from data lakes. Unfortunately, existing works [44, 46] only focused on evaluating the joinability between columns by taking the overlap number of equi-joined records. One example is joining the Name column in Table 1a with the Host column in Table 1b. "Tom" and "Tyke" are equi-join results as they exactly match. However, the tables in data lakes usually do not have an explicitly specified schema and heterogeneous tables may have different representations, the strings in these tables may not be exactly the same, e.g., the "414 EAST 10TH STREET, 4E" and "4E, 414 East 10th St." in Tables 1a and 1b. In these cases, equi-join either produces only a few join results if we use an inner-join, or causes sparsity if we use a left-join. This may not improve the effectiveness for ML tasks and sometimes even degrades the quality due to overfitting.

Name	Income	Age	Address
Tom	50	45	414 EAST 10TH STREET, 4E
S.Bruce	34	24	206 EAST 7TH STREET, 17,18
Jerr.	24	30	616 EAST 9TH STREET, 4W
Tyke	100	20	303 EAST 8TH STREET, 3F

(a) Membership table

Name	Host	Location	Price
Cozy Clean	Tom	4E, 414 East 10th St.	125
Cute & Cozy	Jerry	616 East 9th St.	540
Central Manhattan	Spike	230 West 8th St.	1265
Sweet and Spacious	Tyke	3F, 303 East 8th St.	1000

(b) Hotel information table

Table 1: Examples of similarity joinable tables.

A deeper view on the semantic level, such as word embeddings, enables us to identify text with the same or similar meanings, hence to tackle the data heterogenity. Therefore, we can solve the aforementioned drawbacks of equi-join and cope with a joinable table search problem by representing each record of a column as a multi-dimensional vector, and a column is represented as a set of multi-dimensional vectors. Then, we leverage the *similarity* between vectors to evaluate the relevance (joinability) between columns. The similarity between multi-dimensional vectors can help to better discover the relations between tables.

On the basis of above motivation, in this paper, we study the problem of finding joinable tables with multi-dimensional vectors. To the best of our knowledge, this is the first work targeting multi-dimensional data with the finding of joinable tables. Note that a few recent studies deal with the problem of joining tables for feature augmentation for ML tasks [16, 19, 20, 25, 31]. They assumed a few candidate tables are existed to join, and focused on the efficient processing of feature selection towards these candidate tables, while

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

our work targets the discovery of these candidate tables from data lakes, which can be fed to these solutions.

1.1 Solution Overview

The problem of finding joinable tables with multi-dimensional vectors has two challenges. One is the **expensive similarity computations** for multi-dimensional data. For example, a distributed representation like GloVe [13] transforms a word to a 50- to 300-dimensional vector. It would be time-consuming to naively compute the similarity between all pairs of records. Another challenge is the **large number of tables in data lakes**. Normally, there is a huge number of tables in data lakes, so it would be time-consuming to check whether these tables are joinable or not one by one. Existing research on joining multi-dimensional data only focused on joining two columns efficiently [4, 12, 15, 18], but they were not designed for searching for joinable columns with similarity constraints in data lakes. [4, 6, 18, 28] proposed the pre-indexing of two datasets. These methods are not applicable to our problem since we are unaware of the query table in advance. Another line of work [11, 27] designed an index for joining with a fixed similarity threshold. Heavy index rebuilding is required when users want to adjust the threshold.

Seeing the above challenges, we propose a framework called PEXESO[1] to efficiently solve above challenges of finding joinable tables with multi-dimensional vectors. PEXESO can deal with any query condition and a huge space of similarity functions. From a high-level overview, PEXESO uses a two-step index that consists of a hierarchical grid and an inverted index to prune data with a block-and-verify methodology and pivot-based filtering techniques. The hierarchical grid can quickly block candidates, and the inverted index can efficiently verify these candidates. For the case of a large number of tables in data lakes, we propose a boosting solution called PEXESO forest. We partition the datasets and index each part with a single PEXESO, then assemble all PEXESOs as a PEXESO forest and search with them linearly. We also propose a KL-clustering that groups tables with a similar distribution to boost each PEXESO in the PEXESO forest.

We conducted experiments on real datasets and considered two ML tasks, Airbnb price prediction [1] and company classification [8], to show the effectiveness of our similarity-based approach of joining multi-dimensional vectors in data lakes. In the Airbnb price prediction task, our approach found 5x records compared to equi-join and had +3.25% performance gain in terms of root mean squared error. In the company classification task, our approach found 5.8x records compared to equi-join and had +6.08% performance gain. We also observed that compared to using the original table without joining, equi-join even reduced the performance by 0.62% and 2.30% for two tasks, because it causes sparsity in the joined table, which leads to overfitting. In addition, we evaluated the efficiency of our approach. PEXESO outperforms the baseline methods with a speed-up of up to 300 times. Its processing speed is competitive against the approximate method, and even better in some cases.

Our contributions are summarized as follows.

- We propose a novel problem that involves searching for joinable tables for multi-dimensional vectors in metric space.
- We propose PEXESO, a framework for our search problem that can deal with all similarities/distances defined in metric space. PEXESO offers a block-and-verify solution through a two-step index structure with hierarchical grids and an inverted index, and it efficiently processes pivot-based filtering techniques for reducing distance computations. Moreover, we give a cost model for PEXESO that helps in estimating the expected computations.
- We propose PEXESO forest for processing out-of-core cases with a large-scale dataset. We propose a KL-clustering method that efficiently partitions source columns and boosts the performance of PEXESO forest.
- We conduct experiments on real datasets to demonstrate the effectiveness of our approach in building machine learning models and the efficiency in finding joinable tables in data lakes.

Symbol	Description
$\mathcal{S}$	collection of columns
$\mathcal{S}_V$	all vectors in all columns of $\mathcal{S}$
$\mathcal{S}_V{}'$	pivot mapped vectors of $\mathcal{S}_V$
Q	query column
$d(.)$	distance function
$sim(.)$	column similarity function
τ	distance threshold
T	column similarity threshold
$M_\tau^d(a,b)$	vector a and b are matched, that $d(a,b) \leq \tau$
P	pivot set
$SQR(q',\tau)$	square query region
$RQR(q',p,\tau)$	rectangle query region
$HG_{\mathcal{S}_V{}'}$	hierarchical grid for $\mathcal{S}_V{}'$
m	number of levels in hierarchical grid
$invt$	inverted index

Table 2: Frequently used notations.

2 PRELIMINARIES

In this section, we give an overview of our joinable table search system and present a formal definition of the joinable table search problem. We also introduce background materials on metric space, pivot-based filtering and validation. Table 2 lists the notations frequently used in this paper.

2.1 System Overview

In this work, we focus on finding joinable tables with distance similarities between multi-dimensional vectors. Figure 1 gives an overview of our system. After loading the tables from the data lakes to our table repository, we extract the columns that are expected to be join keys from all tables (e.g., WDC Web Table Corpus [32] contains the key column information). For each extracted column, we transform each record into a multi-dimensional vector, according to the value type of the column [2]. Then the original column is represented as a set of multi-dimensional vectors. For example, we use the word embedding model to transform a column of strings and use the HSV (hue, saturation, value) color model to transform the column of images. We create several proposed PEXESO frameworks to manage

[1] PEXESO is a card game and the objective is to find matched pairs. https://en.wikipedia.org/wiki/Concentration_(card_game)

[2] We skip the transformation if the original values are already multi-dimensional vectors.

the transformed vectors, and each PEXESO framework takes charge of indexing and searching for a kind of value type.

Regarding searching for joinable tables with our system, users are asked to input a table and specify a query column to join. The system first transforms that query column into a set of multi-dimensional vectors based on the value type, and the transformation method is the same as that we used for the source tables from the data lakes. Then, the transformed query column is assigned to a corresponding PEXESO framework. Last, PEXESO processes the searching and returns the joinable tables.

2.2 Problem Definition

For two vectors v_1 and v_2, we define vector matching with a distance function d and a distance threshold τ.

Definition 1 (Vector Matching). *Given two vector v_1 and v_2, a distance function d, and a distance threshold τ, if the distance between v_1 and v_2 is less than or equal to τ, then v_1 and v_2 are defined as matched, which is denoted as $M_\tau^d(v_1, v_2)$, i.e., $\{M_\tau^d(v_1, v_2) | d(v_1, v_2) \leq \tau\}$.*

Given a query column Q and a target column X, we use the number of matching vectors in these two columns to define two kinds of column similarities. One is column similarity for many-to-many vector matching (sim_{mm}). It counts the number of all matched pairs of vectors from Q and X, and normalizes the number by the total combinations.

$$sim_{mm}(Q, X, \tau, d) = \frac{|M_\tau^d(q, x)|}{|Q| \cdot |X|},\ q \in Q, x \in X \tag{1}$$

The other is column similarity for one-to-many vector matching (sim_{om}). It counts the number of vectors in Q that can match vectors in X, and normalized the number by the size of Q.

$$sim_{om}(Q, X, \tau, d) = \frac{|Q_M|}{|Q|},\ \exists M_\tau^d(q, x), q \in Q_M, Q_M \subset Q, x \in X \tag{2}$$

If the similarity between two columns is larger than or equal to a column similarity threshold T, we consider these columns to be joinable, and we can also say that the tables of these two joinable columns are joinable. Finally, we define the joinable column (table) search problem.

Definition 2 (Joinable Column (table) Search). *Given a collection of columns $\mathcal{S}$, a query column Q, a distance function d, a distance threshold τ, a column similarity sim, and a column similarity threshold T, joinable column search problem is to find a subset $\mathcal{X}$ from $\mathcal{S}$, and the similarity between each column in $\mathcal{X}$ and Q is larger than or equal to T. i.e., $\{\mathcal{X} | sim(Q, X, \tau, d) \geq T,\ X \in \mathcal{X}, \mathcal{X} \subset \mathcal{S}\}$.*

2.3 Metric Space

The metric space is a universal and versatile model of similarity that can be applied in various areas of information retrieval. In this paper, we assume that all multi-dimensional vectors from all columns are in a metric space. This metric space is represented as a two-tuple $(\mathcal{S}_V, d)$, in which $\mathcal{S}_V$ is a vector universe and d is a distance function for measuring the similarity between two vectors in $\mathcal{S}_V$. The distance function d satisfies four properties: (1) symmetry: $d(p, q) = d(q, p)$, (2) non-negativity: $d(p, q) \geq 0$, (3) identity: $d(p, q) = 0$ iff $p = q$, and (4) triangle inequality: $d(p, q) \leq d(p, o) + d(q, o)$.

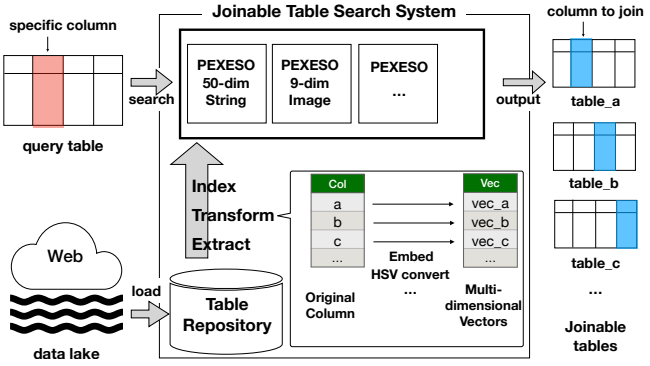


Figure 1: Joinable table search system.

For a specific vector, the problem of finding matched vectors w.r.t Definition 1 is formalized to a range search problem in the metric space. The solutions for accelerating range search in metric space can be classified into two categories: *compact partitioning methods* and *pivot-based methods*. [7] is a sufficient survey and explained that pivot-based methods outperform compact partitioning methods, because of the power of reducing the number of distance computations with pre-computed distances and the triangle inequality. Moreover, the pivot-based methods can map the original data into a low dimensional pivot space, which has the effect of dimension reduction to avoid the "curse of dimensionality" problem. Therefore, in this paper, we focus on the pivot-based methods.

2.4 Pivot-based Filtering and Validation

Pivot-based filtering uses pre-computed distances to prune vectors on the basis of the triangle inequality. Specifically, the distance from each vector to a set of pivots P is pre-computed and stored. Given two vectors q and o, for every pivot $p \in P$, the distance $d(q, o)$ cannot be smaller than $|d(q, p) - d(o, p)|$ on the basis of the triangle inequality. Here, we apply pivot-based filtering to identify the unmatched vectors.

Lemma 1 (Pivot Filtering). *Given a set P of pivots, a target vector o, a query vector q, a distance function d, and a distance threshold τ, if $d(o, p) \notin [d(q, p) - \tau, d(q, p) + \tau],\ p \in P$, then o is not match with q.*

Proof. We prove by contradiction. Assume that there exists an vector o matches with q, i.e., $d(q, o) \leq \tau$, but for a pivot p, it holds $d(o, p) \notin [d(q, p) - \tau, d(q, p) + \tau]$, i.e., $|d(o, p) - d(q, p)| > \tau$. According to the triangle inequality, we can know that $d(q, o) \geq |d(o, p) - d(q, p)| > \tau$, and this contradicts the assumption. □

The triangle inequality also has another feature in that the distance $d(q, o)$ cannot be greater than $d(q, p) + d(o, p)$, and we can adjust the pivot-based matching to identify the matching vectors.

Lemma 2 (Pivot Matching). *Given a set P of pivots, a target vector o, a query object q, a distance function d, and a distance threshold τ, if there exists a pivot p such that $d(o, p) \leq \tau - d(q, p)$, then o is matched with q.*

Proof. For a pivot p, the triangle inequality holds $d(q, p) + d(o, p) \geq d(q, o)$. If $d(o, p) \leq \tau - d(q, p)$, then $d(o, p) + d(q, p) \leq \tau$. Therefore, $d(q, o) < \tau$ and o is matched with q. □

3 PEXESO FRAMEWORK

Given a query column Q and a collection of columns $\mathcal{S}$, the naive method for solving the joinable table search problem is that, for each vector in Q, compute the distance with all vectors from all columns, and compute the column similarity to retrieve the joinable columns. Therefore, we need to compute the distance $|Q| \cdot |\mathcal{S}_v|$ times, where $\mathcal{S}_v$ is a collection of all vectors of all columns in $\mathcal{S}$. If we use the pivot-based filtering and matching in Lemmas 1 and 2, it can help to reduce the actual distance computation, but we still need to check these lemmas $|Q| \cdot |\mathcal{S}_v|$ times.

Motivated by this, we propose the PEXESO framework, which aims for efficient processing of pivot-based filtering and matching in a block-and-verify way. Pairwise vector candidates are blocked with merge-join processing across two hierarchical grid structures and efficiently verified by traversing an inverted index structure.

3.1 Pivot Mapping and Query Regions

Given a pivot set $P = \{p_1, p_2, ..., p_n\}$, pivot mapping for a vector x involves computing the distance between x and all pivots, and assembling these values into a mapped vector x'. Specifically, x is mapped to the pivot space of P as $x' = [d(p_1, x), d(p_2, x), ..., d(p_n, x)]$. It is worth noticing that we always select a pivot size with a lower number than the dimensionality of the original metric space. Therefore, the pivot mapping can reduce the dimensionality. We can run a range query efficiently on the low-dimensional pivot space to avoid the "curse of dimensionality."

Figure 2 gives an example of pivot mapping. Assume that in a 2-d metric space, a query column Q that contains two vectors, i.e., $Q : \{q_1, q_2\}$, and four source columns, each of them has two vectors: $s_1 : \{x_1, x_2\}$, $s_2 : \{x_3, x_4\}$, $s_3 : \{x_5, x_6\}$, $s_4 : \{x_7, x_8\}$. These vectors are represented as points in a 2-d metric space. We select x_1 and x_8 as two pivots and map all vectors to a 2-d pivot space. In the pivot space, we can create a square query region $SQR(q', \tau)$ for a mapped query vector q' and a distance threshold τ that uses q' as the center, and the length of the edge is 2τ. On the basis of Lemma 1, all vectors located outside of the square query region $SQR(q', \tau)$ are not the result of range search in the original metric space; hence, they are not matched with q. Therefore, we only need to verify the vectors located in $SQR(q', \tau)$. In Figure 2, we can verify only x'_2, x'_4, x'_6, x'_7, which are located inside the square query region $SQR(q'_1, \tau)$ (marked in red), and safely filter other vectors.

For matching vectors with Lemma 2, we can create a rectangle query region $RQR(q', p_i, \tau)$ to a mapped query q' with a pivot $p_i \in P$. $RQR(q', p_i, \tau)$ is a rectangle region that starts from the original point, the length of the edge in the i dimension is $\tau - d(q, p)$, and the other edges have infinite length. The vectors in $RQR(q', p, \tau)$ must match query q. Notice that query q' can have different rectangle query regions corresponding to different pivots, and we define q' as not having a rectangle query region with pivot p when $\tau - d(q, p)$ is a negative value. In Figure 2, q'_1 does not have rectangle query regions for all pivots, and q'_2 only has a rectangle query region for pivot x_1, which is $RQR(q'_2, x_1, \tau)$ in green. We can see that x'_3 is located in this region. Thus, x_3 and q_2 are matched vectors, and we do not need to verify them.

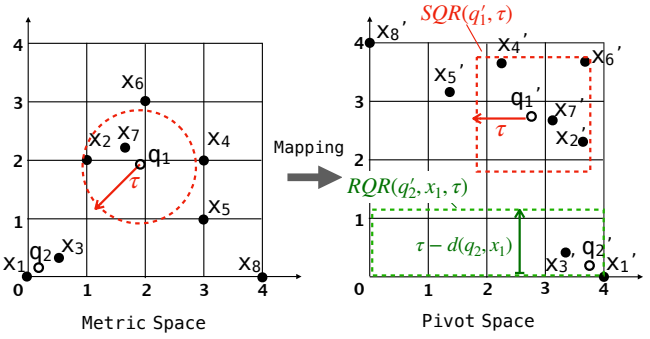


Figure 2: Example of pivot mapping with pivots x_1 and x_8, square query region of pivot filtering for q_1 (red), and rectangle query region of pivot matching for q_2 (green).

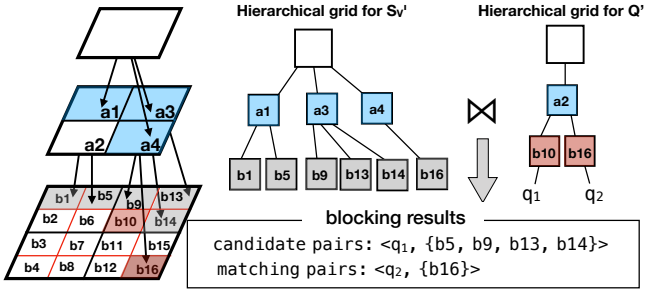


Figure 3: Hierarchical grids for source vectors and query vectors, and the blocking results of matching pairs and candidate pairs.

3.2 Blocking with Hierarchical Grid

To group similar mapped vectors, we equally partition the pivot space into small (hyper) cells and manage the cells in a hierarchical grid. A hierarchical grid has multiple levels for different partition granularity. Figure 3 gives an example of a 2-level (except the root) hierarchical grid structure for the vectors in the pivot space of Figure 2. Assume that we divide each dimension of the pivot space into 2^m partitions, where m is the level number; then, for the 2-d pivot space, we have different levels of granularity with 4 cells and 16 cells. To save on memory cost, the hierarchical grid only indexes the cells that contain vectors. According to Figure 3, the last level (leaf level) has cells of b_1, b_5, b_9, b_{13}, b_{14} and b_{16}, and the first level has a_1, a_3 and a_4.

For a single query vector, we can filter the vectors in a cell at one time if the square query region does not have overlap with this cell.

Lemma 3 (Vector-cell Filtering). *In the pivot space of P, given a cell c, a mapped query vector q', a distance function d and a distance threshold τ, if $c \cap SQR(q', \tau) = \emptyset$ then every original vector $o \in c$ is match with q.*

Proof. $c \cap SQR(q', \tau) = \emptyset$ means all vectors in c is not located in $SQR(q', \tau)$, and for each vector $o \in c$, it holds that $d(o, p) \notin [d(q, p) - \tau, d(q, p) + \tau]$, $p \in P$. Hence, we can prove this with Lemma 1 □

In addition, if we also group query vectors into cells, we can also compute a square query region for a cell of query vectors c_q as

$SQR(c_q.center, \tau + \frac{c_q.length}{2})$, where $c_q.center$ is the center vector, and $c_q.length$ is the edge length of c_q. Then, we can carry out a pivot filtering between two cells:

Lemma 4 (Cell-cell Filtering). *In the pivot space of P, given a cell c, a query cell c_q, a distance function d, and a distance threshold τ, if $c \cap SQR(c_q.center, \tau + \frac{c_q.length}{2}) = \emptyset$, then for all original vectors $o \in c$ and for all vectors $q \in c_q$, they are not matched.*

Proof. $SQR(c_q.center, \tau + \frac{c_q.length}{2})$ is a square range that with the center of c_q and has an edge length of $\tau + \frac{c_q.length}{2}$. For an arbitrary vector which is outside of $SQR(c_q.center, \tau + \frac{c_q.length}{2})$, it holds $d(o, c_q.center) > \tau + \frac{c_q.length}{2}$, and for each $q \in c_q$, it holds $d(q, c_q.center) < \frac{c_q.length}{2}$. Therefore, for all original vectors $o \in c$ and for all vectors $q \in c_q$, they hold $d(o, q) > \tau$, and they are not matched. □

In a similar way, we can also extend Lemma 2 to vector-cell matching and cell-cell matching. If a cell is entirely covered by a rectangle query region of a query vector, then all vectors match this query vector.

Lemma 5 (Vector-cell Matching). *In the pivot space of P, given a cell c, a mapped query vector q', a distance function d and a distance threshold τ, if there exists a pivot $p \in P$ that holds $c \cap RQR(q', p, \tau) = c$, then every original vector $o \in c$ is matched with q.*

Proof. For a single pivot $p \in P$, $c \cap RQR(q', p, \tau) = c$ means all vectors in c is inside of the region $RQR(q', p, \tau)$, and for each vector $o \in c$, it holds that $d(o, p) \leq \tau - d(q, p)$. Hence, we can prove this with Lemma 2 □

To achieve a cell-cell matching, for each pivot $p \in P$, we define the minimum rectangle query region as the intersection region of all $RQR(.)$ from all vectors in c_q , and denote the minimum rectangle query region as $min(RQR(q', p, \tau)), q' \in c_q$. If one of a query $q' \in c_q$ does not have a rectangle query region with a pivot p, then we define c_q as not having the minimum rectangle query region for p.

Lemma 6 (Cell-cell Matching). *In the pivot space, given a cell c, a query cell c_q, a distance function d, and a distance threshold τ, for a pivot p, if c is covered by the minimum matching region $min(RQR(q', p, \tau))$, i.e., $c \cap min(RQR(q', p, \tau)) = c$, then every original vector $o \in c$ is matched with all $q \in c_q$.*

Proof. For a pivot $p \in P$, if $min(RQR(q', p, \tau))$ exists, then all query vectors has rectangle query regions, for p. Therefore, if c is covered by the minimum one, c must covered by all of rectangle query regions. Hence, every original vector $o \in c$ is matched with all $q \in c_q$. □

We index Q' and S'_V in two hierarchical grids $HG_{Q'}$ and $HG_{\mathcal{S}_V'}$. It is important to note that $HG_{Q'}$ associates the mapped vectors of Q' within its leaf cells. However, $HG_{\mathcal{S}_V'}$ is only a frame without associating the vectors from $\mathcal{S}_V'$ in its leaf cells. Since the vectors from different columns of $\mathcal{S}$ may share a common leaf cell in $HG_{\mathcal{S}_V'}$, we only find the leaf cells in the blocking phase, and we will perform an efficient verification in Section 3.3.

Algorithm 1: Block($E_S, E_Q, mPair, cPair$)

```
Input  : E_S, E_Q, mPair, cPair
for each e_S ∈ E_S do
    for each e_Q ∈ E_Q do
        if e_S and e_Q are leaf cells then
            for each v ∈ e_Q do
                if v and e_S are matched w.r.t Lemma 5 then
                    mPair ← mPair ∪ { v, e_S } ;
                else
                    if v can not filter e_S w.r.t Lemma 3 then
                        cPair ← cPair ∪ { v, e_S } ;
        else
            if e_S and e_Q are matched w.r.t Lemma 6 then
                mPair ← mPair ∪ ∀{ v, c }, v ∈ e_Q, c ∈ e_S ;
            else
                if e_S and e_Q cannot be filtered w.r.t Lemma 4
                  then
                    Block(e_S, e_Q, mPair, cPair);
```

The blocking phase aims to find all pairs of $\langle q', leaf\ cells \rangle$ at one time, and there are two kinds of pairs: matching and candidate pairs. Matching pairs are vector-cell pairs that satisfy Lemma 5. Candidate pairs are pairs that cannot be filtered with Lemmas 3 and 4. For the case in Figure 3, the blocking result is: $< q_1, \{ b5, b9, b13, b14 \} >$ for candidate pairs, and $< q_2, \{ b16 \} >$ for matching pairs.

To retrieve match and candidate pairs efficiently, $HG_{Q'}$ and $HG_{\mathcal{S}_V'}$ are constructed with a common level number. We propose a merge-join fashion blocking algorithm to scan both $HG_{Q'}$ and $HG_{\mathcal{S}_V'}$ only once. In particular, cells in $HG_{\mathcal{S}_V'}$ are pruned with the same level cells in $HG_{Q'}$, and the subcells (children) are expanded at the same time on two hierarchical grids. We use Lemmas 4 and 6 to filter and match the non-leaf cells, and use Lemmas 3 and 5 to filter and match the leaf cells. Finally, the pairs of query vectors and corresponding leaf cells in $HG_{\mathcal{S}_V'}$ are retrieved as the blocking results. The pseudo-code is shown in Algorithm 1.

3.3 Verifying with Inverted Index

After getting the matching pairs and candidate pairs with Algorithm 1, we use an inverted index to efficiently verify the distance and compute the column similarity for identifying the joinable columns. The inverted index is a dictionary structure that manages the information of leaf cells for different source columns. We use the leaf cells in $HG_{\mathcal{S}_V'}$ as keys, and each key corresponds to a list of columns that contain that key.

Figure 4 gives an example of the inverted index. We can traverse the inverted index with the query vector and leaf cells from the candidate and matching pairs. We use two global maps to record the number, a *match map* that records the number of matched vectors ($|M_\tau^d(.)|$) for all columns, and an *unmatch map* that records the number of unmatched vectors for all columns. These recorded numbers are used to compute column similarity $sim(.)$ for determining joinable columns.

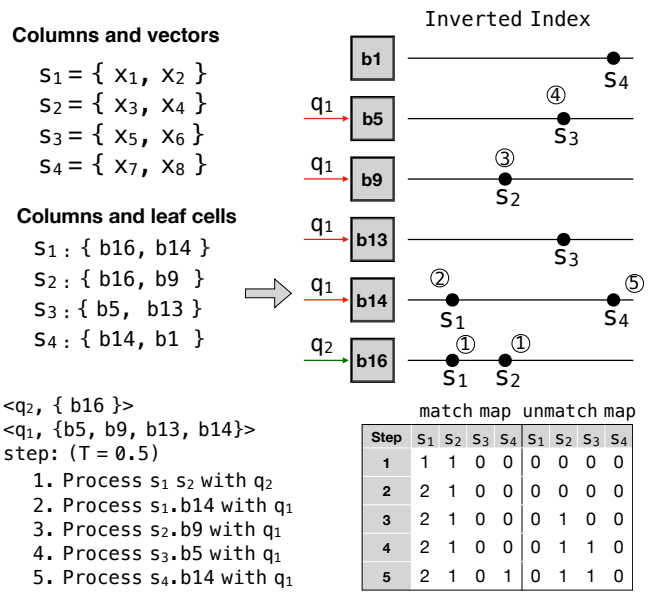
For each matching pair, we just update the match map with the columns contained in the corresponding posting list. For each candidate pair, we need to traverse the posting lists of the leaf cells contained in the candidate pair. During the traversing, for each column in the posting list, we get the cell with the posting list key and access the vectors in the cell. Then, we use Lemmas 1 and 2 to filter and match vectors. We need to compute the exact distance values when we cannot filter or match them. We employ a DAAT (document-at-a-time [5]) paradigm to traverse the inverted index so that we can early stop the further verification for a column when the similarity exceeds the threshold T. If this column has a lot of unmatched vectors and the retained candidates are not enough to make the matched vectors exceed T, we can also early stop the verification of this column according to the following reverse filter.

Lemma 7 (Reverse Filtering). *Given a query column Q, a target column X, a distance function d, a distance threshold τ, a column similarity sim, and a column similarity threshold T, let U be the subset of query vectors from Q that cannot match any vector in X, and if $Max(sim(Q-U, X, \tau, d)) < T$, then X is not a joinable column to Q.*

Proof. We prove by contradiction. Assume that X is joinable to Q and $Max(sim(Q-U), X, \tau) < T$, so there must exist matched vectors in U to make X joinable to Q, which contradicts that U is the subset of query vector from Q that can not match any vector in X. □

Figure 4 also gives the processing steps with the matching pair $< q_2, \{ b16 \} >$ and candidate pair $< q_1, \{ b5, b9, b13, b14 \} >$. Assume that we use the sim_{mm} column similarity function with a threshold of $T = 0.5$, which means we want to find a joinable column that contains at least two vectors matching the query column. In step 1, regards to the matching pair $< q_2, \{ b16 \} >$; we update the s_1 and s_2 in the match map because they belong to the posting list of $b16$. Then we start a DAAT on the candidate pair $< q_1, \{ b5, b9, b13, b14 \} >$ with the steps 2-5. In step 2, we check the cell $b14$ of column S_1, $S_1.b14$ has a vector x_2 and it matches with q_1, so the s_1 in the match map is updated as 2. At the moment, we mark s_1 as a joinable column since $Sim_{mm}(Q, s_1) = 0.5$. In step 3, the $s_2.b9$ has a vector x_4, and it does not match with q_1, so the s_2 in unmatch map is updated as 1. In the same way, s_3 in unmatch map is updated as 1 in step 4. Notice that after step 4, we do not need to check $s3.b13$ since the unmatch number for s_3 is 1, and s_3 can be filtered by the reverse filtering of Lemma 7. In step 5, we check $s4.b14$ and update the match map. Finally, the verification is over and the result is s_1. Algorithm 2 gives the pseudocode of verification.

Quick browsing inverted index. Because we construct HG'_Q and $HG_{S_{V'}}$ with the same level number, the leaf cells between them are also in the same granularity. If a query leaf cell and a source leaf cell have the same cell region, they must satisfy with Lemma 4 and the vectors between them are candidate pairs. Therefore, we can get the leaf cells in HG_Q and probe them with the inverted index directly. We call the above processing *quick browse*. It processes the possible candidates in advance before the blocking processing in Algorithm 1. Moreover, if quick browse is issued, we can adjust Algorithm 1 easily for skipping the cell-vectors candidates in the same cell and avoiding redundant computation.



Step	S_1	S_2	S_3	S_4	S_1	S_2	S_3	S_4
1	1	1	0	0	0	0	0	0
2	2	1	0	0	0	0	0	0
3	2	1	0	0	0	1	0	0
4	2	1	0	0	0	1	1	0
5	2	1	0	1	0	1	1	0

Figure 4: Inverted index for columns and leaf cells.

Algorithm 2: Verify($mPair, cPair, invt, T, \tau$)

```
Input  :mPair, cPair, invt, T, τ
foreach pair p ∈ mPair do
    Get columns in invt by leaf cells in p and update the
      match map;
foreach pair p ∈ cPair do
    foreach set s, cell c in DAAT traversal w.r.t p do
        if s can be filtered with w.r.t Lemma 7 then
            continue to the next set;
        else
            foreach vector v ∈ c do
                if v is filtered or matched with Lemma 1 or 2
                  then
                    update match map and unmatch map;
                else
                    compute d(q, v) and verify M^d_τ(q, v);
                    update match map and unmatch map;
                if f(s, Q) ≤ T then
                    Output s as a result;
                    continue to the next set;
```

3.4 Search Processing and Complexity

We assemble the above techniques and propose a block-and-verify algorithm called PEXESO for solving the joinable column (table) search problem. Algorithm 3 summarizes the flow of the PEXESO algorithm.

Time complexity. We first discuss the time complexity of the search process of Algorithm 3. The mapping and construction of the hierarchical grid for Q takes $O((|P|+m)\cdot|Q|)$. Then the quick browser and block-and-verify method have a complexity of $O(log|Q| \cdot log|S_V|)$.

Algorithm 3: PEXESO($Q, HG_{\mathcal{S}_V'}, inv, \tau, T$)

Input : $Q, HG_{\mathcal{S}_V'}, inv, T, \tau$
Output: $joinableColumns$

1 Map and construct $HG_{Q'}$;
2 Quick browse inv with $HG_{Q'}.leafCell$ and update global match and unmatch maps;
3 $mPair \leftarrow \emptyset, cPair \leftarrow \emptyset$;
4 Block($HG_{Q'}.root, HG_{\mathcal{S}_V'}.root, mPair, cPair$);
5 $joinableColumns \leftarrow$ Verify($mPair, cPair, inv, T, \tau$);
6 **return** $joinableColumns$

Operation	Style	Time Complexity
Search	On-line	$O((\|P\|+m)\cdot\|Q\|+log\|Q\|\cdot log\|\mathcal{S}_V\|)$
Construction	Off-line	$O((\|P\|+m)\cdot\|\mathcal{S}_V\|+D)$
Append	Off-line	$O((\|P\|+m)\cdot\|s\|)$
Delete	Off-line	$O(log\|\mathcal{S}\|)$

Table 3: Summary of time and space complexity

Therefore, the total time complexity for searching with PEXESO is $O((|P|+m)\cdot|Q|+log|Q|\cdot log|\mathcal{S}_V|)$.

For the construction complexity of PEXESO, there are three steps: 1) pivot selection, 2) pivot mapping, and 3) building the index. We used a PCA-based pivot selection algorithm with a complexity of $O(|\mathcal{S}_V|)$ (details are in Section 4.1). Pivot mapping takes $O(|P|\cdot|\mathcal{S}_V|)$. For building the index, it takes $O(m\cdot|\mathcal{S}_V|)$ for the hierarchical grid and $O(D)$ for the inverted index, where D is the total number of cells in all columns. In conclusion, the total cost of PEXESO construction is $O((|P|+m)\cdot|\mathcal{S}_V|+D)$.

Appending a new column s into PEXESO, takes $O(|P|+m)\cdot|s|)$ to pivot map s and insert it into the corresponding cells of the hierarchical grid, and it takes $O(1)$ to insert s into the corresponding posting lists of the inverted index. Deleting a column s from PEXESO, takes $O(1)$ to delete s from the hierarchical grid, and it takes $O(log|\mathcal{S}|)$ to locate and delete s from the inverted index.

Table 3 gives a summary of time complexity for different operations.

Space complexity. There are two hierarchical grids and an inverted index in PEXESO. The space complexity for HG'_Q is $O(|Q|)$, and for $HG_{\mathcal{S}_V'}$ and the inverted index, it is $O(|\mathcal{S}_V|+D)$. Therefore, the total space complexity for PEXESO is $O(|Q|+|\mathcal{S}_V|+D)$.

3.5 Cost Model for Joinable Table Search

We give a cost model for joinable table search with PEXESO by analyzing the expected distance computation number, denoted as E. Recall that PEXESO has a block-and-verify framework, so the overall expected computation number is as follows.

$$E = E_{block} + E_{verify} \tag{3}$$

For the expected computation in the block phase E_{block}, let us assume that, in each level of the hierarchical grid, every dimension of the pivot space is divided into 2^m partitions. There are $2^{|P|\cdot m}$ cells in the leaf level, and each cell contains at least one vector. Therefore,

$$E_{block} = log|Q'|\cdot log(2^{|P|\cdot m}) \tag{4}$$

In the verify phase, we need to check all candidate vector-cell pairs from the block phase, and the expected computation number is:

$$E_{verify} = \sum_{q'\in Q'} N(SQR(q',\tau)) \tag{5}$$

where $N(SQR(q',\tau)$ is the vector numbers of the leaf cells covered by the region of $SQR(q',\tau)$. We give an upper bound for estimating the value of $N(SQR(q',\tau)$. Suppose that we have computed the PDF (probability distribution function) for each dimension of the mapped vectors $\mathcal{S}_V'$, denoted as $PDF_i(\mathcal{S}_V'), i\in[1,|P|]$. Because we need to take the intersection of vectors that cannot be filtered by all dimensions of the pivot space, the maximum number of the above intersection, denoted as $N_{max}(SQR(q',\tau)$, is the minimum number of vectors in the covered region from all dimensions as follows.

$$N_{max}(SQR(q',\tau)) = min_{i\in[1,|P|]}(\int_{q'[i]-\tau-\frac{1}{2^m}}^{q'[i]+\tau+\frac{1}{2^m}} PDF_i(\mathcal{S}_V')) \tag{6}$$

Analyzing optimal m for PEXESO construction. After selecting the pivots P (details are in Section 4.1) and defining a sample of representative queries Q and τ empirically, the only parameter for Equation (3), is the level number m of the hierarchical grid. Therefore, we can find an optimal m by minimizing the expected computation E in Equation (3) with optimizing algorithms such as gradient descent. We confirm the above estimation in Section 5.

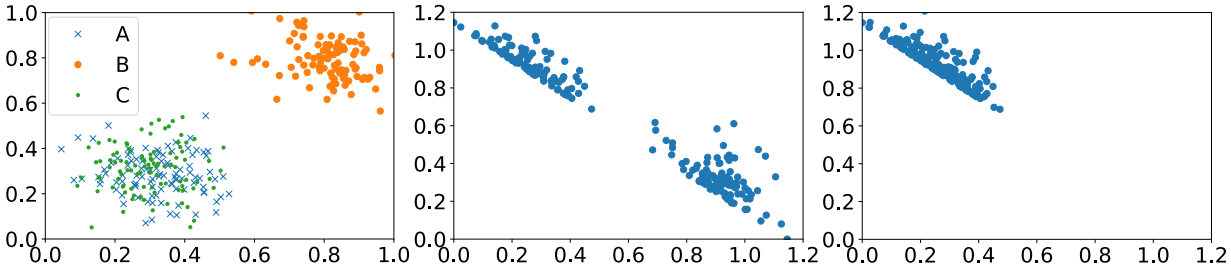
4 PEXESO BOOSTING FOR LARGE DATA

For some cases of the source columns and vectors from data lakes are extremely large, we can not index all data in a single PEXESO and hold them in an in-memory manner. However, benefitting from the flexibility of the PEXESO framework, we can split the source columns into small partitions and solve the searching of joinable tables in a linear processing way. In particular, each partition is indexed into a PEXESO framework, and all PEXESOs are assembled as a PEXESO forest. When processing a joinable table search, every time we read a single PEXESO into memory and search the results. Last, we merge all results from every PEXESO as the final search results.

An important problem is how to make a good partition that can maximize the power of the pivot filtering in each PEXESO. In this section, we first introduce the concept of the pivot selection algorithm, and we propose a boosting of the PEXESO framework with a clustering method with KL-divergence.

4.1 Pivot Selection

The selection of pivots can significantly affect the performance of pivot filtering. Good pivots can map original vectors and make them separate (far away from each other) into the pivot space, so that makes a specific query region cover fewer mapped vectors and filter more. A lot of research on the pivot selection [6, 7, 23] has arrived at a common conclusion: "**Good pivots are outliers, but outliers are not always good pivots**." Due to the cost of the naive pivot selection algorithm is $O(|\mathcal{S}_V|^3)$ in selecting pivots from source vectors $\mathcal{S}_V$, many efficient methods [6, 7, 23] pick some outlier vectors as candidates, then select good pivots from these candidates. In this work, because the source data $\mathcal{S}_V$ is very large, we use the PCA-based method [23] that can select high quality pivots with a low-cost complexity that approximates to $O(|\mathcal{S}_V|)$.



(a) A, B, C columns in original space (b) Pivot mapping A and B columns (c) Pivot mapping A and C columns

Figure 5: Pivot mapping with different groups from A, B and C columns.

4.2 PEXESO Forest with Clustering Columns

Different columns are from different tables in data lakes, so the distributions of the inner vectors of columns are also different. Recall that "Pivots are selected from the outliers", if we group columns with different distributions, the power of selected pivots will decline. For example, there are three columns A,B, and C, in Figure 5a. A has a similar distribution to C, and B has a different distribution from A and C. If we group A and B together and select the pivots from them, the outliers from A are far away from B and the pre-computed distances cannot help with pivot filtering with the vectors in B, and vice versa. If we group similar columns A and C together, the outliers from A can also work for the vectors in B. Figures 5b and 5c are the pivot mapping result for $\{A, B\}$ and $\{A, C\}$, respectively. The white space is the pivot filter region. Obviously, grouping $\{A, B\}$ leads to better filter power than grouping $\{A, C\}$.

Inspired by the above observation, we would like to cluster the source columns according to the similarity between distributions. KLD (Kullback–Leibler divergence) [37] is a measure of how one probability distribution is different from a second. Since KL-divergence is an asymmetric measure, we use the JSD (Jensen–Shannon divergence) [36], which is a distance metric based on KL-divergence, and compare the JSD value while clustering columns with similar distributions.

$$KLD(A||B) = \sum_{x \in X} A(x) \cdot log(\frac{A(x)}{B(x)}) \tag{7}$$

$$JSD(A||B) = \frac{KL(A||B) + KL(B||A)}{2} \tag{8}$$

We propose a KL-clustering algorithm as follows. Notice that KLD is a measure between probability distributions so we need to summarize a column of vectors to a probability distribution histogram with a number of bins, i.e., get the statistics of the probability of points in a space region.

KL-clustering Algorithm

(1) Summarize each column as a probability distribution histogram.
(2) Randomly select the k histogram as the center of k clusters.
(3) For each column histogram, compute the JSD distance in Equation (8) with all k centers, assign this column to the cluster that makes the minimum JSD distance.
(4) For each cluster, compute a new cluster center as the mean of histograms in this cluster.
(5) Repeat (2) - (4) until a user-defined iteration number t.

Figure 6: Search performance of PEXESO forests with KL-clustering, average k-means clustering, and random clustering.

The procedure of the KL-clustering is similar to Lloyd's algorithm [38] of k-means, so it has a complexity of $O(|\mathcal{S}| \cdot k \cdot t)$, where t is the number of iterations.

To confirm the power of the proposed KL-clustering, we conducted a comparison experiment on a real dataset WDC [32] web tables. We took a sample of 10,000 tables with 32,549 columns and transformed each column into a set of 50-dimensional vectors with GloVe embedding [13]. There was a total of 404,598 vectors. We partitioned the data with KL-clustering, random clustering, and average k-means clustering, and then assembled three PEXESO forests and compared the search performance. The average k-means method is an intuitive clustering method; for each column, it takes an average vector to represent this column and runs a k-means clustering with these average vectors.

Figure 6 shows the comparison results of PEXESO forests with different clustering algorithms for varying numbers of clusters. The proposed KL-clustering performance better than the others. It was 1.4 - 1.6 times faster than random clustering and was 1.1-1.2 times faster than the average k-means clustering.

5 EXPERIMENTS

5.1 Setting

Datasets. We used two real datasets to test efficiency. Table 4 summarizes the statistics of them.

- **LOGO**: an image dataset from [21] that contains 2,000,000 images of 194 classes of company logos. We transformed these images into 9-dimensional HSV features; and took a random sample from 194 classes to build 20,000 columns. Each column contained 100

Dataset	Vec #	Col #	Avg Vec #	Dim	Distribution	Size
LOGO	2M	10K	100	9	Normal	132M
SWDC	8.6M	516K	16.7	50	Power-law	3.3G
LWDC	602M	48.9M	12.3	50	Power-law	219G

Table 4: Datasets statistics

vectors of logo images, and we use this dataset in the experiment to test the performance in finding joinable image columns. The size of the LOGO dataset is 182MB which can fit in memory, and we used it for experiments on in-memory processing.

- **WDC**: the WDC Web Table Corpus 2015 dataset with over 233 million Web tables from [32]. We extracted the English-Language Relational Web Tables 2015 and removed the columns with empty values. For each column, we used the GloVe embedding [13] to transform it into a set of 50-dimensional vectors. For each record in a column, we split the strings into words and took the average embedded vector. We extracted a part of the transformed dataset as **SWDC** (small WDC, 3.3GB) which contained 516,000 columns and 8,650,000 vectors for the experiments on in-memory processing. We used the whole transformed dataset as **LWDC** (large WDC, 219GB) which contained 48,965,123 columns and 602,432,530 vectors for experiments on out-of-core processing.

Methods. All existing methods for searching for joinable tables are focused on the equi-join of two string records without similarities. These methods can not process multi-dimensional vectors in metric space. In the experiment, we implemented state-of-the-art pivot-based methods and adapted them to solve the joinable table search problem; then, we used them as baselines and compared them with our proposed method.

- **EPT**: a baseline method with a pivot table [29]. We built an EPT index for all vectors and issued multiple range search for query vectors. Then, the match number for each column was summarized for determining whether a column is joinable or not. [3]
- **HGrid**: a baseline method used the blocking phase in Algorithm 1. It simply verified the block results one by one. Compared with PEXESO, it does not have the quick browse inverted index function and inverted index-based verification in Algorithm 2.
- **PQ**: an approximate method with the PQ (Product Quantization) [17, 24]. PQ approximately computed the distance table for all vectors, and then counted the matched vectors and retrieved the joinable tables.
- **PEXESO**: our proposed PEXESO method in Algorithm 3.
- **EPTFOR/HGFOR/PEXFOR**: the forest version of the above algorithm for experiments on out-of-core processing with large-scale datasets. The datasets ware partitioned with the KL-clustering in Section 4.2 and each part was indexed into a single EPT, HGrid, and PEXESO index. When processing, we read each index into memory and processed the joinable table search, and then merged the results as the final output.

Environments and measures. The experiments were run on a server with a 2.20GHz Intel Xeon CPU E7-8890 and 630GB RAM. All methods were implemented in Python 3.7. We randomly selected 100 columns from the original datasets as the queries and reported the average processing time. We also observed the memory cost and distance computation times. We use Euclidean distance in the experiments. However, our proposed method can support any metric distance. We used the sim_{om} as the default column similarity. Default τ was 6% of the maximum similarity value of $d(.)$ in the metric space, and default T was 60% of the maximum value of the column similarity function $sim(.)$. The optimal parameters of the pivot number $|P|$ and the level number m for every dataset are tuned in Section 5.3 and used as the default setting.

[3] The reason we selected EPT as a baseline method is because [7] studied the experiments of state-of-the-art pivot-based methods and concluded that EPT is a comparative method in most cases.

5.2 Effectiveness

We first test the effectiveness of the similarity join on the multi-dimensional embedded vectors with the following two ML tasks. We used left-join to preserve original samples.

Airbnb price prediction [1]. We used the NYC Airbnb table [1] and a small data lake with five NYC house sales tables [30]. Airbnb table contains 48,895 records of NYC Airbnb information and the data lake contains five tables with total 66,771 records of the house sale information in five areas of NYC. The "neighborhood" columns in Airbnb table and house sales tables contain the area names, but some of them are in different notations such as "castle hill" and "castle hill/unionport". We took a sample of 1000 records from the Airbnb table and searched joinable tables from house sales tables. For similarity join, we embedded the "neighborhood" column as a column of multi-dimensional vectors. We tuned the parameters with $T = 40\%$ and $\tau = 20\%$ to avoid overfitting. We observed that the word embedding can not only match the pairs with different notation but also match with the semantics, e.g. "bronxdale" and "edenwald" were closed in embedding space. For equi-join, we joined the string values in "neighborhood" when they are exactly the same, and it could only match about 8% records of each table. We trained a Linear Regression model with the joined table to predict the Airbnb price. Table 5a shows the RMSE (Root mean square error) of the test data with the no-join, equi-join, and sim-join methods. The "match #" reacts that sim-join found 5x related records than equi-join. Sim-join had +2.65% performance gain than no-join, and had +3.25% performance gain than equi-join. On the other hand, equi-join reduce the performance by 0.62% than no-join.

Company classification [8]. In the company information table from [8], there are 73,935 companies with 13 classes of categories (Professional Services, Healthcare, etc.). We took a sample of 1000 records as the query table and specified the "company_name" column to find joinable tables from the large data lake of the SWDC dataset [32]. For similarity join of our approach, we embedded the "company_name" column as a column of multi-dimensional vectors, and searched joinable tables with a tuned setting of $\tau = 6\%$ and $T = 0.76\%$. T was set to be small since the query column has 1000 records but the average record number of tables in the SWDC dataset is only 16.7. For equi-join, we searched the tables which have the exact same company names with the query column. We observed that the equi-join approach can only match up to 0.13% of records in the SWDC dataset. We joined all the retrieved tables with the query table. To avoid overfitting, we used a simple feature selection that filtering the columns with the cardinality value less than a tuned threshold, we also aggregated the values of the columns with similar

Method	match #	RMSE	Lift by no-join	Lift by equi-join
no-join	-	221.82	-	-
equi-join	8%	223.20	-0.62%	-
sim-join	**40%**	**215.95**	**+2.65%**	**+3.25%**

(a) Airbnb price prediction.

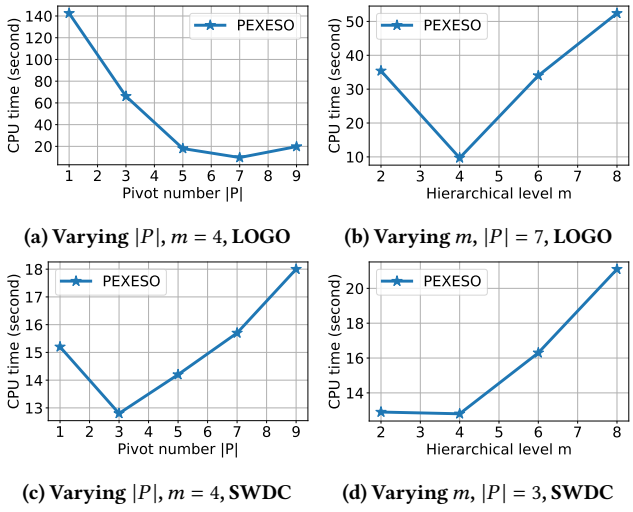
Method	match #	micro-F1	Lift by no-join	Lift by equi-join
no-join	-	0.825 ± 0.057	-	-
equi-join	0.13%	0.806 ± 0.069	-2.30%	-
sim-join	**0.76%**	**0.855 ± 0.045**	**+3.64%**	**+6.08%**

(b) Company classification.

Table 5: Effectiveness of joinable table search with different join policies for two ML tasks.

$\|P\|$	m	LOGO Time(s) index	LOGO Time(s) search	SWDC Time(s) index	SWDC Time(s) search
1	2	56.2	223.5	301.6	16.4
1	4	64.1	142.5	302.9	15.2
1	6	66.9	119.2	315.2	15.2
1	8	58.2	136.9	301.9	16.0
3	2	77.9	145.9	412.3	12.9
3	4	82.6	66.1	421.0	12.8
3	6	81.7	49.7	451.9	16.3
3	8	89.7	47.6	507.1	21.1
5	2	83.7	68.7	448.5	12.7
5	4	78.8	18.0	468.3	14.2
5	6	127.9	21.8	520.8	18.4
5	8	37.5	98.6	595.5	23.2
7	2	79.9	35.4	518.0	14.8
7	4	102.6	9.7	568.3	15.7
7	6	347.2	34.0	619.3	17.9
7	8	665.7	52.4	695.6	20.0
9	2	88.5	24.3	571.0	18.0
9	4	163.0	19.8	610.7	18.0
9	6	499.8	37.9	690.6	21.3
9	8	865.3	45.4	758.4	22.3

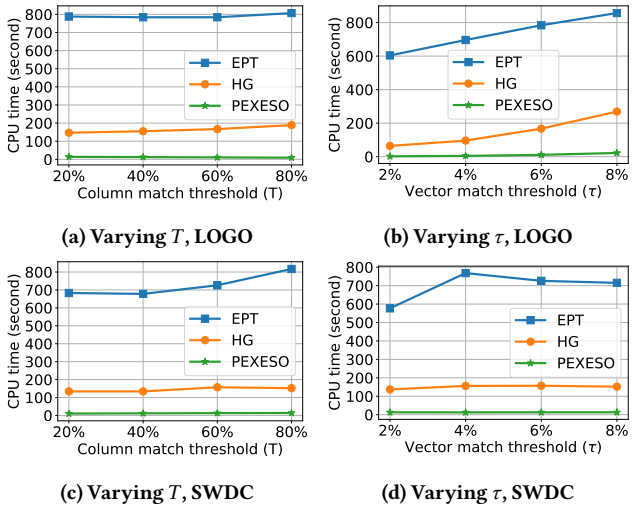
Table 6: Parameter turning for PEXESO: varying $|P|$ and m for LOGO and SWDC datasets.



(a) Varying $|P|$, $m = 4$, LOGO **(b) Varying m, $|P| = 7$, LOGO**

(c) Varying $|P|$, $m = 4$, SWDC **(d) Varying m, $|P| = 3$, SWDC**

Figure 7: A view of the best case of PEXESO in parameter tuning with the LOGO and SWDC datasets



(a) Varying T, LOGO **(b) Varying τ, LOGO**

(c) Varying T, SWDC **(d) Varying τ, SWDC**

Figure 8: In memory processing with varying T and τ under default setting with the LOGO and SWDC datasets.

column names as a single column. We took 4-fold cross-validation that trained the joined tables of our approach and equi-join approach, and the original table without joining, with a Random Forest model and reported the average micro-F1 scores of the test data. According to the results in Table 5b, sim-join has the highest F1 score with +6.08% performance gain than equi-join and +3.64% performance gain than no-join. We also found that the equi-join approach degraded the performance by 2.30% than no-join, this is because it could only match a few records and made the joined table sparse, which easily leads to overfitting.

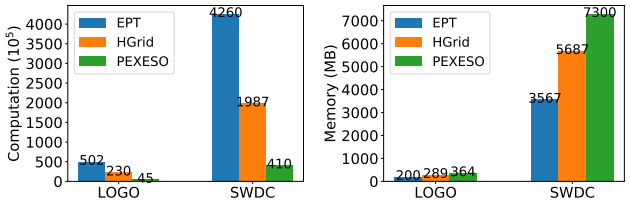
5.3 Parameter Tuning

Before testing the efficiency, we tuned the optimal parameters for all methods for all datasets, and we used their optimal parameters in the rest of the experiments. Due to page limitations, we only report the results of parameter tuning with PEXESO. For PEXESO, there are two parameters, pivot number $|P|$ and level m of the hierarchical grid. Table 6 shows the performance results with different $|P|$ and m. The optimal parameters were $|P| = 7$ and $m = 4$ for the LOGO dataset, and $|P| = 3$, $m = 4$ for the SWDC dataset. We also plot the processing time in Figure 7 to give a view of the trends on varying the parameters.

Varying pivot number $|P|$. Figures 7a and 7c show the search time with varying $|P|$. The searching time of PEXESO first dropped and then increased as $|P|$ increased. This is because more pivots make PEXESO filter more vectors, but also lead to more computations overhead with pivot filtering and matching.

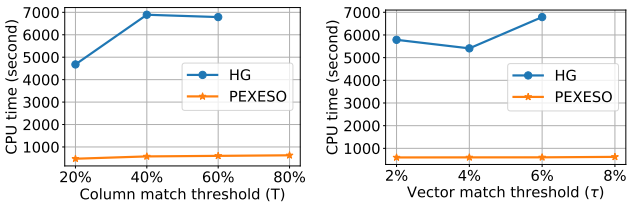
		LOGO Search Time(s)						SWDC Search Time(s)						LWDC Search Time(s)					
		EPT		HGrid		PEXESO		EPT		HGrid		PEXESO		EPTFOR		HGFOR		PEXFOR	
T	τ	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}	s_{om}	s_{mm}
20%	2%	610	612	65.4	125	2.5	7.9	691	679	130	245	9.8	26.5	NA	NA	3567	NA	456	1023
20%	4%	694	702	78.6	176	5.9	13.4	739	786	131	265	10.2	29.4	NA	NA	4156	NA	468	1135
20%	6%	788	787	147	204	13.7	26.8	683	724	134	278	10.2	31.6	NA	NA	4678	NA	475	1267
20%	8%	873	899	234	349	27.5	49.8	696	765	133	281	10.6	37.6	NA	NA	4532	NA	474	1198
40%	2%	611	612	56.7	124	2.3	5.6	642	641	136	341	13.6	26.4	NA	NA	5678	NA	514	1245
40%	4%	693	690	89.5	198	5.4	10.2	655	652	140	298	13.6	29.9	NA	NA	5895	NA	556	1356
40%	6%	784	785	155	287	12.4	19.0	678	720	134	312	11.6	31.0	NA	NA	6892	NA	578	1322
40%	8%	867	872	267	367	24.0	35.9	672	712	143	324	12.0	33.2	NA	NA	6245	NA	602	1355
60%	2%	604	625	64.8	145	2.2	4.5	577	601	137	310	12.8	25.7	NA	NA	5786	NA	598	1402
60%	4%	696	708	96.4	198	4.6	8.7	768	612	156	322	12.5	27.8	NA	NA	5409	NA	601	1412
60%	6%	784	794	167	245	10.8	22.5	726	725	157	309	12.8	31.2	NA	NA	6789	NA	603	1482
60%	8%	857	877	269	398	22.1	39.0	715	814	150	311	13.0	32.5	NA	NA	NA	NA	623	1392
80%	2%	612	618	71.3	154	2.1	7.9	809	825	138	298	13.2	27.5	NA	NA	6157	NA	635	1322
80%	4%	680	702	97.8	214	4.4	14.5	823	856	134	301	13.4	29.8	NA	NA	6245	NA	622	1309
80%	6%	807	789	189	309	9.1	22.5	817	866	152	314	13.4	31.0	NA	NA	NA	NA	627	1444
80%	8%	813	823	256	412	18.1	45.6	829	876	157	326	13.6	30.5	NA	NA	NA	NA	628	1425

Table 7: Search performance in seconds on varying T and τ with LOGO, SWDC datasets (in-memory processing) and LWDC dataset (out-of-core processing). NA means processing time was more than 2 hours.



(a) Computation times **(b) Memory cost**

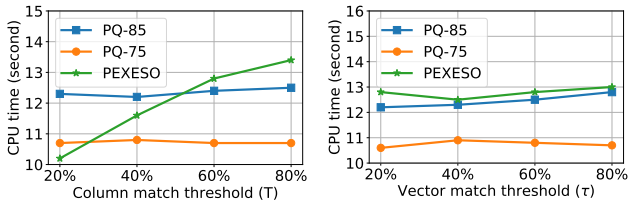
Figure 9: Computation times and memory cost for all methods under default setting with LOGO and SWDC datasets



(a) Varying T **(b) Varying τ**

Figure 10: Out-of-core processing with varying T and τ under default setting with LWDC dataset.

Varying hierarchical grid level m. Figures 7b and 7d show the processing time with varying m. Both the LOGO and SWDC datasets had the minimum search time when $m = 4$, and the search time increased with a larger m. This is because more levels in the hierarchical grid can cause pivot filtering or matching on more vectors, but also lead to more overhead of the traversal with blocking in Algorithm 1 and more inverted index access in Algorithm 2. We also



(a) Varying T **(b) Varying τ**

Figure 11: Compare PEXESO with approximate method PQ with varying T and τ under default setting with SWDC dataset.

computed the theoretical optimal m by minimizing the cost model in Section 3.5. We observed the histogram of each dimension and defined a distribution for fitting. The LOGO dataset was approximate to a normal distribution, and the SWDC was approximate to a power-law distribution (also confirmed by [46]). We input the other variables and minimized the Equation 3 to find a theoretical optimal value of m. The theoretical value for the LOGO dataset is 3.4 and for SWDC dataset is 3.7. They were close to the experimental value $m = 4$; hence, we conclude that the theoretical analysis in Section 3.5 was effective and can help PEXESO with construction.

Indexing time. $|P|$ and m also affected the construction time of PEXESO. According to Table 6, the indexing time increased linearly as $|P|$ and m increased. The indexing time for the optimal parameters with PEXESO was very fast: 102.6 seconds for the LOGO dataset and 421 seconds for the SWDC dataset.

5.4 Efficiency

Performance on in-memory processing. The left side of Table 7 summarizes the performance results for in-memory processing with all methods and the LOGO and SWDC datasets. We also plot the processing time in Figure 8 to show the trends. We can see that PEXESO was the best method in all cases and for all datasets. PEXESO was 45 to 300 times faster than EPT and 10 to 30 times faster than HGrid.

Varying T. Figures 8a and 8c show the comparison results for the real datasets with a different column similarity threshold T. According to the results, the processing time of both HGrid and EPT linear increased with the large threshold T, because a large T requires checking more pairwise objects during a search. However, benefiting from the reverse filtering technique in Lemma 7, the processing of PEXESO was stable.

Varying τ. Figures 8b, 8d show the comparison results for the real datasets with a different distance threshold τ for vector matching. EPT had a linearly increasing processing time with large search range τ since EPT checks and verifies the vectors in τ linearly with a pivot table. HGrid and PEXESO were stable since they have a hierarchical grid-based blocking that can filter (match) multiple vectors at one time with branch-and-bound blocking. On the other hand, a large τ reduces the number of filtering of vectors but increases the number of matching of vectors, so PEXESO and HGrid can perform well in any range of τ.

Column similarity functions. Table 7 also reports the difference in processing time between sim_{om} and sim_{mm}. Due to space limitation, we abbreviate sim_{om} and sim_{mm} as s_{om} and s_{mm}, respectively. We can see that sim_{mm} always took more time than sim_{om} since it needed to check all match vectors between the query and target columns, while sim_{om} only counts each query vector in Q once. Nevertheless, PEXESO was the best method in terms of both sim_{om} and sim_{mm} column match functions.

Computation times. We also observe the similarity (l2 distance) computation times for all methods and report them in Figure 9a. Satisfied with the comparison results of search time, PEXESO had the least on similarity computation, and EPT had the most. PEXESO had less computation than HGrid since it can early stop with an inverted index-based verification.

Memory cost. Figure 9b shows the memory cost of all methods. EPT had the lowest memory cost, and PEXESO had the highest. PEXESO has a more sophisticated structure than other baselines that makes it require more space for indexing. However, the values of the memory cost for all methods were at the same magnitude. It is worth using a little more space and gaining a larger speedup. For those extremely large data lakes like the LWDC dataset, we can use a PEXESO forest and carry on out-of-core processing.

Performance on out-of-core processing. Even though our server has sufficient memory, we simulated the out-of-core processing on the LWDC dataset. We partitioned the LWDC dataset into 10 partitions with the KL-clustering in Section 4.2, and each group was indexed into a single EPT, HGrid and PEXESO index. Then, we assembled all indices with EPTFOR, HGFOR, and PEXFOR for the index forests. The right side of Table 7 and Figure 10 report the performance results, which includes the overhead of reading indices from disk into memory. Notice that we only report and plot the results for processing time of less than 2 hours (7200 seconds). PEXFOR was the fastest method and is significantly outperformed baseline methods, and it was also the only method that could finish all cases of this extremely large LWDC dataset in one hour.

Comparison with approximate method. We compared PEXESO to an approximate method of product quantization (PQ) [17, 24]. For fair compare, we used the pure python implementation of *nanopq* [41]. We adjust the sub-vector numbers in PQ with PQ-75 and PQ-85 to make the recall achieved at least 75% and 85%. Figure 11 shows the comparison results on the SWDC dataset. PEXESO had the same magnitude of processing time and was even faster than approximate methods when T was small. In conclusion, PQ is a better choice if we want to quickly retrieve the tables with a large column similarity threshold T and do not care very much about the recall of the results. Otherwise, PEXESO is a better choice on small T with 100% recall and has competitive processing time.

6 RELATED WORK

Joinable table search. Zhu *et al.* studied the problem of searching for joinable tables with string data by computing the overlap between columns, and the string records were equi-joined [44, 46]. As we introduced in Section 1, they cannot solve our problem because they focus on string values and an equi-join policy.

Set similarity search and join. Regarding the problem of the set similarity join (also called all pair set similarity search), Xiao *et al.* proposed the pp-join [2, 39]. Deng *et al.* proposed a partition-based method [10]. Wang *et al.* designed a similarity that combines token and characters and proposed a solution for the fuzzy join [33, 34]. Deng *et al.* [9] also proposed a related set (table) search system that finds sets with the maximum bipartite matching metrics. Wang *et al.* proposed MF-join [35], which performs a fuzzy match with multi-level filtering. We also refer readers to [22] for various problem settings and methods. The above works are not designed for data lakes (see statements in [44]), and consider only strings. Therefore, they cannot solve our problem with multi-dimensional vectors.

Finding related tables. There is also research on finding related tables with different criteria (not for joinable). Nargesian *et al.* [26] proposed min-hash and sim-hash based techniques for searching unionable tables from data lakes. Zhu *et al.* studied the Auto-join [45], which joins two tables with string-to-string transformation on columns. He *et al.* proposed SEMA-join [14] to find related pairs between two tables with the statistical correlation computed with a big table corpus. Zhang *et al.* studied to find related tables with a composite score of different similarities [42, 43]. Bogatu *et al.* [3] proposed a weighted sum scores function with five attributes of a table, and studied on finding top-k tables according to that score function from data lakes.

Metric similarity search and join. Yu *et al.* [40] applied the iDistance technique to the kNN join problem. Fredriksson *et al.* [12] improved the quick join algorithm [15] for similarity joins. We also refer readers to [6, 7] for various pivot-based indices, and methods. However, since the above methods are designed for the problem of similarity join rather than our joinable search problem, they can

not deal with our joinable searching problem for three reasons: (1) only work with specific cases and require heavy index rebuilding for other cases, (2) Pre-build sophisticated indices for both sets, and (3) One time join without indexing.

7 CONCLUSION

In this paper, we investigated a novel problem of searching for joinable tables with multi-dimensional vectors in data lakes. We proposed a framework, PEXESO, that can reduce expensive distance computations by efficiently pivot-based filtering and matching vectors in a block-and-verify method. We also boosted the PEXESO framework and proposed the PEXESO forest for processing the large-scale tables. We conducted extensive experiments on real datasets. Experimental results showed that PEXESO outperformed the baseline methods with a speed-up of up to 300 times, and sometimes even better than the approximate method. The experiments on effectiveness also showed that PEXESO can find more related tables, and the retrieved tables can help with building better ML models than the tables retrieved by the conventional equi-join policy.

For future work, we would like to extend PEXESO in consideration of the unionable tables in data lakes, and build an end-to-end table fusion system. We also target to extend ML feature selection algorithms in PEXESO.

ACKNOWLEDGEMENT

We'd like to thank our colleague Genki Kusano, NEC corporation, for the daily discussions and the proofread of this paper.

REFERENCES

[1] Airbnb. 2019. NYC Airbnb Open Data, Kaggle. https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data.
[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140. https://doi.org/10.1145/1242572.1242591
[3] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou. 2020. Dataset Discovery in Data Lakes. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 709–720.
[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*. 237–246. https://doi.org/10.1145/170035.170075
[5] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*. 426–434. https://doi.org/10.1145/956863.956944
[6] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. 2017. Efficient Metric Indexing for Similarity Search and Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 556–571. https://doi.org/10.1109/TKDE.2015.2506556
[7] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based Metric Indexing. *PVLDB* 10, 10 (2017), 1058–1069. https://doi.org/10.14778/3115404.3115411
[8] Kaggle Company classification. 2020. Company classification. https://www.kaggle.com/charanpuvvala/company-classification.
[9] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. 2017. SilkMoth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints. *PVLDB* 10, 10 (2017), 1082–1093. https://doi.org/10.14778/3115404.3115413
[10] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *PVLDB* 9, 4 (2015), 360–371. https://doi.org/10.14778/2856318.2856330
[11] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. 2003. Similarity Join in Metric Spaces Using eD-Index. In *DEXA 2003*. 484–493. https://doi.org/10.1007/978-3-540-45227-0_48
[12] Kimmo Fredriksson and Billy Braithwaite. 2013. Quicker Similarity Joins in Metric Spaces. In *SISAP*. 127–140. https://doi.org/10.1007/978-3-642-41062-8_13
[13] GloVe. 2019. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/.
[14] Yeye He, Kris Ganjam, and Xu Chu. 2015. SEMA-JOIN: Joining Semantically-Related Tables Using Big Table Corpora. *PVLDB* 8, 12 (2015), 1358–1369. https://doi.org/10.14778/2824032.2824036
[15] Edwin H. Jacox and Hanan Samet. 2008. Metric space similarity joins. *ACM Trans. Database Syst.* 33, 2 (2008), 7:1–7:38. https://doi.org/10.1145/1366102.1366104
[16] Veit Jahns. 2012. Principles of data integration by Anhai Doan, Alon Halevy, Zachary Ives. *ACM SIGSOFT Software Engineering Notes* 37, 5 (2012), 43. https://doi.org/10.1145/2347696.2347721
[17] H. Jégou, M. Douze, and C. Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.
[18] You Jung Kim and Jignesh M. Patel. 2010. Performance Comparison of the R*-Tree and the Quadtree for kNN and Distance Join Queries. *IEEE Trans. Knowl. Data Eng.* 22, 7 (2010), 1014–1027. https://doi.org/10.1109/TKDE.2009.141
[19] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*. 1969–1984. https://doi.org/10.1145/2723372.2723713
[20] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join?: Thinking Twice about Joins before Feature Selection. In *SIGMOD*. 19–34. https://doi.org/10.1145/2882903.2882952
[21] logdata. 2018. WebLogo-2M Dataset. http://www.eecs.qmul.ac.uk/~hs308/WebLogo-2M.html/.
[22] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB* 9, 9 (2016), 636–647. https://doi.org/10.14778/2947618.2947620
[23] Rui Mao, Willard L. Miranker, and Daniel P. Miranker. 2012. Pivot selection: Dimension reduction for distance-based indexing. *J. Discrete Algorithms* 13 (2012), 32–46. https://doi.org/10.1016/j.jda.2011.10.004
[24] Yusuke Matsui, Yusuke Uchida, Herve Jegou, and Shinichi Satoh. 2018. A Survey of Product Quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10. https://doi.org/10.3169/mta.6.2
[25] Emanuel Zgraggen Raul Castro Fernandez Tim Kraska David Karger Nadiia Chepurko, Ryan Marcus. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. In *arxiv*.
[26] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *PVLDB* 11, 7 (2018), 813–825. https://doi.org/10.14778/3192965.3192973
[27] Rodrigo Paredes and Nora Reyes. 2009. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *J. Discrete Algorithms* 7, 1 (2009), 18–35. https://doi.org/10.1016/j.jda.2008.09.012
[28] Spencer S. Pearson and Yasin N. Silva. 2014. Index-Based R-S Similarity Joins. In *SISAP*. 106–112. https://doi.org/10.1007/978-3-319-11988-5_10
[29] Guillermo Ruiz, Francisco Santoyo, Edgar Chávez, Karina Figueroa, and Eric Sadit Tellez. 2013. Extreme Pivots for Faster Metric Indexes. In *SISAP*. 115–126. https://doi.org/10.1007/978-3-642-41062-8_12
[30] NYC House sales. 2019. NYC House sales with 5 blocks. https://www1.nyc.gov/site/finance/taxes/property-rolling-sales-data.page.
[31] Vraj Shah, Arun Kumar, and Xiaojin Zhu. 2017. Are Key-Foreign Key Joins Safe to Avoid when Learning High-Capacity Classifiers? *PVLDB* 11, 3 (2017), 366–379. https://doi.org/10.14778/3157794.3157804
[32] WDC Web Table. 2015. WDC Web Table Corpus 2015. https://https://en.wikipedia.org/wiki/Jensen-Shannon_divergence.
[33] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2011. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*. 458–469. https://doi.org/10.1109/ICDE.2011.5767865
[34] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2014. Extending string similarity join to tolerant fuzzy token matching. *ACM Trans. Database Syst.* 39, 1 (2014), 7:1–7:45. https://doi.org/10.1145/2535628
[35] Jin Wang, Chunbin Lin, and Carlo Zaniolo. 2019. MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering. In *ICDE*. 386–397. https://doi.org/10.1109/ICDE.2019.00042
[36] Wikipedia. 2020. Jensen–Shannon divergence divergence. https://https://en.wikipedia.org/wiki/Jensen-Shannon_divergence.
[37] Wikipedia. 2020. Kullback–Leibler divergence. https://en.wikipedia.org/wiki/Kullback-Leibler_divergence.
[38] Wikipedia. 2020. Lloyd's algorithm. https://en.wikipedia.org/wiki/Lloyd%27s_algorithm.
[39] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15:1–15:41. https://doi.org/10.1145/2000824.2000825
[40] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Information & Software Technology* 49, 4 (2007), 332–344. https://doi.org/10.1016/j.infsof.2006.05.006
[41] Github Yusuke Matsui. 2020. nanopq. https://github.com/matsui528/nanopq.
[42] Yi Zhang and Zachary G. Ives. 2019. Finding Related Tables in the Data Lake.
[43] Yi Zhang and Zachary G. Ives. 2019. Juneau: Data Lake Management for Jupyter. *Proc. VLDB Endow.* 12, 12 (2019), 1902–1905. https://doi.org/10.14778/3352063.

3352095

[44] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864. https://doi.org/10.1145/3299869.3300065

[45] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-Join: Joining Tables by Leveraging Transformations. *PVLDB* 10, 10 (2017), 1034–1045. https://doi.org/10.14778/3115404.3115409

[46] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *PVLDB* 9, 12 (2016), 1185–1196. https://doi.org/10.14778/2994509.2994534