Università di Genova

Dipartimento di
Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi

Table Augmentation in Data Lakes

Federico Dassereto

Theses Series DIBRIS-TH-2023-42

Ph.D. Thesis

Università di Genova
Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi
Ph.D. Thesis in
Computer Science and System Engineering
Computer Science Curriculum

Table Augmentation in Data Lakes

by

Federico Dassereto

April 2023

Ph.D. Thesis in Computer Science and System Engineering (S.S.D. INF/01)
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova

Candidate
Federico Dassereto
federico.dassereto@edu.unige.it

Title
Table Augmentation in Data Lakes

Advisors
Giovanna Guerrini
DIBRIS, Università di Genova
giovanna.guerrini@unige.it

External Reviewers
Reviewer 1,
Dipartimento di Informatica - Scienzae Ingegneria, Università di Bologna,
reviewer1@unige.it

Reviewer 2,
School of Mathematics and Computer Science, Heriot-Watt University,
reviewer1@unige.it

Location
DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy

Submitted On
April 2023

Abstract

Data lakes are centralized repositories that store large quantities of raw, unstructured, and structured data, allowing for ad-hoc data analysis, exploratory data analysis, and machine learning. However, the lack of metadata and schema in data lakes makes it challenging to work with tabular data and find related information stored in different tables. However, it is still an open problem how efficiently retrieve these tables at large scale when the settings of a data lake holds. The thesis introduces a novel approach to table augmentation that enables efficient data integration from multiple sources in a data lake. Table augmentation involves adding new data to an existing table in a horizontal fashion (by retrieving tables that can be horizontally concatenated to a query that serves as query table). The proposed approach consists of several components, including data lakes hashing, join search, similarity, and augmentation. The proposed approach is named TASH. TASH is a framework based on a spatial index in which tables are mapped and queried. Its goal is to identify the most useful columns for subsequent machine learning tasks. The table retrieval process employs a combination of set containment search and similarity search. Candidate tables are initially identified using set containment search and then ranked based on their similarity to the query. Experimental results demonstrate that TASH can effectively identify joinable tables and select the most relevant features, thereby enabling efficient table augmentation in data lakes. This research contributes to the field of big data by providing a practical solution to the challenges of data integration and analysis in data lake environments.

Acknowledgements

Contents

1

Introduction

In the era of big data, the ability to efficiently manage and process vast amounts of information has become crucial. One approach to this problem is the use of data lakes, which allow for the storage of vast quantities of raw data in a flexible and scalable manner.

A data lake is a centralized repository that allows the storage of large amounts of raw, unstructured, and structured data. The basic principle of a data lake is to store data in its native format without any pre-processing, such as schema or metadata definitions. The goal of a data lake is to enable researchers to perform ad-hoc data analysis, exploratory data analysis, and machine learning on raw data in a cost-effective manner.

In a data lake, data is stored in its native format without any pre-processing or transformation [72]. Tabular data is one such format that is commonly found in data lakes. In the absence of metadata and schema, tabular data can be challenging to work with, as the structure of the data is not readily apparent. The lack of metadata and schema information can make it difficult to understand the meaning of the data, which can lead to incorrect analysis results. However, there are several techniques that can be used to work with tabular data in a data lake environment.

One such technique is to use schema-on-read, where the structure of the data is inferred when the data is read. Schema-on-read allows researchers to explore the data without the need for a predefined schema. Another technique is to use data profiling to understand the structure and content of the data. Data profiling involves analyzing the data to extract statistics, such as the number of null values, unique values, and data types, to gain insights into the data's structure.

Data lakes are becoming increasingly popular as a cost-effective way to store and process large amounts of data. One of the reasons of this increasing popularity relies indeed on the highlighted feature that, unlike traditional databases where the schema is fixed, data lakes allow users to store data in various formats and structures. However, the flexibility of data lakes also makes it challenging to find related information stored in different tables as well as to identify the relationships among data in different tables, which can hinder data analysis and integration.

Finding related information in a data lake is a non-trivial task, since data in data lakes is often stored in various formats, including structured, semi-structured, and unstructured data. The data may also be stored in different file

formats, such as CSV, JSON, or Parquet. In addition, data lakes typically lack a centralized schema. Without a clearly defined schema, it may be challenging to determine which columns in different tables represent the same data.

Another challenge in finding related information in a data lake is that data is often stored in multiple locations. Data can be stored in different buckets or directories, making it difficult to find all the relevant tables. Furthermore, data can be replicated or duplicated, further increasing the complexity.

However, the usefulness of data lakes is heavily dependent on the ability to extract insights from the stored data. One of the key challenges in this regard is table augmentation, which involves finding relevant data across multiple tables and combining it to form a more complete dataset.

1.1 The Problem: Table Augmentation

Table augmentation refers to the process of adding new data to an existing table, in order to increase its size or expand its scope [67]. There are two main types of table augmentation: horizontal and vertical. Horizontal table augmentation [30] involves adding new columns to an existing table, typically by joining it with another table based on a shared key. This type of augmentation is often used in data analysis and machine learning applications, where it can help to improve model accuracy by incorporating additional features or attributes. By contrast, vertical augmentation [68] involves adding new rows to an existing table, typically by synthesizing new data points based on existing data. In both cases, the idea is to retrieve tables that allow to augment the content of the current table with related information either since they are *joinable*, thereby adding new columns, (in the case of horizontal augmentation) or can be *unioned*, thereby adding new rows, (in the case of vertical augmentation).

Horizontal Augmentation. One of the key benefits of horizontal augmentation is that it allows for the efficient joining of large datasets, which can be time-consuming and resource-intensive using traditional SQL join operations. By indexing the columns in both tables and using hash-based similarity search algorithms, it is possible to quickly identify and retrieve matching rows from each table, enabling the creation of a new table with combined data. Another important aspect of horizontal augmentation is the need for consistent metadata and schema across tables. Since each table represents a unique set of attributes or features, it is critical to ensure that the columns being joined are semantically equivalent and compatible with one another. In cases where the metadata or schema are inconsistent or missing, additional preprocessing steps may be required to transform the data into a common format before it can be effectively joined. This type of augmentation is often used in machine learning and data generation applications, where it can help to increase the size of the training dataset and improve model performance. However, creating new rows synthetically also introduces additional complexity and potential bias into the data, as the generated rows may not accurately represent the underlying distribution of the data.

To address these issues, several techniques have been developed for synthetic data generation, including generative adversarial networks (GANs) [39, 63] and variational autoencoders (VAEs) [69]. These methods use deep learning algorithms to learn the underlying distribution of the data and generate new samples that are statistically similar to the original data. However, care must be taken to ensure that the synthetic data is representative of the true distribution, and that any biases or limitations in the original data are not amplified or perpetuated by the augmentation process.

Overall, table augmentation is a powerful technique for expanding and enhancing existing datasets, enabling more accurate and robust data analysis and machine learning models. However, it requires careful attention to metadata and schema consistency, as well as the potential biases and limitations introduced by synthetic data generation. As the field of data science continues to evolve and expand, new techniques and approaches for data augmentation will undoubtedly emerge, enabling even more powerful and effective data analysis and modeling.

Vertical Augmentation. Vertical table augmentation, on the other hand, involves adding new rows to an existing table. This can be done by collecting new data from external sources, or by synthesizing new data based on existing data within the table. Vertical augmentation is often used in data generation and data cleaning applications, where it can help to ensure that a dataset is representative of the underlying population. Vertical augmentation can be performed in different ways, one of which involves vertically merging the tables. This technique can be used when we have multiple tables with exact or overlapping columns, and we want to combine them into a single table with more rows. By doing so, we can create a larger dataset that contains more instances of the same attributes, which can help to improve the performance of machine learning models.

The process of vertical augmentation involves appending rows from one table to another, rather than adding new columns. This is different from horizontal augmentation, which involves adding new columns to a table. When we perform vertical augmentation, we need to ensure that the tables we are merging have the same structure, with the same number and types of columns. One way to perform vertical augmentation is to use the SQL UNION operator. The UNION operator allows us to combine the results of two or more SELECT statements into a single result set. When we use the UNION operator, the columns of the result set are determined by the first SELECT statement, and the types of the columns must be compatible with the columns in the subsequent SELECT statements.

Since we have no metadata nor schemas, one approach to identifying tables with overlapping columns is to use a hashing-based similarity search, similar to what we discussed earlier for horizontal table augmentation. By representing the columns of each table as hashes and comparing the hashes across tables, we can identify tables with similar or identical columns. Once we have identified these tables, we can perform a vertical merge by stacking the rows of the tables

on top of each other.

In addition to using similarity search, we can also use data profiling techniques to identify and merge tables with overlapping columns. Data profiling involves analyzing the content and structure of the data in a table, such as the data types and values of the columns. By comparing the content and structure of multiple tables, we can identify which tables have overlapping columns and merge them accordingly. Another approach to vertical augmentation in a data lake is to synthesize new data based on existing data within the tables. This can be done using various data generation techniques, such as sampling, interpolation, or extrapolation. For example, we can use statistical techniques to generate new data points based on the distribution of existing data within a table.

Synthesizing new rows based on existing data is a common technique in data generation, particularly in cases where the amount of available data is limited or the data is subject to privacy constraints. One popular approach to data synthesis is known as generative adversarial networks (GANs), which use a neural network to generate new samples that are similar to the existing data. This technique has been successfully used in a variety of applications, from generating realistic images to synthesizing patient data for medical research. However, generating new data hides its challenges as well. One issue is ensuring that the synthetic data accurately reflects the distribution of the underlying population. This is particularly important in cases where the data will be used for predictive modeling or decision-making, as inaccurate data can lead to biased results. To address this, researchers have developed a number of techniques for evaluating the quality of synthetic data, such as comparing summary statistics or using statistical tests to compare the synthetic and real data.

Another challenge is preserving privacy when generating synthetic data. This is particularly important in cases where the original data contains sensitive information, such as medical or financial data. Techniques such as differential privacy can be used to ensure that the synthetic data does not reveal any sensitive information about individuals in the dataset. In addition to generating new data, vertical table augmentation can also be used to clean and enhance existing data. For example, missing data can be imputed using techniques such as mean imputation or regression imputation, which use the values of other variables to estimate the missing values. Similarly, outliers or other anomalies can be detected and removed using statistical techniques such as clustering or outlier detection.

Overall, vertical table augmentation is an important tool in the data scientist's toolkit, allowing for the creation of new data and the enhancement of existing data. However, it is important to carefully consider the implications of generating synthetic data, particularly in cases where the data will be used for decision-making or where privacy concerns are a factor. By using appropriate techniques for data synthesis and evaluation, data scientists can ensure that the resulting data is accurate, representative, and preserves the privacy of

individuals in the dataset.

The challenge of exact table augmentation. The table augmentation problem is a fundamental and relevant issue that affects many aspects of the database community [67]. From both theoretical and practical perspectives, table augmentation is an essential concept that plays a vital role in database systems. In this work, we will focus on horizontal augmentation.

From a theoretical perspective, the table augmentation problem is significant because it allows us to better understand the relationships between tables [51]. The problem helps us to identify and represent the various dependencies and relationships that exist among different tables. This knowledge is important because it enables us to design more efficient database systems that can handle complex queries and transactions more effectively. In other words, understanding the table augmentation problem can lead to significant advancements in the field of database theory. From a practical perspective, the table augmentation problem is crucial because it is a common issue that affects many real-world use cases. For example, in data analysis, it is often necessary to combine multiple tables to derive meaningful insights. However, the tables may have different schemas or be incomplete, making it challenging to join them effectively. The table augmentation problem provides a framework for addressing this issue, enabling data analysts to create more accurate and complete models.

In addition, the table augmentation problem is relevant in many other fields beyond data analysis, including computer science, machine learning, and artificial intelligence. For instance, in natural language processing, researchers often use tables to represent data, but the tables may be incomplete or have missing values. By addressing the table augmentation problem, researchers can build more robust and accurate models that can handle such data effectively. In theory, one could perform exact table augmentation by computing all possible joins among the tables in a data lake, and then selecting the ones that satisfy certain relevance criteria. However, this approach quickly becomes unfeasible as the number of tables and their size increases. For example, if we have n tables, each with m columns and k rows, the number of possible joins is exponential in n, and even selecting a subset of relevant columns can be computationally expensive.

Moreover, exact approaches are sensitive to changes in the data, as a small modification in one table may require the recomputation of all the joins. This makes it difficult to scale such approaches to large and dynamic data lakes, where new tables may be frequently added or modified. For these reasons, approximate approaches based on hashing and similarity search have gained attention in recent years, as they offer a scalable and flexible alternative to exact table augmentation. By projecting the tables onto a lower-dimensional space using a hash function, we can quickly identify potential joinable tables and relevant columns, without computing all possible joins. The use of similarity search further reduces the computational cost by allowing us to focus only on the most similar pairs of tables, while ignoring those that are unlikely to yield

meaningful results.

We believe that finding joinable tables in data lakes is a significant challenge due to the flexibility and complexity of data storage in data lakes. Existing solutions have limitations, and the approach we propose in this thesis leverages data fingerprints to efficiently identify joinable tables.

1.2 Key Components of the Approach

The approach we propose to face the challenge of integrating data from multiple sources in a data lake, relies in a number of components, listed in what follows, after making clear why we do not evaluate every possible candidate table after performing the join operation (as we will see, this approach is not feasible) and introducing the table representation we design. An important, final, ingredient is represented by the evaluation schema and metrics we implemented.

Is it feasible to materialize a join over all tables? Joins are common operations in data analysis that involve combining data from multiple tables based on a common attribute. In a data lake environment, joins can be challenging due to the lack of a predefined schema and metadata.

Materializing a join over all tables in a large data lake is a computationally expensive operation. This operation involves joining multiple tables that have no defined schema or indexes. It is a common problem encountered in the analysis of big data, where data is stored in a data lake without a clear structure.

The primary challenge of materializing a join over all tables in a data lake is the sheer size of the data. Joining all tables in a data lake can result in a combinatorial explosion, where the number of possible combinations grows exponentially with the number of tables. This can lead to memory overflow, disk I/O overload, and CPU usage bottlenecks.

In addition to the problem of size, there is also the issue of performance. Joining tables without any indexes or metadata can lead to slow query times due to the need for a full table scan. Full table scans can be particularly costly when dealing with large datasets that don't fit in memory, as disk I/O can quickly become a performance bottleneck.

From a theoretical perspective, the problem of materializing a join over all tables in a data lake is significant because it is an NP-complete problem. This means that there is no known efficient algorithm that can solve this problem in polynomial time. Therefore, any solution to this problem must rely on heuristics or approximation algorithms.

From a practical perspective, the problem of materializing a join over all tables in a data lake is relevant to a wide range of industries, including finance, healthcare, and marketing. For example, in the finance industry, analysts may want to join multiple tables to analyze trends in stock prices, trading volumes, and other financial data. In healthcare, researchers may want to join multiple tables to identify correlations between patient demographics and medical conditions. In marketing, analysts may want to join multiple tables to analyze

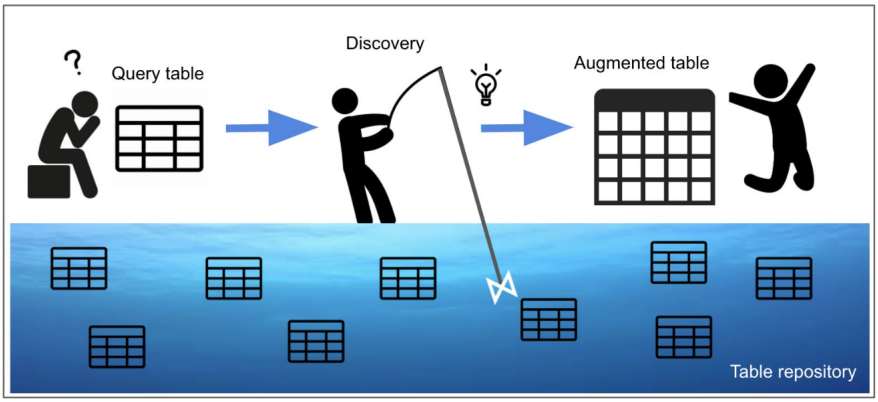


FIGURE 1.1 The table-as-a-query paradigm

customer behavior across different channels.

In conclusion, materializing a join over all tables in a data lake to evaluate table augmentation is a computationally challenging problem. It requires significant computational resources, and there is no known efficient algorithm that can solve it in polynomial time. The problem is relevant to a wide range of industries and has significant practical applications. As such, it is an area of active research in the database community.

Tables representation. In order to efficiently address the computational challenges posed by the materialization of a join over all tables in a data lake, a new approach based on a column-based indexing paradigm can be considered. This approach, referred to as *table-as-a-query* and graphically depicted in Figure 1.1, involves converting each element of each column in the data lake to an integer number and using an index to search for columns rather than tables.

The conversion of the data elements to integer numbers provides several advantages. First, it allows for the efficient comparison and ordering of the data. This is particularly relevant for operations such as sorting and searching, where an ordering of the data is necessary. Moreover, by mapping the data to integer numbers, the memory usage is reduced, since integer numbers require less memory compared to strings or other data types.

In addition to ordering, another advantage of converting the data to integer numbers is the ability to use a vocabulary-based approach. This approach involves creating a dictionary or vocabulary that maps each unique data element in the data lake to a unique integer number. By using a vocabulary, we can efficiently map the data elements to integer numbers, reducing the need for additional memory for storing the mappings.

Hashing. Using an image similarity hashing approach for mapping integers is a powerful technique for simulating pixel intensity. This technique is useful when dealing with large datasets where the columns represent images. In the case of table augmentation, one can convert each element of each column to an integer number and use an image similarity hashing approach to map the

integers.

The use of hashing in image similarity techniques is common in the field of computer vision, where images are represented as sets of features. These features are then hashed using a similarity hashing technique to create a hash that represents the image. Similarly, in our case, we can use the column representation of a table to create a hash that represents the column.

The use of an image similarity hashing technique allows us to consider our integers mapping as input to such hashing, simulating pixel intensity. This technique makes it possible to use image similarity techniques to compare columns from different tables. The similarity metric used in image similarity hashing can be used to measure the similarity between two columns.

One advantage of using hashing is that it provides a compact representation of the data. The hashing function can map the integers to a smaller range of values, reducing the amount of memory required to store the data. Additionally, hashing allows for fast indexing and retrieval of data. This is because the hash function can be used to map the data to a specific location in memory, allowing for fast lookup times.

Another advantage of using hashing for table augmentation is that it allows for efficient set containment searches for joins. One popular technique for set containment searches is the LSH Ensemble approach [92]. This technique can be used to efficiently join tables by performing set containment searches on the hashed columns.

Padding and resampling. When working with column representations, it is important to ensure that the inputs have a fixed length to be able to use hashing algorithms like locality-sensitive hashing (LSH). However, the original data in our tables may not have a fixed length, and this poses a challenge.

One approach to address this challenge is to pad the values with zeros or some other placeholder value to create fixed-length inputs. For example, if we have a column of strings representing names, we can add zeros to the end of each string so that all strings have the same length. While padding ensures fixed-length inputs, it may introduce bias and distort the original distribution of the data. Padding may also lead to increased memory usage if we have many columns with sparse data.

Another approach is to use resampling to create fixed-length inputs. In this approach, we randomly sample a subset of the values from the column to create fixed-length inputs. This can be done using techniques like random sampling or stratified sampling, where we sample values with replacement based on some criteria like the frequency of occurrence. Resampling ensures that the original distribution of the data is preserved, but it may also introduce noise and distort the original values. Moreover, resampling can lead to increased computational costs if the data is very large.

The choice between padding and resampling depends on the specific requirements of the application and the properties of the data. If we care more about fixed-length inputs and the original distribution of the data is not important, then padding may be a better choice. On the other hand, if we want to preserve

the original distribution of the data, then resampling may be a better choice.

Information Theory to Evaluate Table Augmentation. Information theory metrics, such as entropy and mutual information, can be useful to evaluate the quality of table augmentation. Entropy can be used to measure the randomness or uncertainty of a column, while mutual information can quantify the dependence between two columns. These metrics can help in identify columns that are likely to be relevant for joining tables or for machine learning tasks.

When augmenting tables in a data lake, one may encounter many tables with similar column names or similar data, but not all of them may be useful for our purposes. Information theory metrics can help in filtering out irrelevant tables or columns by quantifying their information content and correlation with other tables or columns.

For example, suppose we want to join two tables on a common column. We can use mutual information to identify which other columns in the two tables are most likely to be relevant for the join. By computing the mutual information between the common column and each column in the two tables, we can rank the columns based on their relevance to the join. This can help us select the most relevant columns for the join, while ignoring columns that are irrelevant or redundant.

Similarly, when selecting tables or columns for machine learning tasks, we can use entropy and mutual information to identify the most informative and relevant features. By selecting columns with high entropy and high mutual information with the target variable, we can improve the accuracy and performance of our machine learning models.

Thus, information theory metrics can provide a principled and quantitative way to evaluate table augmentation and feature selection in data lakes. By leveraging these metrics, one can select tables and columns that are most likely to be useful for his purposes, while avoiding and filtering out irrelevant or redundant information.

1.3 Overview of the Approach

To implement our approach, we use a suitable framework based on a KDTree and exploit radius queries for both joining (i.e., identifying points in the radius) and the most useful columns for subsequent machine learning tasks (i.e., identifying columns that are far away from the radius). Our experimental results show that our approach can effectively identify joinable tables and select the most relevant features, thereby enabling efficient table augmentation in data lakes.

Our approach to table retrieval is based on data lakes hashing, join search, similarity, and augmentation. We first convert each column of every table in the data lake to a fixed-length integer vector using a hashing function. The hashing function maps each column element to an integer in a fixed range. We use a hashing function that is commonly used in image similarity, which

simulates pixel intensity. This approach allows us to use algorithms and data structures developed for image similarity to operate on tables.

To perform table retrieval, we use a combination of set containment search and similarity search. Given a query Q, we first use set containment search to find the set of candidate tables that potentially match Q. We then use similarity search to rank the candidate tables based on their similarity to Q. To use similarity search, we represent the query Q as a fixed-length integer vector using the same hashing function used for the data lake tables. We then compute the similarity score between Q and each candidate table using a similarity measure such as cosine similarity.

1.4 Summary of Contributions

The thesis main contributions are:

- The development of a framework that indexes a data lake and allows for the search of joinable tables that improve the predictive ability of a query table.
- The representation of the tables of a data lake with hash fingerprints that captures both the semantic content of each column of the table and keep their values distributions.
- The definition of an algorithm, TASH, that exploits hashing functions, information theoretic measures and spatial indexes to retrieve joinable tables.
- The experimental evaluation of TASH, showing that it can achieve similar performances of state-of-the-art approaches both in identifying joinable tables and ranking them according to their improvement with respect to the performance of a query table in various Machine Learning tasks.

1.5 Outline of the Thesis

The thesis is organized as follows:

- In Chapter 2 we discuss the state-of-the-art in the two fundamental fields dealt by the thesis. We splitted this Chapter in three main parts: similarity and search of tables in data lakes, hashing techniques and augmentation approaches. The Chapter focus on state-of-the-art approaches in each of the depicted categories that individually concur as a key component of our work.
- In Chapter 3 we introduce some preliminaries. In this chapter, we present the basic notions and concepts useful to understand the thesis. We briefly discuss the relational world on top of which data lakes originated, then we discuss the paradigm we follow to design our work. We finally discuss key components of our work, namely information theory, hashing techniques and spatial indexes.

- In Chapter 4 we formalize the problem we are willing to solve and present TASH, our algorithm to provide table augmentation in data lakes. We present many possible base approaches to our solution, discussing the intuitions behind them and highlights their limitations. Finally, we describe in details TASH.
- In Chapter 5 we evaluate our method under different scenarios and comparing with existing methods. We evaluate on two main concepts, the ability to find joinable tables and to retrieve tables that provide augmentation. Finally, we discuss the results emerged from the evaluation.
- In Chapter 6 we conclude our work summarizing the content of the thesis, discussing the limitations of our work and drawing possible future directions for this work.
- In Chapter 7 we present orthogonal work that we carried out in the period of the PhD, presenting pubblications and results that do not directly link to the content of this thesis.

2

Related Work

This thesis has been inspired by a major question: Is it possible to produce a table augmentation, i.e., finding joinable tables with relevant feature columns, without materializing the join operation? To answer this question, we systematically explore the literature to isolate the components of a possible solution. The thesis aims to explore the possibility of a horizontal augmentation for improving the performance of a machine learning model by adding new features to a query table. The approach adopted in this thesis involves two key concepts: *similarity and search* and *hashing*. In this chapter, the related work in each of these three areas is discussed in detail.

The first concept, similarity and search, is explored through existing tools and technologies that enable efficient join and union search in data lakes. These techniques are crucial for identifying relevant data that can be used to augment the query table with new features. The chapter provides an in-depth analysis of the various approaches that have been proposed for similarity search, including methods based on text, image, and graph similarity.

The second concept, hashing, is used to efficiently represent the query table and the new features. The chapter discusses both table and column-oriented hashing techniques and their advantages and disadvantages. The various hashing techniques are analyzed based on their suitability for the proposed approach.

The chapter concludes with a positioning of the proposed approach with respect to the state-of-the-art. It provides a comprehensive overview of the existing work in the three areas of similarity and search, information theory, and hashing, and identifies the most promising techniques for the proposed approach. The proposed approach has the potential to significantly improve the performance of machine learning models by augmenting the query table with new features.

2.1 Similarity and Union Search

One of the greatest novelty of our approach is to consider as input a whole table, in a paradigm called table-as-a-query. Considering a table as a whole query allows taking care of each aspect of the columns all at once and extending the ranking by columns to a complete table. Within this paradigm, in this section, we present the relevant research work on searching tables that are *similar*

according to some similarity measure. Since many of the approaches rely on hashing techniques, these two section are strictly related and connected.

The very starting point was [51] and its extension [76], in which the authors put themselves in a relational scenario and elaborate on the possibility of exploiting measures that suggest whether it is safe or not to avoid a join while doing feature selection. It is worth to notice that they refer to a scenario in which the key-foreign key mapping is explicit and the number of involved relations is modest. Along this way of relying on the schema information, [20] proposed a method to find joinable tables by using the metadata of the tables, involving schema information and key identification, as well as entity recognition to find relevant tables. Despite our problem statement, they do not consider the problem of identifying joinable tables *that contains* relevant columns. This scenario is very typical in many approaches, since many approaches focus on a discovery scenario with an iterative human-in-the-loop situation where the exploration depends on human interactions. Similarity search is not the only task that can be carried on during an exploration, and exhaustive source of other applications such as entity mapping or keyword-based search can be found in [15, 86]. SILKMOTH [24] is proposed as a system capable of rapidly discovering related set pairs in collections of sets. It creates a signature for each set, prunes the search space based on the signatures, and applies the maximum matching metric on remaining candidates to verify relatedness. The authors characterize the space of signatures in a way that guarantee the same output as the brute-force method, as well as the proposal of two filters and a simple optimization to further prune the candidates and improve efficiency.

Focusing on data discovery, in [14] a system called AURUM is described, which is designed to address the problem of data discovery in organizations where analysts spend more time searching for relevant data than analyzing it. AURUM builds and maintains an enterprise knowledge graph (EKG), which captures relationships between datasets and helps users to navigate among disparate sources. On the other hand, [83] proposes a system that performs entity augmentation and attribute discovery by leveraging the vast corpus of HTML tables available on the web. The authors present three core operations to achieve this, with the goal of high precision and coverage, fast response times, and applicability to any arbitrary domain of entities. Along with [83], [54] relies on metadata in addition to table content. ALITE [45] is a proposal for scalable integration of tables discovered using join, union, or related table search. It relaxes previous assumptions that tables share common attribute names, are complete, and have acyclic join patterns. Three new benchmarks for integration using real data lake tables are developed and shared to evaluate the performance of ALITE, which is shown to outperform previous algorithms for computing the Full Disjunction.

A consistent part of the work concerning open data and their integration can be found in [67]. The key idea was to enable data science by exploiting the power of open data and their massive size, posing the basis for internet-scale

algorithm that allow for join and union search.

Union search. One of the first algorithm for table union search was OCTOPUS [10]. The algorithm works with a decomposition of many problems, and it is basically founded on keyword search. They uses search-style keyword queries and returns a ranked list of relevant web tables. They mainly use tf-idf score and mean string length difference to find the similarity between the columns and combine them to infer the table similarity This work has been used as starting point for many other works, such as those originated from Web Tables [52] [73] and [68]. A deep and detailed survey on web tables usage in data integration can be found in [87]. [20] defines the problem of finding unionable web tables as an entity complement problem. Two tables are deemed unionable if they share similar schemas and a "subject" column, which is a single column that contains the entities the table is about. This work assumes that each table has a single subject column, a common trait of web tables, but this is a limiting assumption in data lakes and open data. Also, this approach relies solely on an existing KB. [38] introduces SemProp, which links related tables by creating DAGs of table elements and external sources such as ontologies and embeddings, connected by semantic and syntactic links. Although the goals are similar, we solely leverage the data values in tables whereas SemProp mainly uses the tables' schemas and names to find related tables. [6] proposes an attribute-unionability framework similar to [68] that adds attribute name similarity, regular expression similarity, and distribution similarity to determine the relatedness between tables. We focus on attribute values, and also include relationship and type hierarchy semantics. A recent work on union search is [44], where the authors define union in new way, in which added rows matches semantically through the usage of a knowledge base, inherited and extracted from the data lake itself, where it is easier to produce a semantically meaningful union exploiting the predicates that links entities, that in this case are columns.

Join search. [85] presents algorithms for clustering relational columns into attributes based on common properties and characteristics of the values they contain. These relationships are useful for schema matching and better understanding and working with the data. The algorithms use statistical measures to identify strong relationships between columns and decompose the database into connected components to cluster sets of columns. [61] conducts an extensive analysis of algorithms for set similarity joins. The analysis shows that efficient verification is crucial for the performance of these algorithms. The paper found that most algorithms exhibit similar performance, with the prefix filter being the key technique. In [92], the authors use Jaccard set containment score to measure the relevance of a domain to a query domain. They present an index structure called LSH Ensemble, which uses Minwise Hashing and domain partitioning to cope with data volume and skew. The authors construct a cost model that describes the accuracy of LSH Ensemble and prove that there exists an optimal partitioning for any data distribution. It is worth noticing that Jaccard similarity is expressed on set and not on bag, meaning that

repetition are not considered in the precision results. [91] discusses the development of Auto-Join, a system that can automatically search over a rich space of operators to compose a transformation program, whose execution makes input tables equi-joinable. This system is relevant to ad hoc data analysis in spreadsheets, where users need to join tables whose join-columns are from the same semantic domain but use different textual representations. The authors developed an optimal sampling strategy that allows Auto-Join to scale to large datasets efficiently, while ensuring joins succeed with high probability. Josie [90] (JOining Search using Intersection Estimation), is an algorithm designed for finding joinable tables in massive data lakes. The problem is formulated as an overlap set similarity search problem, considering columns as sets and matching values as the intersection between sets. The authors observe that modern data lakes typically have massive set sizes and dictionaries that include hundreds of millions of distinct values, making traditional set similarity search techniques inefficient. JOSIE minimizes the cost of set reads and inverted index probes. The authors show that Josie performs almost as well, and in some cases even better, on real data lakes. In [71] the authors propose a new principle called the pigeonring principle, which organizes boxes in a ring and constrains the number of items in multiple boxes to yield stronger conditions. By utilizing this new principle, stronger filtering conditions can be established for problems that identify data objects whose similarities or distances to the query are constrained by a threshold. The authors show that the pigeonhole principle is a special case of the new principle, and all pigeonhole principle-based solutions can be accelerated by the new principle. They introduce a universal filtering framework to encompass the solutions to these problems based on the new principle and discuss how to quickly find candidates specified by the new principle.

Embeddings. In the context of machine learning, embeddings refer to the mapping of an entity (such as a word or object) to a high-dimensional vector of real values. The resulting geometric relationship between the vectors of two entities captures their co-occurrence or semantic relationship. Algorithms for learning embeddings are based on the idea of "neighborhoods" – that similar entities tend to belong to the same neighborhood in a contextual sense. As a result, the algorithm aims to position the vectors representing similar entities close to each other in the resulting vector space. Embeddings have been used for task such as entity resolution [32] to achieve high accuracy and efficiency with minimal human involvement. DeepER uses uni- and bi-directional recurrent neural networks (RNNs) with long short-term memory (LSTM) hidden units to convert each tuple to a vector representation that captures similarities between tuples. A locality sensitive hashing (LSH) based blocking approach is used to produce smaller blocks by considering all attributes of a tuple. Approaches have been made to exploit latent information to represent tuples in relational databases and take advantage of the semantic information [7, 8, 9]. On of the first prototype systems in this direction is [**termite**], where embedding are learnt from the data, reducing human effort.

They also implement a Termite-Join operator that identifies related concepts, even in unstructured data. [40] takes into account the semantics of the word embedding and the relational schema. RetroLive, an interactive system, is developed based on Retro, which allows exploring how retrofitted embeddings improve performance for various Machine Learning and integration tasks. The system also includes interactive visualizations to explore the characteristics of the adapted vectors and their connection to the relational database. REMA [48] uses relational embeddings to capture semantic similarity of attributes. Rema embeds database rows, columns, and schema information into multidimensional vectors that reveal semantic similarity. In [11, 12] The authors proposes an algorithm for obtaining local embeddings that are effective for data integration tasks on relational databases. The algorithm uses a compact graph-based representation to specify a rich set of relationships inherent in the relational world. It then derives sentences from such a graph that effectively describes the similarity across elements (tokens, attributes, rows) in the two datasets, and the embeddings are learned based on such sentences. The paper also proposes effective optimization to improve the quality of the learned embeddings and the performance of integration tasks. Other approaches use embeddings for different tasks, such as predicting column types and relationships [79], matching textual content [1] with structured data in an unsupervised setting and linking entities across relations and graphs [56, 35]. Particular interest also to schema matching [84, 82].

2.2 Hashing

Hashing techniques are known to be effective both in space and time to represent data in compressed but efficient way [19]. Among the various approaches to hashing, a very relevant area related to our work is the Locality Sensitive Hashing family. The Locality Sensitive Hashing (LSH) index is a method for solving the approximate nearest neighbor search problem in high-dimensional spaces. It can also be used for the approximate R-near neighbor search problem. The LSH index requires a family of hash functions that have a high collision probability for inputs that are close and a low collision probability for inputs that are far apart. The distance measure used must be symmetric. The formal definition of LSH functions can be found in [43]. An LSH function is a hash function whose collision probability is high for inputs that are close and low for inputs that are far apart. While LSH was originally developed for Jaccard distance, it has been extended to other distance metrics such as Euclidean [23], Hamming [43], and Cosine distance [16]. However, some LSH variations require the data to be vectors with fixed dimensions, which is not practical for domain search at an internet scale. MinHash LSH is better suited for internet-scale domain search as it does not require a fixed set of domain values. It has been extensively compared with SimHash [78], while Asymmetric Minwise Hashing is a state-of-the-art technique for containment search over documents that uses MinHash LSH to index transformed domains. Even if useful for con-

tainment search, the asymmetric transformation can reduce recall when the domain size distribution is highly skewed. Lately, LSH has been the center of both theoretical and practical improvements [89, 17, 3].

2.3 Augmentation Approaches

Arda [18] discusses automatic data augmentation, which involves finding new features relevant to a user's predictive task with minimal human involvement. ARDA is an end-to-end system that takes a dataset and a data repository as input and outputs an augmented dataset that improves the performance of a predictive model. The system has two components: (1) a framework to search and join data based on various attributes of the input and (2) a feature selection algorithm that prunes out noisy or irrelevant features from the resulting join. The authors extensively evaluate different system components and benchmark their feature selection algorithm on real-world datasets. ARDA is part of the family of techniques for automatic machine learning, which aims to automate the process of training predictive models and make machine learning more accessible. A concrete limitation of ARDA is that it assume as input the set of candidate keys, ignoring the discovery of joinable columns and it does materialize each possible join after sampling, making it very expensive under a computational perspective. Pexeso [30] is a framework for discovering joinable tables in data lakes that can handle misspellings, different formats, and capture semantic joins. The framework targets the case when textual values are embedded as high-dimensional vectors and columns are joined upon similarity predicates on these vectors. To efficiently find joinable tables with similarity, the authors propose a block-and-verify method that utilizes pivot-based filtering and a partitioning technique for large data lakes. The solution identifies more tables than equi-joins and outperforms other similarity-based options, with join results useful in data enrichment for machine learning tasks. Pexeso rely on pre-trained embedding mappings, making it difficult to retrieve joinable columns on dataset that have low coverage with respect to embedding vocabulary. Auctus [13] discusses the challenges involved in discovering relevant structured data from the large volumes of structured data available, from Web tables to open-data portals and enterprise data. The authors present the Auctus dataset search engine, which addresses some of these challenges. They describe the system architecture and how users can explore datasets through a rich set of queries. The authors also present case studies demonstrating how Auctus supports data augmentation to improve machine learning models and enrich analytics. Auctus extracts metadata from the input tables to profile data, and act as a human-in-the-loop system, in which every table identified as joinable must be validated and, although the join columns are suggested, the same user must click on them to move on in the process. As expected, it works much more an engine than as an automatic system. Valentine [49] discusses the challenges involved in capturing relationships among heterogeneous datasets in large data lakes, traditionally termed schema matching. The

effectiveness of schema matching methods heavily relies on discovering and integrating datasets. However, evaluating schema matching methods is still a daunting task due to the lack of openly-available datasets with ground truth, reference method implementations, and comprehensible GUIs. The authors propose Valentine, the first system to offer an open-source experiment suite to organize, execute and orchestrate large-scale matching experiments. Valentine provides a user-centric GUI and a scalable holistic matching system that can receive tabular datasets from heterogeneous sources and provide similarity scores among their columns to facilitate modern procedures in data lakes, such as dataset discovery. The demonstration showcases Valentine's functionalities and enhancements. Sketch [75] discusses how dataset search is a critical capability in both research and industry, enabling new applications from data enrichment to machine learning model improvement. The authors focus on a specific type of data-driven query that supports relational data augmentation through numerical data relationships: given an input query table, finding the top-k tables that are both joinable with it and contain columns that are correlated with a column in the query. They propose a novel hashing scheme that allows the construction of a sketch-based index to support efficient correlated table search. The authors show that their proposed approach is effective and efficient, achieving better trade-offs that significantly improve both ranking accuracy and recall compared to the state-of-the-art solutions. Most of the approaches described in this Section actually rely on those presented in Section 2.1 as a low level implementation of joinability discovery.

3

Background

In this chapter, we introduce the necessary background concepts to fully describe the problem we are facing in this work. The chapter is divided into five parts, each focusing on a particular aspect of data management. The first part examines the core principles of Database Management Systems (DBMS) and the fundamentals of relational algebra. Relational tables form the backbone of a majority of data storage systems, as they enable the efficient organization and retrieval of structured data. In the second part the focus shifts to the innovative approach of viewing tables as queries. This paradigm allows for the seamless integration of data from disparate sources, as well as the efficient retrieval of information from large-scale databases and data lakes. The section delves into the underlying principles of this paradigm, exploring its benefits. The third part presents the fundamental concepts of data compression, entropy, and redundancy in the context of data management. Understanding these concepts is essential for designing efficient algorithms and data structures that minimize storage requirements and optimize data transmission. This section also highlights the role of information theory in improving the performance of database systems. The fourth part, covers the techniques and methods used to map data items to hash fingerprints, allowing for efficient retrieval and storage of information. Lastly, the fifth part explores the use of spatial data structures for indexing and querying geometric data.

3.1 Relational Tables

A database management system (DBMS) is a software system that allows users to store, retrieve, and manage data efficiently. In a DBMS, data is organized in tables, which consist of rows (also called records or tuples) and columns (also called fields or attributes). Each row represents a single instance of the entity being described by the table, and each column represents a different aspect or property of that entity.

A common problem in DBMS involves finding a subset of rows that satisfy certain criteria specified by the user. For example, a user might want to retrieve all customers who have purchased a particular product in the last year, or all employees who have worked for the company for more than 5 years. The criteria for selecting the rows can be expressed as a Boolean expression or a combination of Boolean expressions, where each expression is based on one or more columns of the table. The goal is to find an efficient way to evaluate

these expressions and retrieve the matching rows, while minimizing the time and resources required to do so.

Relational Algebra. Relational algebra is a language consisting in a set of mathematical operations used to query data stored in a relational database. These operations are designed to provide a declarative approach to data retrieval, allowing users to specify the desired result set without worrying about the underlying data access mechanisms.

In relational algebra, data is represented as tables, which consist of a set of named columns and a set of rows, with each row representing a single record. Each column is associated with a data type, which defines the kind of data that can be stored in that column. The basic constructs of relational algebra include:

- Selection: $\sigma_\theta(R)$: a unary operation that selects a subset of rows from a table that satisfy a specified condition θ.
- Projection: $\pi_{A_1,A_2,...,A_n}(R)$: a unary operation that selects a subset of columns $A_1, A_2, ..., A_n$ from a table.
- Union: $R \cup S$: a binary operation that combines two tables with the same schema into a single table, eliminating duplicate rows.
- Difference: $R - S$: a binary operation that computes the set of rows that are in one table but not in another.
- Cartesian product: $R \times S$: a binary operation that combines two tables into a single table by taking the cross product of their rows.
- $R \bowtie_\theta S$: a binary operation that combines two tables based on a common column, producing a new table with columns from both tables where the common column values match.

Table Retrieval for Relational Tables. In table retrieval, the goal is to retrieve tables that are similar to a given input table. Regarding the join operation in such a scenario, one approach is to hash each column to a fingerprint, and then use a similarity search algorithm to find columns with similar fingerprints. These similar columns can be treated as if they were in the same table, and the join operation can be performed on them. Specifically, a hash function can be used to map each column's values to a vector of integers, and then a distance metric such as Euclidean distance can be used to measure the similarity between two columns' fingerprints. By setting a threshold on the distance, columns with similar fingerprints can be identified and joined together. One approach to solving this problem is to hash all the columns of the input table to a fingerprint and then perform a similarity search in the resulting fingerprint space. However, this approach does not take into account the relational structure of the data, and can lead to poor performance for certain types of queries. To address this issue, we propose to use the basic constructs of relational algebra to model the table retrieval problem. Specifically, we can represent each table as a set of tuples, and perform selection, projection, and join operations to compare tables. For example, we can select a subset of rows

from a table that match a given condition, project a subset of columns from a table, or join two tables based on a common column. To simulate the join operation in table retrieval, we can hash the columns of the input table to very similar fingerprints, such that columns with the same value in the common column have similar fingerprints. We can then perform a similarity search on the resulting fingerprint space, and retrieve tables that are similar to the input table based on the common column. This approach takes into account the relational structure of the data and can lead to better performance for certain types of queries.

Assuming that we have two tables, Table A and Table B, with a known schema, we want to perform a join operation on these tables based on a common column, such as "ID".

In SQL, we would write something like:

```
SELECT *
FROM TableA
INNER JOIN TableB
ON TableA.ID = TableB.ID
```

This would return a result set containing all the columns from both tables where the ID column matches.

In comparison, a table join via hashing would involve creating hash codes for each column in both tables and comparing them to find matches. For example, we could use MinHash or Locality Sensitive Hashing (LSH) to hash the columns and identify matches between the two tables.

Here's an example of how this might look in Python using the LSH-Ensemble algorithm:

```
    from datasketch import MinHash, MinHashLSHEnsemble

# Assume Table A and Table B are Pandas DataFrames with
    columns "ID" and "Value"

# Convert each column to a set of shingles (in this case,
    just the values)
A_shingles = [set(TableA["Value"])]
B_shingles = [set(TableB["Value"])]

# Create MinHash objects for each column in both tables
A_minhashes = [MinHash(num_perm=128) for i in range(len(
    TableA.columns))]
B_minhashes = [MinHash(num_perm=128) for i in range(len(
    TableB.columns))]

# Update each MinHash object with the shingles from each
    column in each table
for i in range(len(TableA.columns)):
    A_minhashes[i].update(A_shingles[i])
```

```
for i in range(len(TableB.columns)):
    B_minhashes[i].update(B_shingles[i])

# Create an LSH Ensemble object and add the MinHash objects
    from Table A
lsh = MinHashLSHEnsemble(num_perm=128, num_part=16)
for i in range(len(TableA.columns)):
    lsh.add(str(i), A_minhashes[i])

# Query the LSH Ensemble with the MinHash objects from
   Table B
matches = []
for i in range(len(TableB.columns)):
    result = lsh.query(B_minhashes[i])
    matches.append(result)

# Flatten the list of matches and remove duplicates
matches = list(set([item for sublist in matches for item in
    sublist]))

# Join the matching rows from Table A and Table B
result = TableA.loc[TableA["ID"].isin(matches)].merge(
   TableB.loc[TableB["ID"].isin(matches)], on="ID")
```

In this example, we are using MinHash to create hash codes for each column in both tables, and LSH Ensemble to find matches between them. We then join the matching rows from Table A and Table B to create a new table with the desired output. Overall, we can see that table join via hashing can be a powerful alternative to SQL joins when dealing with large data lakes that have no known schema or index.

3.2 Table-as-a-Query Paradigm and Table Retrieval

A data lake $\mathcal{D}$ is a collection of tables $\{T_1, \ldots, T_n\}$. T_i is a table composed of column (i.e., Attributes) and rows (i.e., Tuples). Each table does not hold any particular shape and the header is unknown. Each table T_i is composed of a set of Attributes $\{A_1(T_i), \ldots, A_k(T_i)\}$. An equivalent notation to access the elements of column A_j in table T_i is $T_i.A_j$. Similarly, a tuple of table T_i is identified by $row_l(T_i)$, stating that we are accessing the l-th tuple of the table. Each table is stored in the data lake without any prior information or metadata, the only accessible information lie inside the table themselves. The lack of metadata implies that in order to extract knowledge from the tables they must be accessed or queried one-by-one.

Among the tables in the data lake, we identify a special table Q called *query table*. Such a table must have at least two special attributes, namely:

- an Attribute $\mathcal{J}$, representing the *Join* column;

- an Attribute $\mathcal{Y}$, representing the *Target* column.

Reconnecting to what discussed in Chapter 1, the Join column $\mathcal{Q}.\mathcal{J}$ represents the column on which our similarity search is built. In particular, we want to retrieve all the tables in $\mathcal{D}$ having at least a column which is *similar* to $\mathcal{Q}.\mathcal{J}$. The Target column $\mathcal{Q}.\mathcal{Y}$ refers to the column of the query table against which we would like to perform the augmentation. $\mathcal{Q}.\mathcal{Y}$ represents the outcome of a machine learning task, that can be either classification or regression.

Table Retrieval. Table retrieval is the task of finding a table or a set of tables from a large dataset based on a user's query. This problem can be declined under two main scenarios: when the schema is known (DBMS) and when the schema is unknown (data lakes). This distinction is crucial and poses the problems at the core of this work, because once tackling data lakes and schemaless data, retrieving tables efficiently is not a computational issue only, but also a methodological problem as well. Such a task is particularly challenging when the query is underspecified, the data is unstructured, and there is no schema or indexing information available. In this section, we formalize the table retrieval problem in the context of data lakes and introduce a novel approach based on hashing, join search, similarity, and augmentation.

The problem of table retrieval in a DBMS involves finding tables that are similar to a given query table based on their column-wise content. In Section 1.1 we discussed some intuitions that can be shaped as a table retrieval setting. More formally, suppose we have a set of n tables $T_1, T_2, ..., T_n$, each having m columns $C_1, C_2, ..., C_m$. Given a query table Q with columns $C'_1, C'_2, ..., C'_m$, the goal is to find the tables from the set that are most similar to Q based on their column-wise content. This can be done exploiting the schema information and writing plain SQL queries to solve the problem.

For what concerns the data lake scenario, let us consider a D that consists of n tables $T_1, T_2, \ldots, T_n$, where each table T_i has m_i columns and k_i rows. We assume that the data lake does not have any schema or indexing information, and the tables are unordered. The goal of table retrieval is to find a subset of tables $S \subseteq T_1, T_2, \ldots, T_n$ such that S satisfies a given query Q. The query Q is defined as a table with m columns and k rows, where each cell can be a value or a wildcard. A wildcard represents any value and allows the query to be underspecified.

One solution to this problem is to hash all the columns of the tables to a fingerprint and perform a similarity search in such a space. Each column can be hashed to a binary vector, and the table can be represented as a set of these binary vectors. The similarity between two tables can then be calculated based on the similarity of their corresponding sets of binary vectors. This approach allows efficient retrieval of tables that are similar to the query table, as it reduces the search space to a much smaller dimension. However, choosing an appropriate hash function and setting the parameters of the hash function can be challenging and requires careful consideration to achieve accurate and efficient retrieval. In the following of this Chapter, we will dive better in the formal details of this setting.

3.3 Information Theory

Information theory is the study of quantifying the amount of information in a message or a signal. It provides a framework to measure, store, and communicate information. In this section, we will discuss some fundamental concepts of information theory, including entropy, conditional entropy, and mutual information.

Probability and Density Functions in Information Theory. In information theory, probability, conditional probability, and density functions play a crucial role in understanding the uncertainty associated with a random variable. The probability of an event is defined as the measure of the likelihood that the event will occur, and it is always a value between 0 and 1. If an event is impossible, its probability is 0, and if it is certain to occur, its probability is 1. The probability of an event A is denoted by P(A). Conditional probability refers to the probability of an event occurring given that another event has occurred. It is denoted by P(A|B), where A and B are events. The conditional probability of A given B is calculated as the probability of both A and B occurring divided by the probability of B occurring:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{3.1}$$

In information theory, we also deal with uncertain events, and we use probability theory to model and quantify the uncertainty. Probability theory provides a framework for computing the likelihood of events, given some prior knowledge or assumptions. The basic concept in probability theory is a random variable, which is a variable that can take on different values with some probability distribution. The probability distribution specifies the likelihood of each possible value that the random variable can take. The probability of an event E is defined as the sum of the probabilities of all the outcomes that correspond to E. If $P(E)$ is the probability of event E, then $0 \leq P(E) \leq 1$. If E_1 and E_2 are two events, then their union $E_1 \cup E_2$ is the event that either E_1 or E_2 occurs. The probability of the union is given by the sum of the probabilities of E_1 and E_2, minus the probability of their intersection $E_1 \cap E_2$: $P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$. The intersection of two events E_1 and E_2 is the event that both E_1 and E_2 occur. Conditional probability is a measure of the probability of an event occurring given that another event has occurred. It is defined as the probability of the intersection of two events E_1 and E_2, divided by the probability of E_2: $P(E_1|E_2) = P(E_1 \cap E_2)/P(E_2)$. Conditional probability is used to model the dependence between events, and it is essential in Bayesian inference and machine learning.

Entropy. Entropy is a measure of the amount of uncertainty or randomness in a system. It was first introduced by Claude Shannon in his seminal paper on information theory in 1948 [77]. The entropy of a random variable X, denoted by $H(X)$, is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x), \tag{3.2}$$

where $\mathcal{X}$ is the set of possible outcomes of X, and $p(x)$ is the probability mass function of X. The entropy measures the average amount of information conveyed by each outcome of X. It is maximum when all outcomes are equally likely, and minimum when one outcome is certain to occur.

Conditional Entropy. Conditional entropy is a measure of the uncertainty remaining in a random variable Y after another random variable X is observed. The conditional entropy of Y given X, denoted by $H(Y|X)$, is defined as:

$$H(Y|X) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(y|x), \tag{3.3}$$

where $\mathcal{Y}$ is the set of possible outcomes of Y, and $p(y|x)$ is the conditional probability mass function of Y given $X = x$. The conditional entropy measures the average amount of information needed to describe Y after X is observed. It is minimum when Y is completely determined by X, and maximum when Y is independent of X.

Mutual Information. Mutual information is a measure of the amount of information that two random variables X and Y share. The mutual information between X and Y, denoted by $I(X; Y)$, is defined as:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log \frac{p(x, y)}{p(x)p(y)}, \tag{3.4}$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the sets of possible outcomes of X and Y, respectively, and $p(x, y)$, $p(x)$, and $p(y)$ are the joint and marginal probability mass functions of X and Y. The mutual information measures how much knowing X reduces the uncertainty about Y and vice versa. It is zero if X and Y are independent, and positive if they are dependent. Mutual information is often used in machine learning to measure the dependence between features and labels or between different features. It can be used to select informative features, to cluster data, or to evaluate the performance of a classification or regression model.

Conditional Entropy for Expressing Functional Dependencies in DBMS. In database management systems (DBMS), functional dependencies are used to model relationships between attributes in a relation. These dependencies are used to enforce constraints on the data, ensuring that it is consistent and free from redundancies. Conditional entropy is a measure of the amount of uncertainty in a random variable given knowledge about another random variable. In the context of functional dependencies, conditional entropy can be used to express the degree to which one attribute is dependent on another.

Suppose we have a relation $R(A, B)$, where A and B are attributes. We can define the functional dependency $A \rightarrow B$ to mean that, for any two tuples t_1 and t_2 in R, if $t_1.A = t_2.A$, then $t_1.B = t_2.B$. To express this functional dependency using conditional entropy, we can compute the entropy of B given A, denoted as $H(B|A)$. If $H(B|A) = 0$, then B is completely determined by A and the

functional dependency $A \rightarrow B$ holds. If $H(B|A) > 0$, then B is not completely determined by A and the functional dependency does not hold. Conditional entropy can also be used to express partial dependencies. Suppose we have a relation $R(A, B, C)$, where A and B together determine C. We can express this partial dependency as $A, B \rightarrow C$. To check if this partial dependency holds, we can compute the entropy of C given A, B, denoted as $H(C|A, B)$. If $H(C|A, B) = 0$, then C is completely determined by A and B and the partial dependency holds. In summary, conditional entropy provides a useful tool for expressing functional dependencies and partial dependencies in DBMS. By computing the entropy of one attribute given another, we can determine the degree of dependence between the attributes and enforce constraints on the data to ensure consistency and avoid redundancies.

3.4 Hashing

Many relevant approaches in set similarity search rely on hashing function. In particular, in this section we will see and discuss the broad family of LSH algorithms and the data sketches that rely on it.

LSH. Locality-sensitive hashing (LSH) is a widely used approximate nearest neighbor (ANN) search technique. The idea behind LSH is to hash the high-dimensional data points into lower-dimensional hash codes in a way that the hash codes of similar data points are likely to be the same. This allows for fast similarity search by only comparing the hash codes of data points instead of the original high-dimensional vectors. The LSH algorithm can be divided into two main steps: (1) hash function construction and (2) hash table creation. In the first step, a set of hash functions is generated such that each hash function maps data points to a binary hash code. The hash functions are designed in such a way that the probability of two data points having the same hash code is higher if the data points are similar according to some distance metric. In the second step, hash tables are created by grouping data points with the same hash code into the same bucket.

The quality of LSH heavily depends on the choice of hash functions and on the number of hash tables used. A good set of hash functions should maximize the probability of collision for similar data points while minimizing the probability of collision for dissimilar data points. The number of hash tables used affects the quality of the search results, with a larger number of hash tables generally resulting in more accurate results at the cost of increased computation and memory usage. LSH is widely used in large-scale machine learning and data mining applications, such as image and video search, document retrieval, and recommendation systems. It is particularly useful when dealing with high-dimensional data, where exact nearest neighbor search becomes computationally infeasible. However, it is important to note that LSH is an approximate search technique and may not always return the exact nearest neighbor. The quality of the search results depends heavily on the choice of hash functions and the number of hash tables used.

LSH Ensemble. LSH Ensemble is a popular technique for approximate nearest neighbor search in high-dimensional data. It involves combining multiple hash tables, each using a different random projection, to produce more accurate and robust query results. LSH Ensemble has been shown to outperform single LSH tables and other state-of-the-art indexing methods in many real-world scenarios. One of the key advantages of LSH Ensemble is its ability to handle high-dimensional data efficiently. Traditional indexing methods, such as tree-based approaches, suffer from the curse of dimensionality and become impractical when the number of dimensions is large. LSH Ensemble, on the other hand, can be easily scaled to high-dimensional data without significant loss in query accuracy. Moreover, LSH Ensemble can be implemented efficiently on distributed systems, making it a suitable candidate for large-scale applications.

Another advantage of LSH Ensemble is its flexibility and tunability. By adjusting the number of hash tables and their parameters, one can control the trade-off between query accuracy and query time. Furthermore, LSH Ensemble can be combined with other indexing methods to produce hybrid solutions that offer even better performance. Despite its advantages, LSH Ensemble is not a silver bullet and has its limitations. One of the major challenges is the selection of the optimal hash functions and their parameters. This process requires careful tuning and experimentation, and may not be feasible in some scenarios. Moreover, LSH Ensemble may not perform well on datasets with complex structures or skewed distributions.

Overall, LSH Ensemble is a powerful and widely used technique for approximate nearest neighbor search. Its combination of efficiency, scalability, and flexibility makes it a suitable choice for many real-world applications.

Hashing for Table Retrieval. In table retrieval, the goal is to efficiently search a large database of tables to find those that match a given query. One approach to this problem is to hash the columns of each table to a set of fingerprints and perform a similarity search in the resulting fingerprint space. This approach has several benefits over traditional indexing techniques.

First, hashing allows for fast and efficient retrieval of tables. By converting each column into a fixed-length fingerprint, we can perform similarity searches using algorithms like locality-sensitive hashing (LSH) that are designed to efficiently search large high-dimensional spaces. This enables us to quickly retrieve tables that are similar to the query without performing a full table scan.

Second, hashing is a memory-efficient approach to indexing. Traditional indexing techniques like B-trees or hash tables can be memory-intensive, especially for large databases with many columns. Hashing, on the other hand, requires only a small amount of memory to store the fingerprints of each column, making it an attractive option for systems with limited memory resources.

Third, hashing can be used to enable approximate search. In many cases, an exact match for the query is not required, and it may be sufficient to retrieve

tables that are similar to the query. Hashing allows us to perform approximate search efficiently by defining a similarity threshold and retrieving all tables whose fingerprints fall within that threshold.

Hashing for Column Similarity Search. In the previous section, we discussed how the problem of table retrieval can be approached by hashing all the columns of a table to a fingerprint and then performing a similarity search in the resulting space. However, the question remains of how to choose an appropriate hash function for this purpose. One promising approach is to use a hash function that is based on the k-nearest neighbors (k-NN) algorithm. Specifically, we can treat each column of a table as a vector and use k-NN to project these vectors onto a low-dimensional space that preserves their relative distances. This projection can then be used as the fingerprint for the column.

One advantage of using a k-NN based hash function is that it can capture both local and global patterns in the data. Local patterns refer to correlations between neighboring values in a column, while global patterns refer to correlations between columns themselves. By projecting columns that exhibit similar local patterns to similar fingerprints, we can find related columns that might otherwise be missed by traditional similarity metrics. Similarly, by projecting columns that exhibit dissimilar global patterns to dissimilar fingerprints, we can find columns that are not correlated.

To simulate the join operation, we can keep columns hashed to very similar fingerprints together, effectively grouping together columns that share similar values or patterns. This way, when a user queries the table for a particular value or pattern, the algorithm can quickly locate the relevant column group and return the desired results. Overall, using a k-NN based hash function for column similarity search offers a promising approach to the problem of table retrieval, allowing for efficient and effective search through large tables of data.

3.5 Spatial Indexes

Spatial indexing is a technique used in computer science and geographic information systems to efficiently store and query large sets of objects based on their spatial properties. Spatial indexes use data structures to organize spatial data in a way that allows for fast and efficient queries.

A suitable data structure for implementing the KNN-based hash function is a KD-Tree. KD-Trees are a type of binary search tree that is optimized for performing nearest neighbor searches. The idea behind a KD-Tree is to recursively partition the data into smaller regions of the space by splitting along a dimension at each level of the tree. The splitting plane is chosen to be the median of the data in that dimension, and the data points are partitioned into two halves on either side of the plane. This process is repeated until the regions are small enough to perform a linear search.

The KDTree works by recursively partitioning the space into two parts along

a single dimension. Each node in the tree represents a region of space, and the partitioning is done such that the two children of each node correspond to the two halves of the region along the selected dimension. This process is repeated until each leaf node represents a small region of space containing a small number of points. To query the KDTree, one starts at the root node and recursively descends the tree, following the path that is closest to the query point. At each level of the tree, the algorithm selects the child node that is closer to the query point and continues the descent. Once a leaf node is reached, the algorithm returns the points contained in that leaf node and searches the neighboring nodes to ensure that all nearby points are returned. One of the benefits of the KDTree is that it can be used to efficiently answer range queries, where one wants to find all the points within a certain distance of a query point. This is achieved by searching the KDTree recursively, but skipping over nodes that are further away than the current radius. Overall, the KDTree is a powerful data structure for spatial indexing and is widely used in a variety of applications, such as computer vision, data mining, and machine learning.

With the KNN-based hash function, radius queries can be used for both joining (points in the radius) and machine learning useful columns (far away from the radius). A radius query is a query that returns all data points within a certain radius of a given query point. In the context of table retrieval, radius queries can be used to find columns that are correlated with a given column or set of columns.

In addition to KDTree, there are other types of spatial indexes that are commonly used in data science and machine learning, such as quadtree, R-tree, and ball tree. However, these indexes have different strengths and weaknesses compared to KDTree, which is why we chose to use KDTree for our solution. The *quadtree* [74, 81] is a tree data structure that recursively subdivides space into four quadrants. It is commonly used for partitioning two-dimensional space for efficient querying of points or rectangles. However, it does not work well for high-dimensional data like the ones we are dealing with in our table augmentation problem. The *R-tree* [47, 4, 5] is a tree data structure that is used for spatial indexing of multidimensional data like points, lines, and rectangles. It uses bounding boxes to organize the data in a hierarchical manner, allowing for efficient range queries. However, R-trees can be expensive to build and update, and can have poor performance on skewed distributions of data. The *ball tree* [28] is a tree data structure that partitions space into a series of hyperspheres, making it suitable for nearest-neighbor searches in high-dimensional spaces. However, it can be expensive to build and maintain, and can be sensitive to the choice of distance metric.

Overall, we chose to use KDTree for our solution because it is efficient, easy to implement, and can handle high-dimensional data well. Its performance is not significantly affected by skewed distributions of data, and it can handle both range and nearest-neighbor queries efficiently.

4

Problem Statement and Approach

In this chapter, we formalize the problem statement and present in detail our proposed approach. We begin by formalizing the problem statement and outlining its significance in the context of machine learning.

However, before delving into the details of our proposed approach, we believe it is important to discuss the various basic approaches that we explored during our research. We experimented with several ideas and techniques, some of which proved to be ineffective and led us to blind spots.

Therefore, we dedicate a significant portion of this chapter to discussing these approaches and the insights we gained from them. We will discuss how we designed each of the solutions that resulted ineffective, highlighting both advantages and limitations and how we overpassed the issues.

Specifically, we discuss our attempts at discovering approximate functional dependencies to evaluate relatedness, including the approximate functional dependencies and graph functional dependencies. We also explore the use of information theory measures from marginals and extending the Josie algorithm. Finally, we consider the potential of word embedding to map tables.

After presenting these failed approaches, we then present our proposed solution in detail. By contextualizing our approach within the broader landscape of failed attempts, we aim to provide a more nuanced and comprehensive understanding of the problem of horizontal table augmentation and the potential solutions to it.

4.1 Problem Statement

Let $\mathcal{Q}$ be a table, which will be referred to as *Query Table*, containing

- an attribute $\mathcal{J}$, representing the *Join* column;
- an attribute $\mathcal{Y}$, representing the *Target* column.

We consider $\mathcal{J}$ a column composed of string only values, while $\mathcal{Y}$ is a numeric column acting as the target column of a machine learning task.

Let $\mathcal{D}$ be a data lake consisting of a set of tables, including $\mathcal{Q}$.

Let $\mathcal{T} \in \mathcal{D}$ be a table, that will be referred to as *Joinable Table* in the data lake that contains:

- a column $\mathcal{J}$' that may contain overlapping sets of values with $\mathcal{J}$, so that columns $\mathcal{J}$ and $\mathcal{J}$' can be used to join tables $\mathcal{T}$ and $\mathcal{Q}$, i.e., $\mathcal{Q} \bowtie_{\mathcal{J}=\mathcal{J}'} \mathcal{T}$

- a number of other columns C

Let, finally, be M be a machine learning task on the tables in $\mathcal{D}$ and let $\mathcal{M}$ be its performance. In this setting, $\mathcal{M}$ is intended to be a number whose domain depends on the task. For a classification task, $\mathcal{M} \in [0, 1]$, for regression tasks it represents the mean error, so its range is $[0, \infty]$. Therefore, $\mathcal{M}$ can be considered a contraction of $\mathcal{M}(\mathcal{Y})$.

Let us define a *Candidate Augmenting Column* as a column $\mathcal{CA}$ such that $\mathcal{CA}$ is a column of a joinable table $\mathcal{T}$, different from the join column $\mathcal{J}$', such that

$\mathcal{M}(\mathcal{Q} \bowtie_{\mathcal{J}=\mathcal{J}'} \Pi_{\mathcal{J}',\mathcal{CA}} \mathcal{T}) > \mathcal{M}(\mathcal{Q})$.

A column $\mathcal{CA}$ can contain any type of data, there are no limitations on its type. Thus, the augmenting column improves the performance of the machine learning task with respect to that achieved on the query table alone. This is the desired behavior for applications such as data augmentation for machine learning models, where the goal is to add new columns (features) to an existing training dataset while maintaining the same number of rows.

We can define the *augmentation improvement* of an augmenting column $\mathcal{CA} \in \mathcal{T}$ as the increase in $\mathcal{M}$, that is:

$\Delta_A(\mathcal{CA}) = \mathcal{M}\ (\mathcal{Q} \bowtie_{\mathcal{J}=\mathcal{J}'} \Pi_{\mathcal{J}',\mathcal{CA}}\ (\mathcal{T}))$ - $\mathcal{M}(\mathcal{Q})$.

Among the candidate augmenting columns, we define a ranking $\mathcal{R}$ induced by the order relation $>_A$ such that, given two augmenting columns $\mathcal{CA}_1 \in \mathcal{T}_1$, $\mathcal{CA}_2 \in \mathcal{T}_2$: $\mathcal{CA}_1 >_A \mathcal{CA}_2$ if $\Delta_A(\mathcal{CA}_1) > \Delta_A\ (\mathcal{CA}_2)$.

We are then typically interested, given an integer k, to retrieve the top-k augmenting columns among the candidate ones, i.e., finding the k augmenting columns with the highest augmentation improvement.

Note that this ranking of columns also implies a ranking of tables, since the augmenting columns may belong to different tables and that while the definition of augmentation improvement may seem to refer to a choice of a column only, this actually requires a high degree of joinability as well. Tables are ordered according to their most relevant column performance, expressed in terms of distance from the target column.

The ranking of tables must satisfy the rule that the top-k tables of the ranking *produce an augmentation*, meaning that the suggested join between the query table $\mathcal{Q}$ and the top-ranked tables in the ranking $\mathcal{R}$ will improve the predictive ability of $\mathcal{Q}$ with respect to $\mathcal{T}$. The idea is to suggest to the user the most interesting table that guarantees an improvement *without* the need of materializing any join to catch such information. Our system returns the list of tables worthy of being joined to the query table.

4.2 Basic Approaches

DB-Like functional dependencies. Our first intuition was to consider functional dependencies to evaluate the grade of relatedness of two columns. As it is known, functional dependencies have a high degree of complexity in computing their presence [62]. Before focussing on the possibility of computing

Element	Symbol	Description
Data lake	$\mathcal{D}$	A repository of tables without any metadata information. Typical scale of such a structure is in hundreds of thousands tables.
Query table	$\mathcal{Q} \in \mathcal{D}$	A table identified as query. It will be the table object of search by the index.
Join column	$\mathcal{J} \in \mathcal{Q}$	A column of the query table acting as join search key. All columns of all tables in the data lake similar to this column will be identified.
Target column	$\mathcal{T} \in \mathcal{Q}$	A column of the query table acting as target column. All columns of all joinable tables in the data lake that contains relevant features to help the columns of the query table in predicting this column will be identified.
Candidate table	$\mathcal{T} \in \mathcal{D}$	A table that can be joined with the query table, i.e., a table that contains at least a column very similar to the Join column
Candidate Column	$\mathcal{CA} \in \mathcal{T}$	A column of a candidate table that can produce an augmentation, i.e., a column that contains relevant features to improve the predictive ability of $\mathcal{Q}$
Candidate join column	$\mathcal{J}' \in \mathcal{C}$	A column of $\mathcal{C}$ that contains common values with $\mathcal{J} \in \mathcal{Q}$
Model Performance	$\mathcal{M}$	The performance of a machine learning in predicting $\mathcal{Y}$.
Ranking	$R(C_1, \ldots, C_n)$	An ordered list of tables that, joined with the query table, will produce an improvement in the query table ability of predicting the target column.
Hash Function	$H(C)$	An hash function applied to all column at the time of index creation.

TABLE 4.1 Ingredients involved in the problem statement

FDs, either in an exact or approximate way as we will discuss later on, we put the attention on its mathematical validity. In this phase, the research question was: *Is there a relation between the existence of a functional dependency among columns and their relatedness?*

A functional dependency is a relationship between two attributes in a relation that describes how the values in one attribute determine the values in the other attribute. Formally, we can define a functional dependency as follows:

Let R be a relation with attributes C_1 and C_2, and let t_1 and t_2 be tuples in R. We say that C_1 functionally determines C_2 in R, denoted as $C_1 \rightarrow C_2$, if for any two tuples t_1 and t_2 in R:

If $t_1[C_1] = t_2[C_1]$, then $t_1[C_2] = t_2[C_2]$.

In other words, the value of C_2 is uniquely determined by the value of C_1 in any tuple in R. We can also say that C_1 is the determinant and C_2 is the dependent attribute.

For example, let consider a relation R with attributes employee_id, employee_name, and employee_department. We can say that employee_id $\rightarrow$ employee_name because the value of employee_name is uniquely determined by the value of employee_id for any employee in the database. Similarly, we can say that employee_id $\rightarrow$ employee_department for the same reason.

To access the tuples $t_1[C_1]$ and $t_2[C_1]$, we simply use the subscript notation to access the values of attribute C_1 in tuples t_1 and t_2, respectively. For example, $t_1[C_1]$ would access the value of attribute C_1 in tuple t_1.

It is worth noticing that, in a database scenario, it is known what is the primary key of the relation; following the nature of the functional dependencies, the key is the determinant of each other column. Applying this scenario to our use-case suggested us that the belonging of two columns to the same relation it is not a mandatory fact. The meaning of that is the following: we can evaluate the same information, i.e., the functional dependency, between two columns belonging to different tables. The complexity of the operation is not affected. What is truly affected is the implication of a column of table T_1 completely determining another column in table T_2. From a learning perspective, a feature that fully determines another features is irrelevant, or redundant. These intuitions suggested us two relevant facts: a) columns that act as key with respect to the target column are not relevant to provide augmentation and b) discrete values for functional dependencies does not help us in quantifying how much a column determines another.

Functional dependencies, as a matter of principle, are effective in giving insights on the possibility of finding relevant tables, on the augmentation side only. They cannot be easily used to search for joinable table, even if they were computable in reasonable times.

Graph Functional Dependencies. While considering FDs, we moved our attention to graph functional dependencies for a while. This is because we started to consider a data lake as a graph rather than a repository of tables. Other approaches do the same thing [55] while considering different problems. This can be done at different levels of granularity: a node of the graph can be

a table, a column or even a value. We focused on the latter, supported by the existence of large scale graph engines such as Neo4J [66] that support efficient operation on data lake size use cases. Many approaches exist that search for FDs in graphs [36, 37]. Since the complexity of such discovery is the same as traditional FDs, the problem of GFDs has been posed as detection of patters in a graph, the identification of subgraphs to make the problem easier and detect violation of FDs rather than considering their discovery. Summarizing this brief excursion, the idea of considering a data lake as a graph is surely interesting but of limited usefulness, since it does not simply of any order the discovery and make possibly more difficult to retrieve joinable tables. Furthermore, in addition to the graph representation, a mapping table-column-value must be kept, suggesting that the space required for such a solution would be greater than the data lake itself.

Discovering Approximate Functional Dependencies to Evaluate Relatedness. Stated that functional dependencies can be somehow interesting to evaluate augmentation, we focused on the possibility of actually computing them. As well as our join search, functional dependencies discover can be done either exactly on approximately. Exact discovery is harder and in the scope of our work less interesting, since we rely on a probabilistic background, so we focused on approximate approaches. A very relevant work is this scenario is PYRO [50]. PYRO is based on a lattice structure that is useful to generate all possible dependencies while keeping reasonable pruning techniques to make the problem computable. Lattices allow for a bottom up traversal in which previously computed results can be reused without the need of extra computation. PYRO has been built on top of previous works in the field of FDs discovery such as [42, 34]. We tested some of the algorithms that tackle this problem, although some consideration emerges. The main disadvantage of considering traditional FDs is that they do not provide any insight on how a column or a set of columns can improve the predictive ability of query table. Alternately, they can be used as metadata after the augmentation has been provided, for example by reducing the number of involved columns to avoid redundancy. This analysis is out of the scope for this work, since the issue of actually computing these measures in a reasonable time is still open.

Reliable Fraction of Information. While considering functional dependencies a promising direction, we stepped back for a while to a wider look at what a functional dependency is. Since we were deep in a database-oriented perspective, we asked ourselves which other communities could have faced a similar problem. We came out the whole community that focused on data and knowledge discovery. We ended up on works that tries to discover functional dependencies from data outside the relation world [60, 59, 58]. This approach discusses the Fraction of Information $F(X;Y)$ as an ideal choice for measuring dependency between categorical input and output variables. $F(X;Y)$ is defined as the ratio between mutual information and Shannon entropy, representing the proportional reduction of uncertainty about Y by knowing X. However, estimating mutual information from empirical samples is challenging and

can lead to overestimation, known as dependency-by-chance. To address this issue, the authors propose a reliable mutual information estimator $\hat{I}_0(X;Y)$ by correcting the standard plug-in estimator with an expected value under a null hypothesis model. This estimator accounts for dependency-by-chance and is computationally efficient. However, maximizing this estimator is NP-hard. The authors develop bounding functions for $\hat{I}_0$, enabling efficient exact, approximate, and heuristic algorithms to be used in branch-and-bound and heuristic search, thus reducing the search space. This approach discovers reliable dependencies among categorical input features and an output variable. Meaning that, the selected groups of input features approximate well the output variables. This approach prove to be useful and effective in a slightly different scenario to ours: It finds the subset of input categorical features that correlates better to an output feature. It does not search for features that can provide an augmentation, rather it finds group that correlate to the output variable. In our scenario, that would not provide augmentation, rather it would add redundancy to query table because the found columns would behave accordingly to the output. Furthermore, since the problem has a different setup and statement, there are many difficulties in adopting such a framework to our scenario. First, the search for joinable tables should be done elsewhere. None of the phases of the approach seems to be suitable to a join integration. Secondly, it is not meant to work on large datasets, composed of many tables even of small sizes. Another disadvantage of this framework is either its speed in computing FDs (they are actually reliable but slow or even infeasible). Summarizing the hardness of an integration to our case, we can say that adopting this framework would be as difficult as materializing a join over joinable tables (identified using different algorithms). Later in this chapter we will discuss and idea of extending an approach based on hashing that could have been considered the join search step for this adoption. The takeaway of such an experience is that, once again, information theory seems to be an interesting area to invest in for figuring out a solution.

4.3 The Impossibility of Estimating Conditional Entropy from Individual Entropies

In information theory, entropy is a measure of uncertainty or randomness in a random variable. It is defined as the average amount of information contained in each event of the random variable. Entropy is a fundamental concept in information theory and has many applications in fields such as statistics, physics, and computer science. Conditional entropy is a measure of the amount of uncertainty in a random variable given the value of another random variable. It is defined as the average entropy of the conditional distribution of the first variable given the second variable. Intuitively, conditional entropy measures the amount of uncertainty in a random variable that is left after the value of another random variable is known. One might think that given the individual entropies of two random variables, one could estimate their conditional entropy.

However, it turns out that this is not possible in general. In this section, we will prove this result mathematically.

$$H(X) = -\sum_{x \in \mathcal{X}} P(X = x) \log_2 P(X = x) \; H(Y) \quad = -\sum_{y \in \mathcal{Y}} P(Y = y) \log_2 P(Y = y)$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the support sets of X and Y, respectively.

The conditional entropy of X given Y is defined as:

$$H(X|Y) = \sum_{y \in \mathcal{Y}} P(Y = y) H(X|Y = y) \quad = -\sum_{y \in \mathcal{Y}} P(Y = y) \sum_{x \in \mathcal{X}} P(X = x|Y = y) \log_2 P(X = x|Y = y)$$

We want to show that $H(X|Y)$ cannot be determined from $H(X)$ and $H(Y)$ alone. Suppose that we have access to $H(X)$ and $H(Y)$, and let us assume that $H(X|Y)$ can be expressed as a function of $H(X)$ and $H(Y)$, i.e.,

$$H(X|Y) = f(H(X), H(Y))$$

for some function f. We will show that this assumption leads to a contradiction.

Consider the following example. Let X and Y be two independent Bernoulli random variables with $P(X = 1) = p$ and $P(Y = 1) = q$. Then, we can compute the entropies as follows:

$$H(X) = -p \log_2 p - (1 - p) \log_2 (1 - p) \; H(Y) \quad = -q \log_2 q - (1 - q) \log_2 (1 - q)$$

The joint distribution of X and Y is given by:

$$P(X, Y) = P(X)P(Y) = pq(1 - p)(1 - q)$$

The conditional entropy of X given Y is:

$$\begin{aligned} H(X|Y) &= -\sum_{y \in 0,1} P(Y = y) \sum_{x \in 0,1} P(X = x|Y = y) \log_2 P(X = x|Y = y) \\ &= -\sum_{y \in 0,1} q^y (1 - q)^{1-y} \sum_{x \in 0,1} P(X = x|Y = y) \log_2 P(X = x|Y = y) \end{aligned}$$

As we can see, the conditional entropy $H(X|Y)$ reduces to the entropy $H(X)$, which does not depend on Y. This is because X and Y are independent, and hence, knowing Y does not provide any additional information about X.

This example illustrates that, in general, knowledge of the individual entropies $H(X)$ and $H(Y)$ is not sufficient to determine the conditional entropy $H(X|Y)$. The reason is that the conditional entropy takes into account the dependencies between the variables, which cannot be inferred from the individual entropies alone. Therefore, to accurately estimate the conditional entropy, we need to consider the joint distribution of the variables or use other methods, such as machine learning algorithms, to learn the conditional probabilities from data. In the following, we will present two demonstrations. In the first one, we show that $H(X|Y)$ cannot be determined from $H(X)$ and $H(Y)$ alone. In the second one, we show that we cannot derive their conditional entropy $H(X|Y)$ or mutual information $I(X; Y)$.

Proof: Assume we have two columns, X and Y, with known entropies $H(X)$ and $H(Y)$. We aim to show that we cannot derive their conditional entropy $H(X|Y)$ or mutual information $I(X;Y)$.

First, let us consider the definition of conditional entropy:

$$H(X|Y) = \sum_{y \in Y} P(Y = y) H(X|Y = y)$$

By Bayes' rule, we have:

$$P(Y = y) = \frac{P(X = x, Y = y)}{P(X = x)} = \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)}$$

Substituting this into the equation for conditional entropy, we have:

$$H(X|Y) = \sum_{y \in Y} \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)} H(X|Y = y)$$

We can simplify this expression by noting that $P(X = x) = \sum_{y \in Y} P(X = x, Y = y) = \sum_{y \in Y} P(X = x|Y = y)P(Y = y)$, giving:

$$H(X|Y) = \sum_{y \in Y} P(X = x|Y = y)H(X|Y = y) - \sum_{y \in Y} P(Y = y) \log P(Y = y)$$

Similarly, the definition of mutual information is:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} P(X = x, Y = y) \log \frac{P(X = x, Y = y)}{P(X = x)P(Y = y)}$$

Again, we can substitute in the expressions for $P(X = x)$ and $P(X = x, Y = y)$, giving:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} P(X = x|Y = y)P(Y = y) \log \frac{P(X = x|Y = y)}{P(X = x)}$$

Using the same substitution as before, we get:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} P(X = x|Y = y)P(Y = y) \log \frac{P(X = x|Y = y)}{\sum_{y' \in Y} P(X = x|Y = y')P(Y = y')}$$

Thus, both $H(X|Y)$ and $I(X;Y)$ involve the conditional probabilities $P(X = x|Y = y)$, which cannot be derived solely from the entropies $H(X)$ and $H(Y)$.

Therefore, we have shown that if we know the entropy of two columns, we cannot derive their conditional entropy or mutual information. □

Proof: Assume we have two random variables X and Y, and we know their joint probability mass function $P_{X,Y}$, as well as the probability mass functions P_X and P_Y for X and Y, respectively. We want to show that given this information, we cannot determine the conditional entropy $H(X|Y)$ or the mutual information $I(X;Y)$.

First, consider the case of $H(X|Y)$. By definition, we have:

$$H(X|Y) = \sum_{y \in \mathcal{Y}} P_Y(y) H(X|Y = y)$$

where $H(X|Y = y)$ is the conditional entropy of X given $Y = y$. Using the chain rule of entropy, we have:

$$H(X|Y = y) = \sum_{x \in \mathcal{X}} P_{X|Y}(x|y) \log_2 \frac{1}{P_{X|Y}(x|y)} \quad = -\sum_{x \in \mathcal{X}} P_{X,Y}(x,y) \log_2 P_{X|Y}(x|y)$$

where $P_{X|Y}(x|y)$ is the conditional probability mass function of X given $Y = y$. Since $P_{X|Y}(x|y)$ is not given, we cannot compute $H(X|Y)$.

Next, consider the case of $I(X;Y)$. By definition, we have:

$$I(X;Y) = H(X) - H(X|Y)$$

Since we cannot compute $H(X|Y)$, we cannot compute $I(X;Y)$ either.

Therefore, knowing only the joint probability mass function $P_{X,Y}$ and the probability mass functions P_X and P_Y, we cannot determine the conditional entropy $H(X|Y)$ or the mutual information $I(X;Y)$. □

4.4 An Inverted Index based on Information Theory

Motivated by the promising results from information theory in discovering functional dependencies, we tried to build an approach based on an inverted index that leverages information theory to evaluate the relatedness of columns for horizontal table augmentation.

An inverted index is a data structure that stores a mapping from words or terms to their locations in a set of documents. It is the backbone of many search engines, as it allows for efficient querying and retrieval of relevant documents. In our case, we adapt the concept of an inverted index to map column values to their occurrences in tables, enabling efficient search for joinable tables. A visual representation of such an index can be found in Figure 4.1.

Our approach consists of the following steps:

- Key identification: For each table, we identify the column that is more likely to be a key. This can be done in many ways: by treating each column as a set and normalizing by the number of rows or exploiting information theory.
- Index construction: For each column in every table, compute the conditional entropy between the column and the column of the table more likely to be a key. Store these values in an inverted index data structure, where the keys are the column values and the values are the occurrences of these values in the tables. For the association of values and tables where the value does not belong to the table, a special value is inserted. This particular will be useful to the search of joinable tables. We decided to store the conditional entropy with respect the key because our intuition was to exploit the similarity between the key of the table and the join

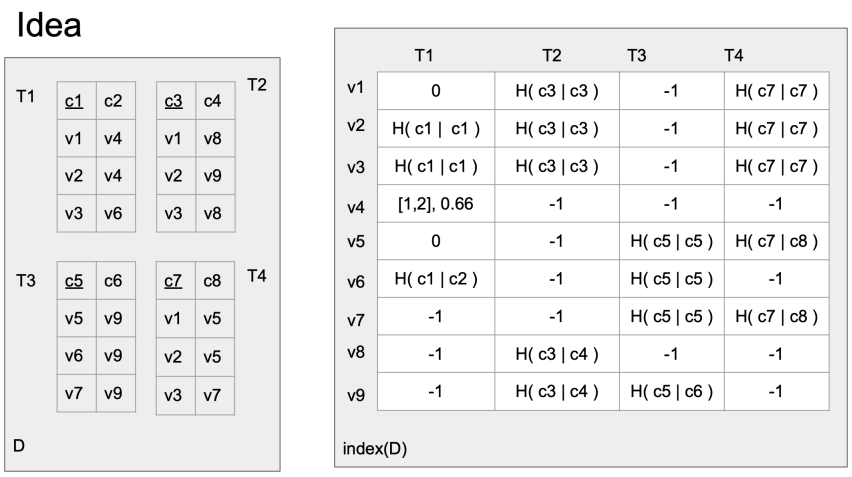


FIGURE 4.1 Idea of the inverted index to store conditional entropy

column (the search column). As long as these two column (join and local key) are similar, the store value approximate well the behavior of the tables.

- Candidate generation: Using the inverted index, generate a list of candidate pairs of columns that have the potential to be joinable. This step involves searching the index for columns with similar value distributions or by filtering rows and columns to prune non-joinable tables.
- Augmentation evaluation: We rank each candidate table according to decreasing order of the index value (the mutual information between the key and the column).
- Augmentation: Perform the horizontal table augmentation using the selected candidate pairs. This step involves joining the tables based on the identified related columns, resulting in an enriched query table with additional relevant features.

Our approach offers several advantages over traditional methods. First, it leverages information theory to provide a more accurate and interpretable measure of relatedness between columns. This helps in identifying the most relevant columns for horizontal table augmentation. Second, the use of an inverted index enables efficient search for joinable tables, resulting in faster query processing and a more scalable solution. Finally, by focusing on the conditional entropy, our approach accounts for dependency-by-chance and provides a statistically reliable estimator for high-dimensional mutual information. Summarizing, the proposed inverted index-based approach seemed to be a novel and efficient method for horizontal table augmentation. By identifying and prioritizing related columns based on their mutual information, this method facilitates the discovery of meaningful relationships between columns and improves the overall quality of the augmented query table. However, it proved to be low effective. This is mainly due to a fact: our idea of storing con-

ditional entropy centered the point, but it did not consider the target column at all. Storing both values would have made the index far too complex and ineffective, although we actually investigated skyline queries to optimize with respect to different dimensions.

Ideal scenario. Figure 4.2 depicts the ideal scenario that we are trying to achieve. The example shows two tables, the one on the left is the query table (J is the join column, T is the target column), while the one on the right shows a candidate table (Jís the candidate join column and C_i is the column that will provide augmentation). In the example, T is a binary column {0,1} where 0 means that the city is in Europe while 1 means it is outside. The diagram represents the probabilities. Even if expressed in terms of probabilities, the step to entropies is straightforward with the content of Chapter 3. Among all, $P(T|J)$ is computable anytime, since both information lie in the query table. If we conduct a strict join search, then we can consider J $\approx$ J; so $P(C_i|J)$ can be considered $\approx P(C_i|J')$. The very relevant information, that cannot be computed is $P(T|C_i)$, the probability of a fixed value of T given a specific value of C_i. Expressed in terms of entropy and conditional entropy, that would express the reduction of uncertainty of T while knowing C_i. The approximation of this quantity is the real missing bit to make the inverted index working. The idea that this quantity cannot be estimated by simple marginals, probability distributions or conditional entropy has been discussed in Chapter 1. We tried to derive such a quantity from the top equation of Figure 4.2, but expanding and substituting values in such an equation does not lead to any conclusion, since there are always two hidden terms. We also tried to put upper and lower bounds to individual quantities, such as conditional entropy by making it collapse to entropy: this is the case in which the values and can be 0 in its lower bound and the entropy value. Again, none of the substitution led a valid estimation of the quantity. We consider this lesson the most significative, although probably the most negative, because is the deeper blind spot in which we ended up but at the same time it opens the door to the following intuition of involving and hash function, that encapsulates in it many probabilistic properties of what we need. But before getting in the direction of hash function, we investigate the possibility of exploiting word embeddings.

Word embedding to map tables. Following the line of previous research that we conducted in past years [21, 26], we also considered the possibility of using word embeddings. Roughly speaking, word embeddings are representation of word as vectors in a high dimensional space. Such dimensionality is high (traditional word embeddings lie in 300-d spaces), but it is very limited if we consider the dimensionality of the space derived by the one-hot-encoding representation a vocabulary. The great power of embeddings is that they are pre-trained on extremely large text corpora, the most used are W2V [65], Glove [70] and Bert [25]. This allows us to use them without the need of a computational power difficult to be afforded. These embeddings naturally encapsulate the semantic meaning of words, being able to solve equations like Paris:France=Rome:x . The embeddings would tell you, by vectorial operations,

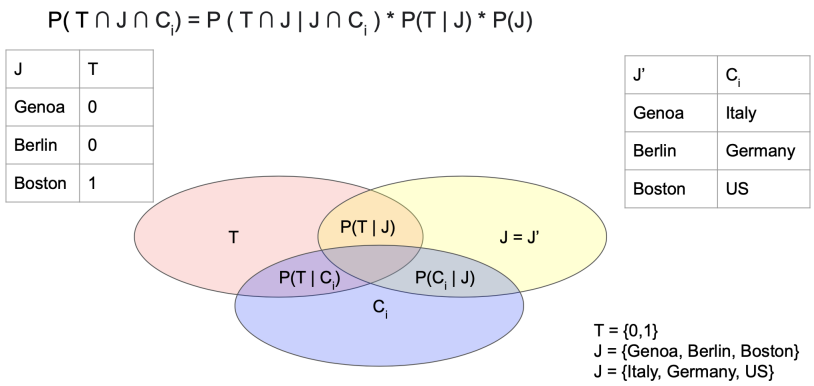


FIGURE 4.2 Ideal scenario

that the vector most likely to the right choice (i.e., the closest) is the vector representing the world Italy.

For element-wise similarity operations, embeddings seem to be very promising and much research is involving them both in explainability and effectiveness. In our case, their usage is surely a matter of investigation which is apart from the scope of this work. We had the idea of filling the inverted index with embedding vectors for table values, but doing that would have not provided any extra information. There exists approaches that actually use word embeddings, such as Pexeso [30] or [6] and [2], where embeddings are basically used to perform join search or schema inference in data lakes. A very good reason for which we did not integrate word embeddings in our work is that *they only maps strings* to vectors. We wanted to implement a system that is able to perform a join only on strings but being able of considering numerical columns while evaluating the augmentation. This is impossible using the nowadays word embedding technologies. Furthermore, word embeddings can still be considered for the join search phase, but the method we proposed somehow encapsulate the same idea in a bit different way. For a matter of completeness, the field of knowledge graphs embedding (KNE) could be a promising bridge in unifying our work to a broader data integration framework, in which tables are not only augmented but also linked to other entities. This step was in principle in our workflow, but the problem of augmentation proved to much harder than what it seemed to be at the time we draw the timeline of this work.

4.5 The TASH Approach: Overview

TASH also includes column augmentation, which allows us to augment a table with missing columns to match the query Q. To perform column augmentation, we first identify the missing columns in the candidate tables. We then search for columns in the data lake that can be augmented to the missing columns in the candidate tables using similarity search. We use the same hashing function

Approach	Pros	Limitations
Functional Dependencies	Helpful in identifying related tables	Hard to compute, not helpful in join search, need to materialize each possible join before discovering
Reliable Fraction of Information	Multi-column correlation among tables	Inefficient to compute, Represent how columns could substitute the target rather than searching for new features, needs to materialize each possible join before discovering
Inverted Index	Join search and augmentation at the same time, ad-hoc data structure to store information theory measures to exactly rank the columns	The required information to be stored would be greater and slower than materializing the join and search table by table without indexing
Embeddings	Semantic information and representation of the values in the tables	In the case of pre-trained embeddings, likely to be low the coverage of values, vectors of word embeddings tend to be high dimensional and unlikely to be at data lake scale

TABLE 4.2 Recap of naive approaches

used for the data lake tables to represent the missing columns and compute the similarity scores between the missing columns and the columns in the data lake.

TASH is not dependent on the chosen hash function. Any suitable hash function can be used as long as it meets the requirements of the algorithm. In fact, the algorithm can be adapted to use any hash function of the user's choice. The hash function is used to create a fingerprint for each column in the tables. The fingerprints are then used to identify joinable columns between tables. TASH is flexible and can work with any hash function because the hashing function is only used to create fingerprints and does not directly influence the joinability of columns. TASH is agnostic to the choice of hashing function, so users can choose the hashing function that works best for their use case. This flexibility allows for a customized and optimized approach to joining tables in a data lake. Additionally, if a user believes that their hashing function works better for their particular use case, they can easily plug in their hashing function into our algorithm.

When using a hash function for TASH, it is important to ensure that the function satisfies certain properties. Firstly, it is essential that columns with

similar values have similar fingerprints. This is what allows us to perform efficient joins based on the similarity of the fingerprints. Secondly, columns with dissimilar distributions must be far apart from each other in the hash space. This is necessary to prevent unrelated columns from being erroneously grouped together. Therefore, if a user chooses to implement their own hash function, it is critical that they are aware of these properties and ensure that their function satisfies them. Failure to do so can result in incorrect or inefficient joins, leading to poor performance and inaccurate results.

In the proposed approach for table augmentation using hashing, each table in the data lake is represented as a matrix of integers obtained from a vocabulary mapping. Specifically, each word in the data lake is assigned an ordinal number, which is then used to construct the matrix. This matrix is then treated as a 2D image and flattened into a 1D array of pixel intensities, where each pixel represents a unique word in the table.

The column vectors of each table can then be thought of as a collection of images, where each image represents a column and each pixel represents a word in that column. These images can then be compared using techniques from image similarity search, such as K-NN hashing. In K-NN hashing, the image vectors are converted into binary fingerprints using a hashing function, which are then stored in a KDTree. This allows for efficient nearest neighbor searches by querying the KDTree for points within a certain radius of the query point.

However, in order to use K-NN hashing, all images must be of the same length. To achieve this, we use padding and resampling to bring each column vector to a standard length. Padding involves adding padding values to the end of the vector to bring it up to the desired length, while resampling involves repeating or deleting pixels to adjust the length of the vector. By using a consistent length for all column vectors, we ensure that the resulting fingerprints are of the same length, making them compatible for nearest neighbor searches.

The output of the hashing algorithm is a matrix of binary fingerprints, where each row represents a unique column in the data lake. We keep a mapping of each fingerprint to the original column and table, allowing us to retrieve the original data when needed. The fingerprints are stored in a KDTree for efficient nearest neighbor searches using radius queries.

Before diving into the details of the approach, we summarize its main components:

- Compute vocabulary of all the values in each table
- Convert each table to the numerical indexing of each value in the vocabulary
- Add padding to make each table of same size
- Compute the hash fingerprints
- Fill the KDTree with fingerprints
- Probe the to augment the Query Table

4.6 The TASH Approach: Index Construction

ID	Name	Age	Gender
1	Alice	25	F
2	Bob	30	M
3	Claire	35	F

ID	Name	Weight
1	Alice	60
2	Bob	70
3	Dave	80

TABLE 4.3 Tables A and B

To illustrate TASH, consider the following example. Suppose we have two tables, Table A and Table B as represented in Table 4.3, we can construct a vocabulary mapping from the words in both tables. Using this vocabulary mapping, we can represent each table as a matrix of integers. For Table A, we get two tables of the same sizes but with new values. Tables converted are represented in Table 4.4.

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	14

1	5	17
2	6	18
3	7	19
4	16	20

TABLE 4.4 Table A and B converted to integers using the vocabulary mapping

In more formal terms, the process can be described as follows. Let $T_1, T_2, ..., T_n$ be the tables in the data lake, and let V be the vocabulary of all distinct words found in the data lake. For each T_i, we create a matrix M_i of size $d_i \times l$, where d_i is the number of distinct words in T_i and l is the length of the longest column in the data lake. Each cell (j, k) of M_i is assigned the ordinal number of the word at that position in the corresponding column of T_i, as determined by the vocabulary mapping.

We then perform padding on each matrix M_i to ensure that they all have the same size $d \times l$. This is necessary for the K-NN hashing algorithm to work, as it requires all input vectors to be of the same length. Padding is done by adding dummy rows or columns of padding values to the matrix as needed, while resampling would involve either upsampling or downsampling the matrix to achieve the desired size. In this work, only padding is investigated while resampling is left as future work.

The resulting matrices $M_1, M_2, ..., M_n$ are then fed into the K-NN hashing algorithm [41], which converts each matrix into a binary fingerprint vector of length k. These fingerprints are stored in a K-d tree, which can be efficiently queried for nearest neighbors within a given radius. The mapping of each fingerprint to its original column and table is also stored, allowing us to retrieve the associated data when a match is found.

To illustrate the process, consider the following example. Suppose we have two tables in the data lake:

ID	Name	Age
1	Alice	25
2	Bob	30
3	Charlie	20

TABLE 4.5 Table T_1

ID	City	Country
1	London	UK
2	Paris	France
3	Berlin	Germany

TABLE 4.6 Table T_2

We first create the vocabulary V = {ID, Name, Age, City, Country, Alice, Bob, Charlie, London, Paris, Berlin, UK, France, Germany} We then convert each table to a matrix as follows:

$$M_1 = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix} \tag{4.1}$$

The resulting binary fingerprints can then be used for similarity search, where given a query table or column, we can find the most similar table or column in the data lake by finding the nearest neighbors in the KDTree. This allows for efficient table augmentation and data exploration, as we can quickly find related tables or columns based on their similarities. It is worth noticing that in the following we will discard the values acting as headers of the tables.

To further improve the search efficiency, we can use techniques such as locality-sensitive hashing (LSH) and random projection, which can reduce the search space by mapping similar tables or columns to the same bucket or projection space. This can significantly speed up the search process, especially when dealing with large datasets with high-dimensional features.

Overall, the approach of using hashing for table augmentation provides a powerful tool for exploring and analyzing large-scale data lakes. By converting tables to binary fingerprints and storing them in a KDTree, we can efficiently perform similarity search and find related tables or columns based on their similarities. This can help to improve data quality, uncover hidden relationships, and facilitate data-driven decision-making.

As an example, consider a data lake containing customer transaction data from a retail store. By converting the transaction tables to binary fingerprints and storing them in a KDTree, we can quickly identify related tables based on their transaction patterns, such as which products are frequently purchased together or which customers tend to make large purchases. This information can then be used to improve the store's marketing strategies, optimize inventory management, and enhance the overall customer experience.

Padding and resampling for hashing algorithm. The matrix generated from the previous step serves as the input for the hashing algorithm. However,

the algorithm requires that all tables have the same length, which can be an issue when dealing with tables of varying sizes. To overcome this problem, we can use padding to ensure that all tables have the same dimensions.

Padding involves adding extra rows to the matrix so that it matches the size of the largest table. The additional rows or columns can be filled with padding values or other values that do not affect the overall structure of the table. For instance, if we have two tables with dimensions 5x5 and 3x3, we can pad the second table with two rows of padding values to make it 5x3.

An example of padding is as follows. We consider two tables: table T_A (on the left of Table 4.7) and table A (in the middle of Table 4.7). In this example the tables have two different dimensions. Table A has dimensions 3x3, while Table B has dimensions 5x3. We can pad Table A with two more rows of padding values to make it 5x3, which now matches the dimensions of Table B. Table A with padding values is depicted on the right of Table 4.7.

ID	Name	Age
1	Alice	25
2	Bob	30
3	Charlie	20

ID	Name	Age
1	Alice	25
2	Bob	30
3	Charlie	20
4	Dave	15
5	Elly	35

ID	Name	Age
1	Alice	25
2	Bob	30
3	Charlie	20
-1	-1	-1
-1	-1	-1

TABLE 4.7 From the left: Table T_A in its original form. T_B in its original form. T_A with two extra row of padding values.

Padding ensures that all tables have the same dimensions, which allows us to apply the hashing algorithm to all tables. Additionally, it preserves the overall structure of the table and does not affect the integrity of the data.

When performing table augmentation using hashing, it is important to ensure that all tables have the same length. This is because the hashing algorithm requires inputs of the same shape to be able to produce the desired outputs. One way to achieve this is through resampling.

Resampling is a process by which we adjust the size of each table so that they are all the same length. The most common way to resample tables is to pad the shorter ones with zeros until they match the length of the longest table. However, this can lead to a significant increase in the size of the input data, which can slow down the hashing algorithm and increase the memory requirements.

4.7 The TASH Approach: Probing

The KDTree index. The basic idea is to use the fingerprints of similar columns to narrow down the search space for join queries.

In the KDTree, each node is associated with a sub-region of the data space. The region is divided along one dimension by a splitting hyperplane, and the two halves of the region are associated with the two child nodes. Each

node contains a set of fingerprints that represent the data points within its sub-region. The fingerprints can be thought of as compact representations of the data points that capture some of their key properties.

To perform a join search query, we start by computing the fingerprints for the query data points. We then perform a radius query in the KDTree to find all nodes whose fingerprints are within a certain distance of the query fingerprints. The distance metric can be any metric that is appropriate for the fingerprint representation, such as the Euclidean distance or the cosine similarity.

Once we have identified the relevant nodes, we can traverse the tree to find the actual data points that match the join condition. To do this, we start at the root node and recursively descend down the tree, choosing the child node whose splitting hyperplane is closest to the join condition. At each node, we check the fingerprints of the data points within the node to see if they match the query fingerprints. If they do, we add them to the set of matching data points.

By using fingerprints to narrow down the search space, we can significantly reduce the computational cost of join queries. However, the effectiveness of this approach depends on the quality of the fingerprints and the choice of distance metric. If the fingerprints are too coarse or the distance metric is inappropriate, we may end up with false positives or miss some relevant data points.

In addition to radius queries, there are other search techniques that can be used with KDTrees, such as k-nearest neighbor queries and range queries. These techniques may be more suitable for some types of join queries, depending on the specifics of the data and the query conditions. Ultimately, the choice of search technique will depend on the characteristics of the data and the performance requirements of the application.

- Join search with radius query on a specific column: The first step is to perform a join search on a specific column, identified as the join search column. This means that we will be searching for matching values in this column between the two tables. We perform a radius query to find all values in the join search column that are within a certain distance (the radius) of the query value. This returns a set of candidates for the join.
- Keeping the candidates in memory: We keep the candidates for the join in memory, as we will be using them in the next step.
- Mapping fingerprints to columns and tables: We use the vocabulary mapping and the fingerprints to map each column in each table to its corresponding integer value. We also keep track of which table each column belongs to. This allows us to easily access all the columns in a table that are linked to a given candidate column.
- Creating a new space for projection: We create a new space in which to perform the next step. This space is constructed by projecting the query table, the candidate columns, and all the columns linked to the candidate

columns. This means that we are creating a new space in which we can compare the columns from the query table to the candidate columns and their linked columns.

- Performing radius queries on candidate columns: We perform a radius query on each candidate column in the new space. This returns a set of columns that are within the specified radius of the candidate column.
- Collecting outliers: We collect all the columns that are outliers in the new space. These are columns that are joinable with the candidate columns (because they belong to joinable tables), but are far away from other joinable columns. These columns contain new information that can be used to augment the join.

Overall, this approach allows us to use fingerprints to efficiently search for joinable columns in large datasets. We start with a radius query to find candidate columns, and then use the mapping between fingerprints and columns/tables to project the data into a new space for comparison. By performing radius queries on the candidate columns in this new space, we can quickly identify columns that are joinable with the candidates and those that contain new information to augment the join.

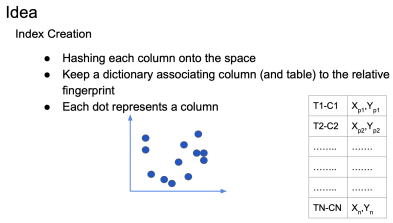
After performing the join search with a radius query on the specific join search column, we have a set of candidate columns that are potentially joinable with the query table. To efficiently explore the space of joinable columns, we exploit the mapping between fingerprints and the actual columns and tables.

We create a new space in which we project the query table, candidate columns, and all columns linked to the candidate columns. This space consists of multiple dimensions, where each dimension corresponds to a column in one of the tables. We use the fingerprints to map the columns to their corresponding dimensions in the space.

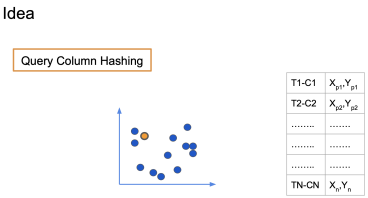
Once we have this space, we perform a radius query on each candidate column. This query returns all the columns that are within a certain radius of the candidate column in the joint space. The columns that are returned are those that are highly correlated with the candidate column and are potentially joinable.

However, we also want to identify the columns that are not highly correlated with the candidate column but are still joinable. We call these columns outliers. We can identify these outliers by looking for columns that are far away from other columns that are highly correlated with the candidate column.

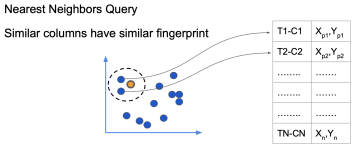
Once retrieved, we order the columns by their euclidean distance from the target column, that has been projected onto the space as well as all other columns of the query table.



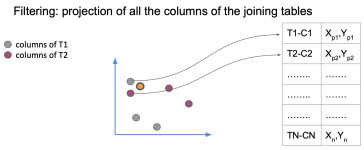
(A) Initial state: the hashing space is full of all columns



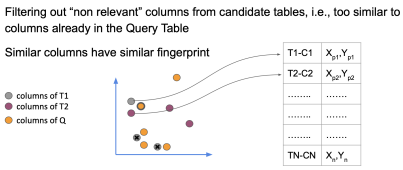
(B) The orange dot represents the join column of the query table



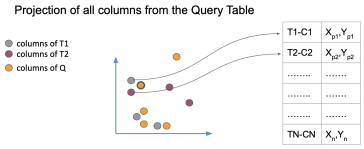
(C) Radius query around the join column. Retrieved table are recognized using the dot-column-table association



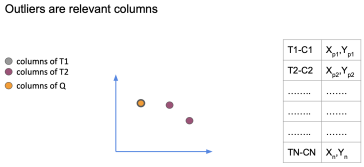
(D) Exploiting the association table, all irrelevant tables are removed from the space



(F) After another radius query for each column of the query table, all the columns inside the radius are discarded because they are likely to be redundant



(E) Columns of the query table are projected



(G) All the residual columns in the space are likely to be relevant

FIGURE 4.3 Steps in the approach

5

Evaluation

Several approaches have been proposed to address the challenge of finding joinable tables in data lakes. One approach is to use schema inference to identify the relationships between tables. This involves analyzing the data in each table to infer the schema and then using this information to identify joinable tables. Another approach is to use metadata management tools to track the relationships between tables. Metadata management tools can track the lineage of data and provide information on the relationships between tables. However, these approaches have limitations. Schema inference can be inaccurate, especially when dealing with semi-structured or unstructured data. Metadata management tools can also be challenging to implement and maintain, especially in large data lakes with multiple data sources.

In our approach, we leverage the unique properties of data fingerprints to identify joinable tables in data lakes. We use a combination of resampling and KD-Tree algorithms to efficiently identify the relationships between tables. Our approach allows us to quickly identify joinable tables in a data lake, even when the data is stored in various formats and locations. By leveraging data fingerprints, we can accurately identify columns that represent the same data, even when the columns have different names.

In this Chapter, we focus on the evaluation of our proposed approach for solving the augmentation problem, and compare it against a baseline approach and other existing methods. Our approach utilizes a novel algorithm that leverages a combination of hashing and statistical features to identify joinable columns between tables that provides an augmentation.

The baseline approach we use is a LSH ensemble search that retrieves joinable columns and orders the tables by containment. This approach has been widely used in the literature as a benchmark for evaluating the performance of new methods. Our proposed approach is expected to outperform the baseline approach due to its use of a different hashing idea.

In this chapter, we present the schema of our evaluation, the datasets used for evaluation, the results of the baseline approach, and a comparison of our proposed approach with other existing methods. We focus on precision and recall as the primary measures of performance evaluation.

First, we describe the schema of our evaluation, which includes the definition of the metrics used to evaluate the performance of the different methods. We explain how we measure the precision and recall of the joinable column search algorithms, and the criteria used to select the datasets for evaluation.

Next, we provide a description of the datasets used for evaluation. We describe the structure of the datasets and the size of the tables. We also explain the selection criteria used to ensure the datasets are representative of real-world scenarios.

Afterwards, we present the results of the baseline approach. We compare the performance of the LSH ensemble search algorithm against our proposed approach, using the same datasets and evaluation metrics. We analyze the results, identifying the strengths and weaknesses of the baseline approach.

Finally, we present a comparison of our proposed approach with other existing methods. We analyze the performance of the different methods in terms of precision and recall, and identify the areas where our approach performs better.

5.1 Evaluation Schema

We evaluate our approach against competitors on different levels. First of all, we discriminate between the approaches that computationally conclude against those that do not. This is relevant because some approaches materialize the join, meaning a large amount of memory and time spent in creating or probing the index. The second level of analysis is qualitative, splitting our evaluation in *join search capability* and *augmentation power*. For what concern the join evaluation, we use two metrics, *precision* and *recall*. With precision, we aim at giving a qualitative analysis of the ability of our approach to retrieve all the tables that it is expected to retrieve. With recall, we want to quantify how many of the table retrieved as "joinable" should have been actually found, measuring the approach ability to filter out tables that are not joinable enough. Finally, we evaluate if we can rank the retrieved tables recognized as joinable ordering them by their contribution in improving the predictive ability of a machine learning task. The learning task is the prediction of a column of the query table.

5.2 Datasets

OPEN. OPEN is a dataset of relational tables from Canadian Open Data Repository [92]. We extract English tables that contain more than 10 rows.

WDC. The WDC 2015 Web Table Corpus [53], contains 10.24 billion genuine tables. The extraction process consists of two steps: table detection and table classification. The percentages of relational, entity, and matrix tables are 0.9%, 1.4%, and 0.03%, respectively. The remaining 97.75% accounts for layout tables. When storing a table, its orientation is also detected, indicating how the attributes are placed. In horizontal tables, the attributes are placed in columns, while in vertical tables they represent rows. There are 90.26 million relational tables in total. Among those, 84.78 million are horizontal and 5.48 million are vertical. The average number of columns and rows in horizontal

Dataset	#tables	#terms	#columns
OPEN	10.2K	17.2M	21.6K
WDC	516K	8.6M	516K

TABLE 5.1 Datalakes statistics. OPEN is used for joinability, WDC is used for both joinability and augmentation search.

tables are 5.2 and 14.45. In vertical tables, these numbers are 8.44 and 3.66, respectively. [53] also extract the column headers and classify each table column as being numeric, string, data, link, boolean, or list. The percentages of the numeric and string columns are 51.4% and 47.3%, respectively. Besides, the text surrounding the table (before and after) is also provided. Furthermore, [53] provide the English-language Relational Subset, comprising relational tables that are classified as being in English, using a naive Bayesian language detector. The language filter considers a table's page title, table header, as well as the text surrounding the table to classify it as English or non-English. The average number of columns and rows in this subset are 5.22 and 16.06 for horizontal tables, and 8.47 and 4.47 for vertical tables. The percentages of numeric and string columns are 51.8% and 46.9%. A total of 139 million tables in the WDC 2015 Web Table Corpus are classified as entity tables. Out of these, 76.70 million are horizontal and 62.99 million are vertical tables. The average number of columns and rows are 2.40 and 9.08 for horizontal tables, and 7.53 and 2.06 for vertical tables. The column data types are quite different from that of relational tables. String columns are the most popular, amounting to 86.7% of all columns, while numeric columns account for only 9.7%.

Ending up in the following situation:

- the dataset of relational tables is made of 50M tables (250 GB size)
- cleaned by metadata, that we do not manage, the total size goes down to 21.5 GB
- applying the same filtering of [30], we extract English tables that contain more than 10 rows, keeping the 33% of the total number of tables, resulting in 16M tables
- on top of the 16M, we filtered out numerical columns to create ground, because we only allow for join search on string column (and LSH ensemble is proved to work on string only columns)

Machine Learning Task. We also identified two datasets (tables) that have a bunch of join matches (according to LSH with threshold > 0.8) in a subset of WDC tables. These two dataset have a column that act as target, solving two classification tasks.

Amazon toy product classification contains 10,000 rows of toy products from Amazon.com [1]. The task is to predict the category from 39 classes (hobbies, office, arts, etc.) of each toy. We use "product name" as query column.

1 https://www.kaggle.com/PromptCloudHQ/toy-products-on-amazon

Video game sales regression contains 11,493 rows of video games with attributes and sales information [2]. The task is to predict the global sales. We use "Name" (game's name) as query column.

Preprocessing. Many steps of preprocessing have been performed on the datasets. Mainly, it has been necessary to uniform the upper and lower cases, since hashing is sensible to such an information. Secondly, stopwords have been removed as well, since they generally act as noise and do not provide any new information. Furthermore, stopwords in the join search column would make the search harder and longer.

5.3 Comparison with Existing Approaches

In this section, we compare our proposed approach with some existing approaches. As we will see, there is no direct comparison among us and our competitors, since, to the best of our knowledge, the statement of our problem has not been tackle yet in literature.

5.3.1 Baseline

The first evaluation we made, in order to test the capability of our approach to retrieve joinable tables, was to compute the fingerprints of tables that do not belong to any data lake. We did this because we wanted to check the ability of the hashing function to actually stick close in the hashing space columns with similar values. We identified a table containing information about the United States, whose join column was the list of all states. Furthermore, we retrieved 17 tables from the web that shared the same column. Some tables had the very same column, while others had slight different values, such as missing or extra states (for example, some tables presented Puerto Rico as a state). We preprocessed them by applying the transformation to lowercase and removing stopwords and then computed the fingerprints with [41]. Once we queried the KDTree to get the closest point to the query column, we retrieved all the expected columns, identifying all the 17 tables as joinable. After that, we include 100 random tables from OPEN, to be hashed in the same way. We did so to check if the hashing algorithm was overlearning the representation. Results confirmed the joinability for the expected tables and discarded 95% of the others, meaning that the hashing algorithm is able to catch the similarity among columns.

Secondly, we randomly selected 30 tables from each of the datasets (OPEN and WDC) and use them as query table to test the precision and recall ability of our approach. Since LSH Ensemble is considered as state-of-the-art for joinability discovery, we used it as ground truth. Results are reported in Table 5.2. The table compares ourselves with the ground truth and Pexeso. As well as LSH Ensemble, we are able to retrieve all the joinables tables expected to be

2 https://www.kaggle.com/gregorut/videogamesales

	OPEN		WDC	
Method	Precision	Recall	Precision	Recall
LSH Ensemble	1.000	0.611	1.000	0.589
Pexeso	0.911	0.821	0.948	0.868
TASH	1.000	0.722	1.000	0.747

TABLE 5.2 Precision and recall with LSH Ensemble, Pexeso and our approach

Method	#Match	Micro-F1
No-join	-	0.589 ± 0.077
Equi-join	0.13%	0.586 ± 0.051
Jaccard-join	0.54%	0.585 ± 0.073
TF-IDF join	0.72%	0.594 ± 0.049
Pexeso	0.76%	0.613 ± 0.072
TASH	0.89%	0.604 ± 0.041

TABLE 5.3 Amazon toy product classification

retrieved. A partial limitation is the recall, since it shows that our approach consider some tables as joinable, while they should not be. This fact opens an interesting distinction in our framework: evaluation the algorithm itself, meaning the steps, and the selected tool to implement it. This is a limitation due to the chosen hash function, that we remind it adapted from image similarity [41]. The schema we propose is agnostic from the chosen hash function, making it possible for a user to plug in a custom function.

5.3.2 Our Approach

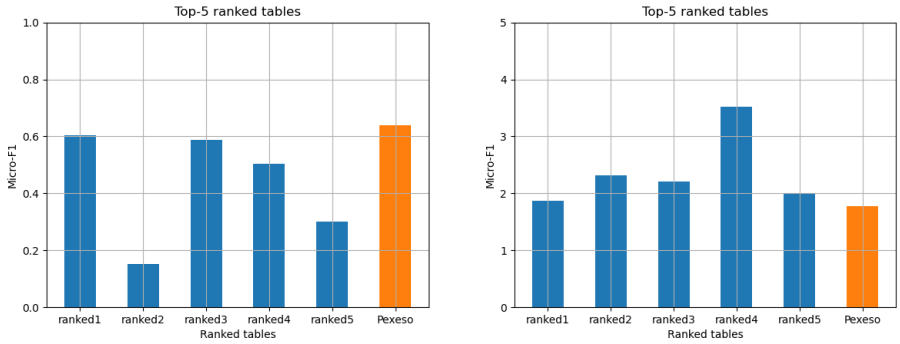
To evaluate the ability of our approach in improving the predictive ability of the query table, we ran our framework on a subset of WDC, that we will call SWDC. For a matter of comparison with [30], we randomly select 1000 rows from the query tables and search for joinable. We retrieve the joinable tables and then rank them according to the number of columns that act as outliers, i.e., we rank higher the joinable tables with higher number of columns that results from the Project-Filter-Search steps of our framework. Table 5.3 shows the performance results for the Amazon toy product classification. Accordingly with previous results, the percentage of joinable tables discovered is low in percentage but higher respect to other approaches. This is due to the fact that the hashing function puts closer columns that should not be. However, we achieve good results, comparable with [30]. Our method results in an augmentation which is less accurate than [30], but comparable. The advantage of our method is that it is faster and cheaper in terms of memory, since our binary representation is lighter than their embedding counterpart. Similarly, Table 5.4 shows good results in the regression task, showing that our method is suitable for both classification and regression tasks.

Figure 5.1 shows the top-5 ranked tables retrieved by our solution. On the

Method	#Match	Micro-F1
No-join	-	2.09 ± 0.75
Equi-join	0.05%	2.05 ± 0.65
Jaccard-join	0.14%	1.98 ± 0.73
TF-IDF join	0.31%	2.02 ± 0.55
Pexeso	0.64%	1.78 ± 0.66
TASH	0.89%	1.87 ± 0.42

TABLE 5.4 Video game sale regression

left, the closer to 1 means the better performance. We report as best result the same table of Table 5.3, and the others four accordingly. As expected, the top-1 ranked table is the one performing better. This not true for all other tables, meaning that the hash function is not perfect. The same thing happens for the second task, with values closer to zero meaning a better result. For both experiments we reported the result of [30], although in their research there is no indication about which table performs better or if the prediction results are the aggregation of different tables via multiple joins.



(A) Top-5 ranked tables for Amazon toy product classification. The closer to 1 the better

(B) Top-5 ranked tables for Video game sale regression. The closer to 0 the better

FIGURE 5.1

Parameters tuning. In order to achieve the presented results, we had to deal with different parameters tuning phases. The first parameter to tune was the threshold for LSH-Ensemble. We recall that we used LSH-Ensemble as ground-truth for identifying tables as joinable. We tried different threshold values $(0.7, 0.8, 0.9)$ opting for 0.8 since it was the last value allowing us to consider joinable similar tables without the constraints of being too similar, as supposed with higher thresholds. For what concerns the generation of the fingerprints, there two main parameters to tune: the k value, meaning the number of neighbors used in the learning of hash fingerprints, and n, meaning the number of bits of the fingerprints. For what concerns k, the authors [41]

themselves suggest tuning such a parameter ad-hoc for the reference dataset, ranging from a lower value of 5 up to 50. This parameter proved to have a great influence on the execution time of the framework, posing us in front of trade-off choice between accuracy of the fingerprints and efficiency. We tested 5, 10, 15 and 20 ending up with choosing 10, because it allowed us to have good joinability performances at a reasonable time cost. For what concern the choice of n, we tested many possibilities using a joinability problem as evaluation. The dimension we tested are 16, 32, 64, 128 and 256, choosing 128 because it was a cheaper solution to 256 while showing very similar performances. The latest tuning required was the radius of the query in the KDTree. We experimentally choose 0.3 as a value.

The problem of augmenting tables in data lake is a task that many researchers are facing at this time. Since many approaches focus on different aspects, we tried to summarize and classify them. Table 5.5 reports the results of our investigation. It is worth noticing that it is hard to compare to the approaches actually tackling the augmentation problem for many reasons, among which:

- There are approaches that only focus on showing the theoretical feasibility of an approach without testing the method;
- There are approaches that only focus on evaluating the proposed approach in terms of time execution, without quantifying the quality of the provided augmentation;
- There are approaches that prove themselves to be effective on relatively small data lakes, closing their scenario to a bunch of tables among which search and materializing all the possible joins.

It is worth noticing that all the approaches providing augmentation work under the closed world assumption. Meaning that it is possible to select as query table only a table that is already in the data lake at the time of creation of the index or the infrastructure that allows for the search. This is not true for all join search algorithms that do not provide augmentation.

	Heuristic	**Index**	**Hash**	**Embedding**	**Learning**
Joinability	-	[90, 13]	[92]	[30, 12, 80, 6, 31]	[57]
Unionability	-	[49, 64, 27, 46, 13]	-	[11, 12, 6]	[31]
Augmentation	[76, 51]	[33, 18, 29]	[75]	[11, 12]	[88, 57]

TABLE 5.5 When to use state-of-the-art methods for a specific problem

5.3.3 Limitations

As strongly dependent on the hash function, our framework is deeply influenced by the chosen hash function performance. Furthermore, since the hash function encapsulates the distribution probability of the elements of each column, the quality evaluation must be done at a different time by actually materializing the joins between the query table and each candidate. This is

mitigated by the fact that our framework is expected to return a list of tables *that can be joined with the query table and produce an augmentation* rather than joining itself. Another limitation is that our framework only works table-to-table. This means that it is not able to evaluate if a group of tables together could perform better that a single join. This is the reason for which Figure 5.5 shows individual performance of the retrieved tables. Such a behavior is similar to what happens in feature selection algorithms, where there are filter methods that evaluate the relevance of a feature to a target and wrapper methods, that evaluate the performance of a subset of feature with respect to the target. Another interesting point is that we only use the padding technique to fit each table to the same size, as prerequisite for the fingerprint generation algorithm. Surprisingly, it worked pretty well. We conjecture that this is due to the fact the tables used as query tables were of the average size of the tables in the data lake. Tables with higher or lower number of rows might perform worst with the padding. This is because the padding itself is recognized by a valid value from the algorithm, meaning that a predominance of the padding value in a column can ruin the representation. We redirect to Chapter 6 for further discussions on this element.

6

Conclusion ad Future Work

In this thesis, we presented an algorithm for searching joinable tables in data lakes and ranked all candidate tables by their ability of improving the predictive ability of a query table. We showed how hashing and information theory are fundamental to solve the problem, and we provide results that are competitive with the state-of-the-art in this field.

Contributions. We presented an approach, named TASH, based on an indexing of a data lake, which can be built in a fast way and it is not too space consuming, such that the index makes it probable to identify joinable tables that provide augmentation. We exploited two different concepts, hashing and information theory, to build such a framework. The data lake is primarily converted to an ordered dictionary, so that each column of each table is represented as a vector of identical size with the ordinal id of the term in the vocabulary. Each column is then treated as an image and hashed to generate a binary fingerprint. Each fingerprint is then stored into a spatial index, on which it is possible to execute a radius query to get joinable tables with respect to a query table. Each joinable table is ranked according to its relevance in augmenting the predictive ability of the original query table. The approach allows for one-to-one join search, and the search for joinable tables is limited to string columns only. Our approach is agnostic of the content of all other columns, but it does not discard them, because they can contain relevant numerical information for improving the performances of the query table. We demonstrated that it is possible to encapsulate the steps of searching for a joinable table and evaluate its contribution in improving the performance of a Machine Learning model in a single framework without the need to materialize the join for each table. The evaluation shows that we are competitive with state-of-the-art approaches that solve the same problem (or a single aspect of our problem) using fully explainable algorithms, such as hashing, rather than relying on deep learning techniques. It has to be clear that our approach solves a particular case of augmentation, the one-to-one join between columns, and it has been tested on basic machine learning tasks. The infrastructure seems to be scalable to very large data lakes, as long as the machine on which the framework will be run is powerful enough to handle large vocabularies.

Limitations. As we have seen, TASH is able to identify tables that can be joined and that provide an augmentation. There are, however, several aspects of our work that open up significant possibilities for further investigation. The hash function is a pluggable component of the framework. This can be

seen as a feature of our approach, because it allows a user to plug her own custom function to solve the task. On the other hand, the framework is very dependent on the quality of the hash function. Plugging a function unable of catching similarity among columns while keeping the information about the distribution of data in the fingerprint would result in very poor results.

Ranking individual tables. As it is known in feature selection, the contribution of two individual features alone is different from the contribution that the same features would provide together.

Padding and resampling. Even if padding seems to be pretty effective, for very large tables as input the amount of padding values might dramatically decrease the performances. This is due to the fact that the number of padding values might overcome the content of the column, ruining both the joinability search and the Machine Learning performance. A more effective solution could be considering to filling the spaces left blank by the size difference with a resampling of the values that already belong to the column. This would allow the system not to increase the vocabulary size (since values would be recycled) and to preserve as much as possible the distribution of the data inside the columns. By preserving the distribution, the hash function would then be free to exploit its power without dealing with padding values.

Static indexing. Another limitation is given by the fact that it is not possible to index a new table on the fly. Once the fingerprints are created, the framework can only be probed but not updated. This is due to the fact that, in order to create the fingerprints, all columns must be processed, i.e., the fingerprint of each table depends on the whole set of columns. This is true also for other competitor approaches. Approaches that rely on pre-trained embeddings or knowledge graphs mitigate a bit this problem at the cost that not all the values might be mapped to entities, resulting in a partial representation of the input table.

Future Work. As the research in tackling the problem we discussed in this thesis is florid, we are in an early stage of progress. So, there are many possible ways of extending and improving this work. In the following, we will discuss some possible future directions.

Test new hash functions. The most obvious extension of this work could be to use different hash function that guarantee the required properties of catching similarity and distribution. The whole world of *Learning to Hash* functions is florid as well and the integration of Machine Learning and Deep Learning algorithms for hashing is making this area interesting and proficient. The core of the TASH framework is that the hash function is pluggable, so that it will be easy to integrate a custom function in the workflow.

Split indexes by table size. To overcome the padding vs resampling limitation, a possible solution could be to create many groups of tables of similar size to be given as input to the hash algorithm. This step would bring many advantages, among which: each group would include a subset of all the tables, each fingerprint would be more precise since less padding or resampling would be required. At this point, a query table could be queried against many different

partitions, even by sampling elements of the query table or adding padding to fit well. Instead of padding or resampling tables to a fixed length, we can map each table to the index that covers its length range, and then resample the table to the length of the corresponding index. This can significantly reduce the amount of padding or resampling needed, leading to a more efficient and effective hashing process. To illustrate this, consider an example where we have two tables, A and B. Table A has a length of 4, while table B has a length of 3. Instead of padding table B with zeros to match the length of table A, we can map table B to an index that covers lengths 1-4, and then resample it to a length of 4. Similarly, we can map table A to an index that covers lengths 1-4 and resample it to a length of 4 as well. This way, both tables are the same length, and we have minimized the amount of padding and resampling required. Furthermore, having multiple indexes can also help to improve the accuracy of the hashing algorithm. By mapping tables to indexes based on their length range, we can ensure that similar tables are mapped to the same index, which can lead to more accurate results. This is because tables that are similar in length are more likely to be similar in content as well, and thus should be treated similarly by the hashing algorithm. In summary, by using multiple indexes to cover a range of table lengths, it would be possible to minimize the amount of padding and resampling required during the table augmentation process. This can lead to a more efficient and effective hashing algorithm.

Incorporate in a DBMS to run a SQL query rather than relying on KD-tree. The choice of using a spatial index can be substituted by incorporating the fingerprints in a DBMS and using plain SQL queries to retrieve the results.

7

Other Contributions

In this chapter, we briefly discuss other works that have been published during the years of the PhD. The works lie in a bit of different area than the content of this thesis, they are mainly concerned with geographical knowledge and Geo-Names. They are centered on the role of embeddings in representing geographical concepts, like geographical ontologies and taxonomies. The core project is presented in [26], an consists of an algorithm for inferring the position of a tweet based on the Geo-Names (i.e., names referring to objects with a geographical position) in its content. The main contribution in this work is designing and developing the usage of word embeddings in the representation of geographical objects in a small area, like that of a city, as embedding vectors. We developed a complete evaluation of the embedding technique able of qualitatively characterizing the embedding. This quality study has been the subject of further publications [21, 22].

Some of the concepts we developed in these works could be applied in the table augmentation setting for the problem we faced in this thesis. However, the main problem arising when trying to use them in for the problem faced in the thesis is that embeddings encapsulate semantic information that can be useful, but it presents some limitations:

- Ideally, the embedding representation can be alternative to the hash representation. The problem is that embeddings are at value-level, while our framework works at both column and table level.
- Previous limitation suggests that the hashing should be computed on the vectorial representation, adding an extra level of complexity. Alternatively, the column representation could be the mean of values in each column, making it harder to capture similarity between tables of different length.
- Euclidean word embeddings lie in high dimensional spaces. This would make the index structure much larger and potentially intractable.
- To exploit embeddings, a entity matching layer should be introduced. Since Euclidean embeddings are pre-trained, it is possible that the coverage of entities in the data lake is poor. This is true also considering knowledge graphs techniques.
- Embeddings can be used on string columns only, that is the case of join search, but it is not the case once augmentation is evaluated.
- Hyperbolic embeddings do not seem to be suitable for the problem, since there is a lack of scalability in computing and there is a hierarchical

structure in the data lake.

Bibliography

[1] Naser Ahmadi, Hansjörg Sand, and Paolo Papotti. Unsupervised Matching of Data and Text. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 1058–1070. Cited on p. 17.

[2] Nour Alhammad, Alex Bogatu, and Norman W Paton. Towards Schema Inference for Data Lakes. In: *arXiv preprint arXiv:2206.03881* (2022). Cited on p. 44.

[3] Alexandr Andoni, Ilya Razenshteyn, and Negev Shekel Nosatzki. Lsh forest: Practical algorithms made theoretical. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2017, pp. 67–78. Cited on p. 18.

[4] Lars Arge, Mark de Berg, Herman Haverkort, and Ke Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In: *ACM Transactions on Algorithms (TALG)* 4.1 (2008), pp. 1–30. Cited on p. 31.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 1990, pp. 322–331. Cited on p. 31.

[6] Alex Bogatu, Alvaro AA Fernandes, Norman W Paton, and Nikolaos Konstantinou. Dataset discovery in data lakes. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020, pp. 709–720. Cited on pp. 15, 44, 59.

[7] Rajesh Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities. In: *arXiv preprint arXiv:1712.07199* (2017). Cited on p. 16.

[8] Rajesh Bordawekar and Oded Shmueli. Exploiting Latent Information in Relational Databases via Word Embedding and Application to Degrees of Disclosure. In: *CIDR*. 2019. Cited on p. 16.

[9] Rajesh Bordawekar and Oded Shmueli. Using word embedding to enable semantic queries in relational databases. In: *Proceedings of the 1st workshop on data management for end-to-end machine learning*. 2017, pp. 1–4. Cited on p. 16.

[10] Michael J Cafarella, Alon Halevy, and Nodira Khoussainova. Data integration for the relational web. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1090–1101. Cited on p. 15.

[11] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1335–1349. Cited on pp. 17, 59.

[12] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Embdi: generating embeddings for relational data integration. In: *Proceedings of the 29th Italian Symposium on Advanced Database Systems, SEBD*. 2021, pp. 331–338. Cited on pp. 17, 59.

[13] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. Auctus: a dataset search engine for data discovery and augmentation. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 2791–2794. Cited on pp. 18, 59.

[14] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A Data Discovery System. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1001–1012. Cited on p. 14.

[15] Adriane Chapman, Elena Simperl, Laura Koesten, George Konstantinidis, Luis Daniel Ibáñez, Emilia Kacprzak, and Paul Groth. Dataset search: a survey. In: *VLDB J.* 29.1 (2020), pp. 251–272. Cited on p. 14.

[16] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 2002, pp. 380–388. Cited on p. 17.

[17] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In: *International Conference on Learning Representations*. 2021. Cited on p. 18.

[18] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. ARDA: Automatic Relational Data Augmentation for Machine Learning. In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1373–1387. Cited on pp. 18, 59.

[19] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–36. Cited on p. 17.

[20] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. Finding Related Tables. In: SIGMOD '12. Association for Computing Machinery, 2012, 817–828. Cited on pp. 14, 15.

[21] Federico Dassereto, Laura Di Rocco, Giovanna Guerrini, and Michela Bertolotto. Evaluating the effectiveness of embeddings in representing the structure of geospatial ontologies. In: *Geospatial Technologies for Local and Regional Development: Proceedings of the 22nd AGILE Conference on Geographic Information Science 22*. 2020, pp. 41–57. Cited on pp. 43, 65.

[22] Federico Dassereto, Laura Di Rocco, Shanley Shaw, Giovanna Guerrini, and Michela Bertolotto. How to Tune Parameters in Geographical Ontologies Embedding. In: Association for Computing Machinery, 2020. Cited on p. 65.

[23] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In: *Proceedings of the twentieth annual symposium on Computational geometry*. 2004, pp. 253–262. Cited on p. 17.

[24] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. In: *arXiv preprint arXiv:1704.04738* (2017). Cited on p. 14.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *arXiv preprint arXiv:1810.04805* (2018). Cited on p. 43.

[26] Laura Di Rocco, Federico Dassereto, Michela Bertolotto, Davide Buscaldi, Barbara Catania, and Giovanna Guerrini. Sherloc: a knowledge-driven algorithm for geolocating microblog messages at sub-city level. In: *International Journal of Geographical Information Science* 35.1 (2021), pp. 84–115. Cited on pp. 43, 65.

[27] Hong-Hai Do and Erhard Rahm. COMA—a system for flexible combination of schema matching approaches. In: *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. 2002, pp. 610–621. Cited on p. 59.

[28] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. In: *arXiv preprint arXiv:1511.00628* (2015). Cited on p. 31.

[29] Yuyang Dong and Masafumi Oyamada. Table Enrichment System for Machine Learning. In: *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2022, pp. 3267–3271. Cited on p. 59.

[30] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2021, pp. 456–467. Cited on pp. 2, 18, 44, 55, 57–59.

[31] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. In: *arXiv preprint arXiv:2212.07588* (2022). Cited on p. 59.

[32] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1454–1467. Cited on p. 16.

[33] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. COCOA: COrrelation COefficient-Aware Data Augmentation. In: *EDBT*. 2021, pp. 331–336. Cited on p. 59.

[34] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. In: *IEEE Transactions on Knowledge and Data Engineering* 23.5 (2010), pp. 683–698. Cited on p. 37.

[35] Wenfei Fan, Liang Geng, Ruochun Jin, Ping Lu, Resul Tugay, and Wenyuan Yu. Linking Entities across Relations and Graphs. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 634–647. Cited on p. 17.

[36] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. Discovering graph functional dependencies. In: *ACM Transactions on Database Systems (TODS)* 45.3 (2020), pp. 1–42. Cited on p. 37.

[37] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In: *Proceedings of the 2016 international conference on management of data*. 2016, pp. 1843–1857. Cited on p. 37.

[38] Raul Castro Fernandez, Essam Mansour, Abdulhakim A Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 989–1000. Cited on p. 15.

[39] Maayan Frid-Adar, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Synthetic data augmentation using GAN for improved liver lesion classification. In: *2018 IEEE 15th international symposium on biomedical imaging (ISBI 2018)*. 2018, pp. 289–293. Cited on p. 3.

[40] Michael Günther, Maik Thiele, Erik Nikulski, and Wolfgang Lehner. RetroLive: Analysis of Relational Retrofitted Word Embeddings. In: *EDBT*. 2020, pp. 607–610. Cited on p. 17.

[41] Xiangyu He, Peisong Wang, and Jian Cheng. K-nearest neighbors hashing. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2839–2848. Cited on pp. 47, 56–58.

[42] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. In: *The computer journal* 42.2 (1999), pp. 100–111. Cited on p. 37.

[43] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 604–613. Cited on p. 17.

[44] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J Miller, and Mirek Riedewald. SANTOS: Relationship-based Semantic Table Union Search. In: *arXiv preprint arXiv:2209.13589* (2022). Cited on p. 15.

[45] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. Integrating Data Lake Tables. In: *Proc. VLDB Endow.* 16.4 (2022), 932–945. Cited on p. 14.

[46] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J Miller. Integrating Data Lake Tables. In: *Proceedings of the VLDB Endowment* 16.4 (2022), pp. 932–945. Cited on p. 59.

[47] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 2002, pp. 546–557. Cited on p. 31.

[48] Christos Koutras, Marios Fragkoulis, Asterios Katsifodimos, and Christoph Lofi. REMA: Graph Embeddings-based Relational Schema Matching. In: *EDBT/ICDT Workshops*. 2020. Cited on p. 17.

[49] Christos Koutras, Kyriakos Psarakis, George Siachamis, Andra Ionescu, Marios Fragkoulis, Angela Bonifati, and Asterios Katsifodimos. Valentine in action: matching tabular data at scale. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 2871–2874. Cited on pp. 18, 59.

[50] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 759–772. Cited on p. 37.

[51] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. 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*. 2016, pp. 19–34. Cited on pp. 5, 14, 59.

[52] Oliver Lehmberg and Christian Bizer. Stitching web tables for improving matching quality. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1502–1513. Cited on p. 15.

[53] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A Large Public Corpus of Web Tables Containing Time and Context Metadata. In: *Proceedings of the 25th International Conference Companion on World Wide Web*. 2016, 75–76. Cited on pp. 54, 55.

[54] Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. The mannheim search join engine. In: *Journal of Web Semantics* 35 (2015), pp. 159–166. Cited on p. 14.

[55] Aristotelis Leventidis, Laura Di Rocco, Wolfgang Gatterbauer, Renée J Miller, and Mirek Riedewald. DomainNet: Homograph Detection for Data Lake Disambiguation. In: *EDBT 2021* (2021). Cited on p. 36.

[56] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, Jin Wang, Wataru Hirota, and Wang-Chiew Tan. Deep Entity Matching: Challenges and Opportunities. In: *J. Data and Information Quality* 13.1 (2021). Cited on p. 17.

[57] Jiabin Liu, Chengliang Chai, Yuyu Luo, Yin Lou, Jianhua Feng, and Nan Tang. Feature augmentation with reinforcement learning. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 3360–3372. Cited on p. 59.

[58] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. Discovering dependencies with reliable mutual information. In: *Knowledge and Information Systems* 62.11 (2020), pp. 4223–4253. Cited on p. 37.

[59] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. Discovering Reliable Correlations in Categorical Data. In: *2019 IEEE International Conference on Data Mining (ICDM)*. 2019, pp. 1252–1257. Cited on p. 37.

[60] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. Discovering Reliable Dependencies from Data: Hardness and Improved Algorithms. In: *IEEE International Conference on Data Mining, ICDM*. 2018, pp. 317–326. Cited on p. 37.

[61] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. In: *Proceedings of the VLDB Endowment* 9.9 (2016), pp. 636–647. Cited on p. 15.

[62] Heikki Mannila and Kari-Jouko Räihä. On the complexity of inferring functional dependencies. In: *Discrete Applied Mathematics* 40.2 (1992), pp. 237–243. Cited on p. 34.

[63] Giovanni Mariani, Florian Scheidegger, Roxana Istrate, Costas Bekas, and Cristiano Malossi. BAGAN: Data Augmentation with Balancing GAN. In: *International Conference on Machine Learning*. 2018. Cited on p. 3.

[64] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: *Proceedings 18th international conference on data engineering*. 2002, pp. 117–128. Cited on p. 59.

[65] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In: *arXiv preprint arXiv:1301.3781* (2013). Cited on p. 43.

[66] Justin J Miller. Graph database applications and concepts with Neo4j. In: *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*. Vol. 2324. 36. 2013. Cited on p. 37.

[67] Renée J Miller. Open data integration. In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 2130–2139. Cited on pp. 2, 5, 14.

[68] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. Table union search on open data. In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 813–825. Cited on pp. 2, 15.

[69] Hiromitsu Nishizaki. Data augmentation and feature extraction using variational autoencoder for acoustic modeling. In: *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. 2017, pp. 1222–1227. Cited on p. 3.

[70] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543. Cited on p. 43.

[71] Jianbin Qin and Chuan Xiao. Pigeonring: A Principle for Faster Thresholded Similarity Search. In: *Proc. VLDB Endow.* 12.1 (2018), 28–42. Cited on p. 16.

[72] Franck Ravat and Yan Zhao. Data lakes: Trends and perspectives. In: *Database and Expert Systems Applications: 30th International Conference, DEXA*. 2019, pp. 304–313. Cited on p. 1.

[73] Dominique Ritze, Oliver Lehmberg, Yaser Oulabi, and Christian Bizer. Profiling the Potential of Web Tables for Augmenting Cross-domain Knowledge Bases. In: *Proceedings of the 25th International Conference on World Wide Web, WWW*. 2016, pp. 251–261. Cited on p. 15.

[74] Hanan Samet. The quadtree and related hierarchical data structures. In: *ACM Computing Surveys (CSUR)* 16.2 (1984), pp. 187–260. Cited on p. 31.

[75] Aécio Santos, Aline Bessa, Christopher Musco, and Juliana Freire. A sketch-based index for correlated dataset search. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 2928–2941. Cited on pp. 19, 59.

[76] Vraj Shah, Arun Kumar, and Xiaojin Zhu. Are Key-Foreign Key Joins Safe to Avoid when Learning High-Capacity Classifiers? In: *Proceedings of the VLDB Endowment* 11.3 (2017). Cited on pp. 14, 59.

[77] Claude E Shannon. A mathematical theory of communication. In: *The Bell system technical journal* 27.3 (1948), pp. 379–423. Cited on p. 26.

[78] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. In: *Artificial intelligence and statistics*. 2014, pp. 886–894. Cited on p. 17.

[79] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çağatay Demiralp, Chen Chen, and Wang-Chiew Tan. Annotating Columns with Pre-Trained Language Models. In: SIGMOD '22. Association for Computing Machinery, 2022, 1493–1503. Cited on p. 17.

[80] Sahaana Suri, Ihab F Ilyas, Christopher Ré, and Theodoros Rekatsinas. Ember: no-code context enrichment via similarity-based keyless joins. In: *Proceedings of the VLDB Endowment* 15.3 (2021), pp. 699–712. Cited on p. 59.

[81] Waldo Tobler and Zi-tan Chen. A quadtree for global information storage. In: *Geographical Analysis* 18.4 (1986), pp. 360–371. Cited on p. 31.

[82] Sandhya Tripathi, Bradley A Fritz, Mohamed Abdelhack, Michael S Avidan, Yixin Chen, and Christopher R King. Deep Learning to Jointly Schema Match, Impute, and Transform Databases. In: *arXiv preprint arXiv:2207.03536* (2022). Cited on p. 17.

[83] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. InfoGather: Entity Augmentation and Attribute Discovery by Holistic Matching with Web Tables. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, 97–108. Cited on p. 14.

[84] Jing Zhang, Bonggun Shin, Jinho D Choi, and Joyce C Ho. SMAT: An attention-based deep learning solution to the automation of schema matching. In: *Advances in Databases and Information Systems: 25th European Conference, ADBIS*. 2021, pp. 260–274. Cited on p. 17.

[85] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. Automatic discovery of attributes in relational databases. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011, pp. 109–120. Cited on p. 15.

[86] Shuo Zhang and Krisztian Balog. Web Table Extraction, Retrieval, and Augmentation: A Survey. In: *ACM Trans. Intell. Syst. Technol.* 11.2 (2020), 13:1–13:35. Cited on p. 14.

[87] Shuo Zhang and Krisztian Balog. Web table extraction, retrieval, and augmentation: A survey. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 11.2 (2020), pp. 1–35. Cited on p. 15.

[88] Yi Zhang and Zachary G. Ives. Finding Related Tables in Data Lakes for Interactive Data Science. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*. 2020, pp. 1951–1966. Cited on p. 59.

[89] Bolong Zheng, Zhao Xi, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S Jensen. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. In: *Proceedings of the VLDB Endowment* 13.5 (2020), pp. 643–655. Cited on p. 18.

[90] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*. 2019, pp. 847–864. Cited on pp. 16, 59.

[91] Erkang Zhu, Yeye He, and Surajit Chaudhuri. Auto-join: Joining tables by leveraging transformations. In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1034–1045. Cited on p. 16.

[92] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. LSH Ensemble: Internet-Scale Domain Search. In: *Proc. VLDB Endow.* 9.12 (2016), pp. 1185–1196. Cited on pp. 8, 15, 54, 59.