Fachbereich Informatik

Bachelor's Thesis

Design and Implementation of a Distributed Dataset Search Infrastructure

Author:
Nhat Anh Pham

submitted on:
June 17, 2024

Supervisor
Prof. Dr.-Ing. Sebastian Michel

Reviewers

.....

Prof. Dr.-Ing. Sebastian Michel

M.Sc. Patrick Hansert

Abstract

In a distributed environment, where servers communicate via wireless networks and each server hosts a collection of datasets, finding relevant information across servers can be challenging. If a user from one server wants to identify datasets from other servers that can be joined or unioned with their own, traditional search systems would fall short because they don't consider the attributes of the datasets. Moreover, it is infeasible to transmit entire datasets across the network to find potential matches due to their large size. In this thesis, we introduce a lookup infrastructure for relational datasets that allow users to query with a dataset and receive results from other servers that are joinable/unionable with the query. Our system utilizes different measures to create compact summaries the datasets, which is significantly smaller in size than the original data. The summaries are then used to compare datasets for similarities. Through empirical evaluation, we show that the system is efficient in finding related datasets and can provide comparable response time to state-of-the-art dataset search systems, while minimizing the network traffic.

Zusammenfassung

In einer verteilten Umgebung, in der Server über drahtlose Netzwerke kommunizieren und jeder Server eine Sammlung von Datensätzen hostet, kann das Finden relevanter Informationen über Server hinweg eine Herausforderung darstellen. Wenn ein Benutzer von einem Server Datensätze von anderen Servern identifizieren möchte, die mit seinen eigenen zusammengeführt oder vereinigt werden können, stoßen herkömmliche Suchsysteme an ihre Grenzen, da sie die Attribute der Datensätze nicht berücksichtigen. Darüber hinaus ist es aufgrund der großen Datenmengen nicht praktikabel, ganze Datensätze über das Netzwerk zu übertragen, um potenzielle Matches zu finden. In dieser Arbeit stellen wir eine Lookup-Infrastruktur für relationale Datensätze vor, die es Benutzern ermöglicht, mit einem Datensatz zu suchen und Ergebnisse von anderen Servern zu erhalten, die mit der Anfrage joinbar oder unionbar sind. Unser System nutzt verschiedene Maßnahmen, um kompakte Zusammenfassungen der Datensätze zu erstellen, die deutlich kleiner sind als die Originaldaten. Diese Zusammenfassungen werden dann verwendet, um Datensätze auf Ähnlichkeiten zu vergleichen. Durch empirische Evaluierung zeigen wir, dass das System effizient verwandte Datensätze findet und eine vergleichbare Reaktionszeit zu modernen Datensatz-Suchsystemen bietet, während der Netzwerkverkehr minimiert wird.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

In the modern digital world, large and diverse data are being constantly generated. To manage this influx, organizations often scale horizontally, distributing data across multiple servers. This distribution of datasets however makes it difficult to discover and search for datasets. Current data management solutions like CKAN or Socrata [3] primarily rely on keyword and metadata filters such as file type and creation date to perform dataset search, which are contingent on the quality and availability of those metadata. Furthermore, we are interested in searching data for augmentation purpose: given a relational dataset, we want to identify which other datasets out there can be joined or unioned with one or more attributes from our query dataset. With limited search capability of such traditional dataset search systems, this task is often challenging, as humans are required in the process to perform specific keywords, and tediously sift through datasets to determine if there are any matching attribute in it. Therefore, a faster and automated approach is needed.

To illustrate the problem, consider figure 1.1, which depicts a cluster with three nodes, each containing a number of datasets. Node N1 possesses dataset A, and wants to find out which datasets from node N2 and N3 can be joined with A. A naive approach would involve sending dataset A to nodes N2 and N3, requesting them to perform joins with each of their respective datasets. Firstly, this would incur a lot of traffic in the network, since dataset size can range anywhere from megabytes to gigabytes. Moreover, the feasibility of this method diminishes with an increasing number of nodes within the cluster. Secondly, once the dataset arrives in N2 and N3, how can they manage to perform join between A and all other datasets they have? And how do they decide that a join between an attribute in A and an attribute in another dataset is a reasonable join? (figure 1.2).

The goal of our work is to find a way to answer these questions. During research, we discovered that the task of searching for related datasets bears a lot of resemblance with the task of schema matching: finding semantic mapping between two schemas. Therefore, we apply schema matching techniques in our solution to help efficiently discover joinable and unionable attributes. Then, to accelerate the schema matching process and minimize network traffic, we employ a modern implementation of MinHash sketching and locality sensitive hashing (LSH) indexing.

The structure of the thesis is as following: chapter 2 supplies background knowledge on the topics of schema matching, its techniques, MinHash sketch, locality sensitive hashing, and REST API. Chapter 3 introduces other works that are related to or is the base of our implementation. Chapter 4 presents an overview of the architecture of our system and the schema matching techniques used. In chapter 5, we provide a detailed implementation of the system and an algorithm to optimize the search speed. Chapter 6 carries out an empirical evaluation of

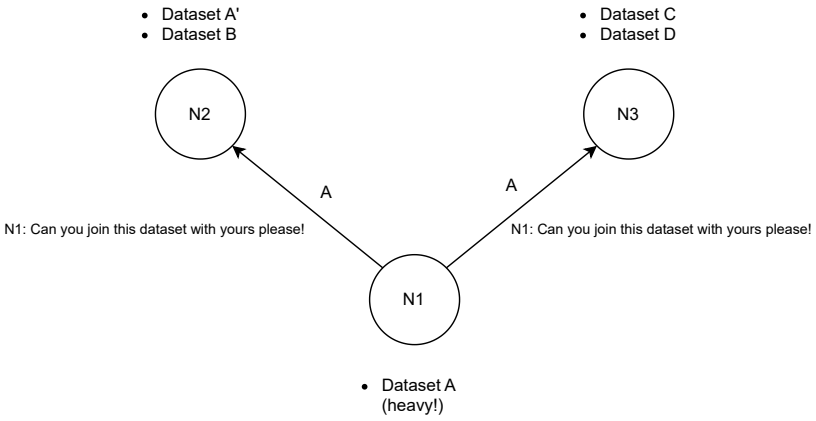


Figure 1.1: A cluster of three nodes, N1 is attempting to send dataset A over the network

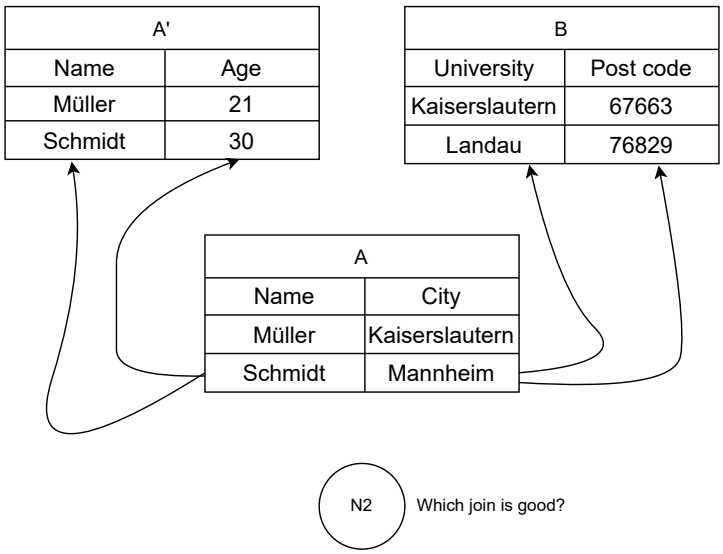


Figure 1.2: Caption

the system's performance. Finally, the thesis ends with the conclusion, limitations and future work.

Chapter 2

Background Knowledge

In this chapter, we give context on the terminologies, definitions and algorithms employed in our system.

2.1 Schema Matching

Schema matching is the task of finding semantic correspondence between two schemas, where a schema is defined as a set of elements connected by some structure [18]. It plays a critical role in data integration and translation applications. An use case example is the integration of a real estate schema and a property tax schema, where each schema may have structural and terminological differences, but occur in the same real-world domain and thus can be unified under the same schema. In the case of relational data, schema matching techniques can be used to determine the joinability and unionability between attributes of two tables [10]. Schema matching techniques for relational data work on attributes/columns of tables and can be roughly classified into two categories: schema-based and instance-based:

Schema-based These types of matchers only consider the schema-level information of a dataset such as table name, attribute names, description, data type, constraints, etc [13]. The similarity of the information can be calculated by e.g. how much the attribute name syntactically or semantically overlap, is the data type similar (short and int or text and varchar).

Instance-based When schema-level information is limited, the instances of an attribute can provide correct interpretation of schema information. Examples include the similarity in how the instances of two attributes overlap, or how the format patterns look like (money-related instances contain a currency symbol, whereas a zip code does not).

Matching process usually involves calculating the degree of similarity by a normalized numeric value in the range 0 to 1. It is also possible to combine different matching techniques together, then an algorithm is applied to aggregate the score of different techniques to one single score that is used to rank the results [5]. In our implementation, we will follow this hybrid strategy.

2.2 Similarity Metrics

2.2.1 Levenshtein distance similarity

In order to capture the syntactic relatedness between two words s and t, we use the Levenshtein edit distance. The Levenshtein distance between two words is the minimum required number of insertions, deletions or substitutions of a character to transform one word into another [12]. Since this metric measures

distance, we convert it into a similarity metric by dividing the distance with the length of the longest word, normalizing it to the range 0 and 1, and then subtract it from 1 to get a similarity score:

$$sim_{levenshtein} = 1 - \frac{dist(s,t)}{max(s,t)}$$

2.2.2 WordNet Similarity

In order to capture the semantic relatedness between two words, we use the machine-readable lexical database WordNet [15], which contains English words grouped into sets of synonyms (synsets) and structured in a hierarchical is-a relation. One algorithm that can be used to calculate the similarity of two words s and t is the Wu-Palmer algorithm [21], which considers the depths of two synsets in the Wordnet hierarchy as well as the depth of the least common subsumer (LCS) i.e. the most specific concept that is the ancestor of both s and t:

$$sim_{wordnet}(s,t) = 2 \cdot \frac{depth(LCS)}{(depth(s) + depth(t))}$$

To extend the similarity of two words to two sets of words / sentences q_i and q_j, we use an equation similar to ones from [14] and [13]:

$$score(q_i, q_j) = \frac{\Sigma_{s \in q_i} sim_m(s, q_j) + \Sigma_{s \in q_j} sim_m(s, q_i)}{|q_i| + |q_j|}$$

where $sim_m(s, q_j$ is the similarity score of s and the word in q_j that has the highest similarity score. If a word does not exist in the dictionary, we fall back to Levenshtein distance similarity for the calculation.

2.2.3 Jaccard Similarity

The Jaccard similarity/coefficient is a metric used to calculate the similarity of two sets [11]. Given two sets A and B, the Jaccard coefficient $J(A,B)$ of the two sets is defined by the ratio between the size of the intersection to the size of the union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

For example, if $A = \{1,2,3\}$ and $B = \{2,3,5\}$, then $J(A,B) = \frac{1}{2}$. The value of Jaccard coefficient ranges from 0 to 1. In [16], the authors show that a high Jaccard coefficient value between sets suggests a high probability that the elements of the sets are drawn from the same domain. Therefore, we can utilize this measure to find unionable columns for a given column of a dataset.

2.2.4 Set Containment

One problem with Jaccard coefficient is that the measure is symmetric. In the case where sets are of vastly different sizes, the measure is biased to sets with smaller sizes [22]. For example, consider the query set:

- $Query$ = {Ontario, Toronto}

and the two sets:

- $Provinces$ = {Alberta, Ontario, Manitoba}
- $Locations$ = {Illinois, Chicago, New York City, New York, Nova Scotia, Halifax, California, San Francisco, Seattle, Washington, Ontario, Toronto}

We see that while the $Locations$ set contain all the elements of the $Query$ set, their Jaccard similarity is only 0.083, while the similarity between $Query$ and $Provinces$ is at 0.25, indicating a higher degree of relatedness despite $Provinces$ containing only one similar element with $Query$. Due to this reason, there is another asymmetric measure used in the literature for comparing set similarity called set containment.

Given two sets A and B, the set containment of A in B is defined as:

$$J_C(A, B) = \frac{|A \cap B|}{|A|}$$

and vice versa for the containment of B in A:

$$J_C(B, A) = \frac{|A \cap B|}{|B|}$$

Using set containment, we see that $J_C(Q, Provinces)$ is 0.5, while $J_C(Q, Locations)$ is now 1.0. This measure is proven to be useful in identifying primary-key/foreign-key relationships, which enables the process of finding joinable datasets [7].

2.3 Q-gram

Another way to measure the syntactic relatedness of two words beside using Levenshtein distance similarity, is by comparing their q-grams sets. A q-gram (or n-gram) of a string is simply a substring of length q [20]. The set of q-grams of a string is then constructed by finding all q-grams within that string. We utilize the Jaccard coefficient mentioned above to calculate the similarity of the q-grams sets. For example, to compare "house" and "horse" with a q value of 2:

- 2-grams of "house": $s = \{ho, ou, us, se\}$
- 2-grams of "horse": $t = \{ho, or, rs, se\}$, therefore:

$$sim_{qgram}(house, horse) = J(s, t) = \frac{1}{3}$$

The advantage of this measure compared to Levenshtein distance similarity is, they can be efficiently calculated using locality sensitive hashing, which is explained in section 2.5.

2.4 MinHash Sketch

When a set size goes to the degree of thousands of elements or higher, calculating the Jaccard coefficient/containment of two sets can become computationally very expensive. Instead, we transform the sets into smaller, fixed-size representations called "sketches" and calculate the similarity measures using only these sketches [11]. One of these types of sketches is called MinHash.

The core idea behind MinHash is to apply a hash function to the elements of a set, simulating a random permutation, and then store the minimum hash value observed. By doing this multiple times with different hash functions, we create a set of minimum values that serves as a compact representation of the original set. The interesting property of MinHash is that, the probability of 2 sets having the same minimum hash value when applied the same hash function is directly related to the Jaccard coefficient between two sets:

$$Pr[h_{min}(A) = h_{min}(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$$

Thus, by using k multiple hash functions, the Jaccard coefficient can be estimated as:

$$J(A, B) = \frac{1}{k} \cdot \Sigma\ \mathbf{1}((h^i_{min}(A) = h^i_{min}(B))$$

Where the function **1** returns 1 if the condition inside the argument is true and 0 otherwise.

2.5 Locality Sensitive Hashing

While MinHash allows us to efficiently calculate similarity of sets, it remains a challenge to find the pairs with greatest similarity among large collection of sets, since we would need to perform up to pairwise calculations of all pairs of sets to get the correct result. However, for our task we would want only the most similar pairs or pairs that are above some similarity threshold. Therefore, we accept a slight loss in accuracy and use approximate methods to focus our attention only on pairs that are likely to be similar, without investigating every pair. One such method is locality sensitive hashing (LSH), which is widely used in data discovery tasks due to its exploratory and imprecise nature. [7, 11]

The general idea of LSH is to use a locality sensitive hash function to hash items into buckets, where the amount of buckets is significantly smaller than the universe of the possible input items. Locality sensitive hash functions are functions which map inputs that are similar to the same hash value with high probability, while minimizing that probability for inputs that are dissimilar [4].

If we use these hash functions to map elements in the buckets, similar items would fall into the same buckets with high probability. Thus, in order to find similar items with our input, we need only to check the similarity of the items in the bucket that our input falls into.

In literature, the way a MinHash sketch in hashed into a LSH index is by first diving it into b bands of equal length, then hash each of these bands into individual bucket array. Two sketches are similar if in any band, there is a bucket collision (figure 2.1). The use of LSH functions mean the more similar two sketches are, the higher the probability they will be identical in at least one band, making them candidate pairs. This banding strategy thus enhances the likelihood of detecting similar columns while minimizing accidental collisions for dissimilar pairs.

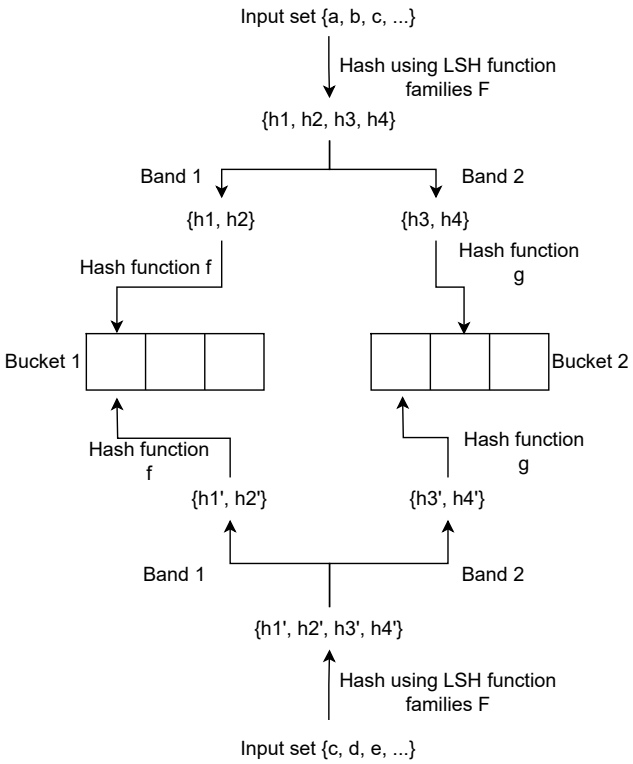


Figure 2.1: Example of the LSH banding technique. The two sketches collide in bucket 1, hence a match

2.6 REST API

Representational State Transfer (REST) is a term coined by Fielder in his PhD dissertation [8]. It is a software architectural style that defines a set of con-

straints on how the components of a distributed hypermedia system like in Web should communicate with each other. Rest API is an application programming interface (API) that adheres to the constraints of REST architectural style. While there are six constraints that Fielder introduced in his work, the three most important constraints for our work are:

- **Uniform interface**: REST components communicate with each other via representation or abstraction of resources but still need to contain enough information to manipulate the underlying resources. They are identifiable using Unique Resource Identifiers (URIs).
- **Client-server decoupling**: Client and server are independent of each other. Communications between client and server happen between an interface, i.e. the URI of a resource.
- **Statelessness**: Each request from client to server must contain all the information required to process the request, without utilizing additional server context.

Overall, these constraints allow a system to have better performance thanks to its flexibility and scalability. In practice, REST APIs work through HTTP connections, which fit the constraints of REST perfectly. A REST API request contains the following component:

- **Endpoint**: The URI where the requested resource can be retrieved. For instance, the URI for a HTTP request is an Unified Resource Locator (URL): `https://song.com/trackID/1`
- **Method**: The HTTP method that is used to operate on the resources. The common methods are GET (retrieve a resource), POST (Create a new resource), PUT (update existing resource) and DELETE (remove a resource).
- **Headers**: Provide metadata-level information about the request like the content type or its encoding.
- **Parameters**: Additional information that is appended to the URL to specify server's action in a more detailed way.
- **Request body**: The data, which is a representation of resource that is sent to the server. It is often formatted in JSON.

Upon receiving a request, the server responds with a HTTP response, containing the response metadata, a status code to indicate whether the request was successful or not and the resource representation itself in the body. Due to the nature of our system working in a distributed environment, REST APIs are employed to facilitate communication between the components.

Chapter 3

Related Work

3.1 Valentine

In Valentine [10], Koutras et al. presented an experiment suite designed to evaluate various schema matching methods for dataset discovery, as these tasks rely heavily on schema matching to find joinable and unionable datasets. The schema matching methods they evaluated were group into six categories:

- **Attribute overlap matcher**: Two columns are related if their attribute names have syntactic overlap above a predefined threshold.
- **Value overlap matcher**: Two columns are related when their value sets/instances have high degree of overlap.
- **Semantic overlap matcher**: Two columns are related when they have significant overlap in their labels or domain, according to some source of external knowledge.
- **Data type matcher**: Matches columns that have similar data types.
- **Distribution matcher**: Matches columns based on how similar the distributions of the values are.
- **Embeddings matcher**: Two columns are related if they are highly similar in the embeddings of the values, which are derived from an existing pre-trained model on natural language corpora.

The methods are then ranked across four relatedness scenario:

- **Joinable relations**: There exists at least one pair of attributes coming from each relation on which a join can be executed.
- **Semantically-joinable relations**: There exists at least one pair of attributes coming from each relation that is semantically (must not syntactically) related and on which a semantic join can be executed. Such joins are called "fuzzy".
- **Unionable relations**: Two relations have the same arity and there exists a 1-1 mapping, denoting semantic equivalence between their attribute sets.
- **View-unionable relations**: There exists a view of each relation such that the two views are unionable.

For each scenario, a match is defined as a pair of attributes from two different relations. The matches are ranked in descending order according to their similarity scores. The notions introduced in Valentine, along with the datasets they used, serve as a baseline for how we implement and evaluate our systems.

3.2 D3L

In the paper "Dataset Discovery in Data Lakes" by Bogatu et al., the authors introduced D^3L, a dataset discovery framework that utilizes LSH indexes to to efficiently determine the relatedness (joinability and unionability) between attributes of datasets in data lake. The similarity measures that are inserted into the indexes are:

- Attribute name: Calculate similarity based on the n-gram sets of the attribute name.
- Attribute values overlap: Calculate similarity based on how much the attributes' values overlap.
- Word-embedding: Calculate similarity based on semantic relatedness of the attribute' values by representing words as multi-dimensional vectors.
- Format representation: Calculate similarity based on the regular representation patterns of the attributes' values
- Domain distribution: For numerical attributes, calculate similarity based on how close the values are in their distribution.

The values of each individual measurements are aggregated into a 5-dimensional vector, which is then used to compute a combined distance score representing the overall similarity between the two tables. The combined distance score is calculated using a weighted Euclidean distance formula, which is trained using machine learning, taking into account the relative importance of each measure type. Our work is inspired by the use of LSH indexes for similarity calculation and the weighted sum used to calculate the final score. Similar to D^3L, we will also be using attribute name, attribute values overlap and the format representation as our measure of similarity. We choose not to use word-embedding similarity due to low precision and recall results evaluated in the paper as well as in [10]. The distribution similarity is also not used due to it not being indexable into LSH index and require the reading of all attribute instances, which is not suitable when we are working in a distributed environment.

3.3 Lazo

In [7] by Fernandez et al., the authors proposed Lazo, a method that efficiently estimate both Jaccard similarity and set containment of sets by redefining Jaccard similarity to include set cardinality, which allows for independent estimation of intersection and union of sets. At the beginning of the paper, they also mentioned some problems with traditional LSH index that makes it difficult to be directly applied to the data discovery problem:

- LSH cannot estimate set containment since that requires estimating the size of the intersection, which is difficult to do with LSH. On the other hand, LSH methods that do support set containment do not support incremental indexing, which means items can no longer be hashed to the index after being queried. This makes those methods unable to work in the

context of dataset discovery/schema matching, where datasets are always changing or there are new data arriving.

- The indexes do not return a similarity score. As mentioned in 2.5, the LSH index only return candidates that are hashed into the same buckets. In order to obtain a score, we would either have to manually perform the calculation, or use multiple indexes with different similarity threshold and query them all, both of which are costly and inefficient.

As a result, they presented Lazo index, a modified LSH index that allows query using arbitrary input thresholds and returns a set of candidates along with their corresponding Jaccard similarity and set containment. Evaluation of their approach shows that the Lazo index significantly enhances indexing speed and provides 2-10x faster query speed compared to traditional LSH indexes. Additionally, their method outperforms other LSH methods that support set containment estimation, providing higher estimation quality. Testing on large datasets also showed that the overhead of applying the Lazo method is negligible in practice, making it suitable for data discovery scenarios. In fact, Lazo has already been used in some recent novel dataset search systems [2, 6]. In our implementation, we will also be using Lazo and Lazo index to efficiently query similarities between datasets.

Chapter 4

System Overview

This chapter presents the measures of similarity used in our system as well as giving a high-level introduction to the system's architecture, how it is set up in a distributed environment, how user can use it to query for similar datasets.

4.1 Measures of similarity

As mentioned in section 2.1, the task of similarity search is comparable to the task of schema matching, therefore, we also use schema matching techniques to measure the relatedness between two datasets. The system follows a hybrid schema matching approach, as both the schema and the instances are utilized.

4.1.1 Schema similarity

First, the *data type* of the attributes are used as a hard measurement for how similar the two attributes are, meaning if an attribute does not have a similar type to a query attribute, they are inherently dissimilar. Therefore, this attribute will be excluded from the candidate set, preventing further calculations with it.

Second, the *table name* is considered in the similarity calculation. We provide two options on how a similar score between table names can be calculated:

- **Q-gram sets similarity**: The syntactic similarity of two table names is calculated by measuring the Jaccard coefficient between their q-gram sets. By comparing their q-grams, the table names can be fuzzily matched, so that a table whose name is a typo of the query table's name can still be recognized as a candidate.
- **WordNet similarity**: Measures the similarity of the table names according to their semantics. The equation to the similarity calculation is mentioned in section 2.2.2. This method requires however pair-wise calculations, since the semantic information of a word cannot be summarized into a sketch.

The last schema-level information taken into consideration is the *attribute/column name*. The same methods of calculation for table name are also applied for this case.

4.1.2 Instance similarity

The two types of evidence inspected at the instance level are *value sets overlap* and *format overlap*. To calculate the similarity score, first we create a set from all the values of a column (i.e. retrieving a collection of unique values from the column).

For value sets overlap, the unique values sets are transformed into a MinHash sketch, on which the similarity calculation are performed. To avoid pair-wise comparison of all sketches, all sketches are added to a Lazo LSH index (section 3.3.

Format overlap, inspired by [1], relies on the format of the values to determine relatedness. The format of a value is a regular, predictable structure of the value string's representation, describable through a regular expression (regex). Using this approach, we are able to differentiate between a column of zip code (format: numbers) and a column of email (format: characters, followed by a special symbol "@" then characters, a dot and some more characters), for example. The details of how the regular expression are created are discussed in chapter 5. From the unique values set, we create a set of all regular expressions that can describe all values of a column. The regex sets for all columns are also transformed into MinHash sketches, similar to the unique values sets above, and inserted into their own LSH index.

4.2 System Architecture

The system consists of two main components: the client application and the server application. The client application should be run on machines where the query function is desired, while the server application is set up on one single machine, where it communicates with the database server and the clients can send their query requests to in order to be processed (figure 4.1).

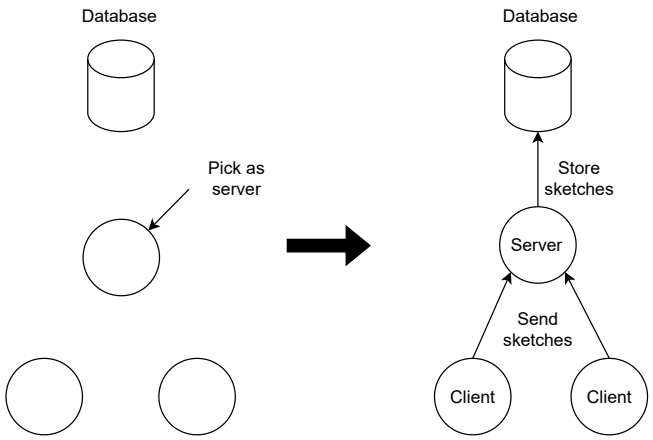


Figure 4.1: Architecture overview

The rationale behind this set up is as follows: The goal of our system is to minimize the amount of data sent across the network, therefore the datasets should be locally processed to create a summary which is smaller in size and thus transferable over the internet. At this point, a server is not needed, since a machine can simply send the summaries to the database server to store and query the database to find similar datasets. However, as already discussed in

section 2.5, pair-wise similarity calculations are costly, hence we use LSH for similarity search, specifically the Lazo implementation proposed by Bogatu et al. [7]. Unfortunately, one drawback of Lazo is that the indexes are only capable of running in main memory, which we discuss in chapter 7. As a result, there needs to exist one central machine to store the LSH indexes. An alternative would be for all machines to store indexes, which would require all machines to retrieve new summary and update their indexes whenever new summaries enter the database, incurring unnecessary network traffic.

4.3 Usage

The system starts by letting an user from a machine boot up the server application, providing the application with the information of the host and port of the database server. Then, users from client machines boot up their respective client applications, providing the application on startup with the base IP address and port where the server can be reached. Were the server application to be moved to another machine, users from client-side boot up the application again with the new IP address and port. Once the server and client are both running, user can access a graphical interface from the web browser. From there, user can choose between two primary actions: upload and query.

- **Upload action**: Submit datasets for the client application to process, create summaries and send to the server via REST request to store. In the interface, user can either choose to upload a single file, by providing the path from the file system to that exact file, or user can upload an entire folder of datasets by specifying the path to that folder (figure 4.2). Depending on the result of the action, the UI displays a success or error message.
- **Query action**: Submit a dataset for the client application to process, create summaries and send to the server via REST request to query. In the interface, user provide the path to the file to be queried, the limit on how many matches to receive, the threshold at which a candidate is considered a match and choose between the two query modes: join and union (figure 4.3). If the action failed, an error will be displayed on the form interface, meanwhile a successful action will redirect the user to the result interface, where all matches are displayed in descending order of similarity score (4.4).

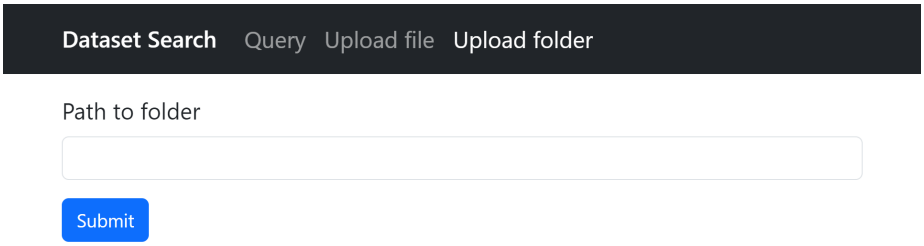


Figure 4.2: Upload interface

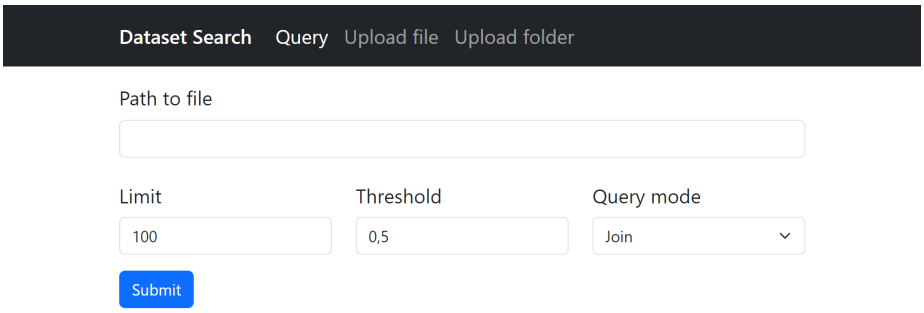
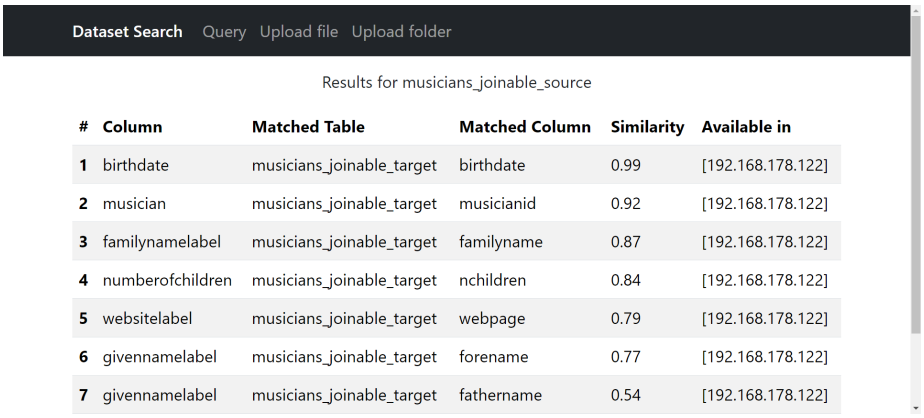


Figure 4.3: Query interface



#	Column	Matched Table	Matched Column	Similarity	Available in
1	birthdate	musicians_joinable_target	birthdate	0.99	[192.168.178.122]
2	musician	musicians_joinable_target	musicianid	0.92	[192.168.178.122]
3	familynamelabel	musicians_joinable_target	familyname	0.87	[192.168.178.122]
4	numberofchildren	musicians_joinable_target	nchildren	0.84	[192.168.178.122]
5	websitelabel	musicians_joinable_target	webpage	0.79	[192.168.178.122]
6	givennamelabel	musicians_joinable_target	forename	0.77	[192.168.178.122]
7	givennamelabel	musicians_joinable_target	fathername	0.54	[192.168.178.122]

Figure 4.4: Result interface

Chapter 5

Implementation

In this chapter, a detailed view of the system is presented. First, the objects used to communicate between server and clients are introduced. Then, a brief introduction to MongoDB, our database of choice is given. After that, we provide the implementation of the client application and lastly the server's implementation. We make the following assumptions about the datasets we are working with: they are relational tables in CSV format and the first row of the files contain the headers/attribute names of the tables. The system is implemented on Java using the Spring framework.

5.1 Communication Objects

There are two main types of objects that are exchanged between client and server: dataset summaries and query results. The summaries are created from the client side and sent to the server using POST requests either to be stored or to be queried. The query results object is the response that the server returns whenever the query endpoint is accessed.

5.1.1 Dataset Summaries

A table is summarized on a column-by-column basis. The summary of a column contain two main component: the metadata and the sketches:

- **Metadata**: Provides basic information about a column of a table (listing 5.1): the name of the table that this column belongs to, the name of this column, its datatype, the size/amount of elements the column contains, and a set of IP-Addresses of machines that carry this dataset. An additional id field that contains an UUID representing the table name, concatenated with the column name is associated with each metadata to uniquely identify the metadata and provide easy retrieval of all metadata of a dataset, which is discussed more in detail in the upcoming section.

```
1  class Metadata {
2      // id = uuid + SEPARATOR + column_name
3      // uuid is the same for all metadata of a dataset
4      String id;
5      String table_name;
6      String column_name;
7      String type;
8      int size;
9      int arity;
10     Set<String> addresses;
11 }
```

Listing 5.1: The metadata object

- **Sketches**: A set of Lazo's implementation of MinHash sketches, represented by the stored hash values and the cardinality of the set that is sketched (listing 5.2), which the authors used to improve estimation accuracy. The sketches are generated from the sets that were mentioned in section 4.1.

```
class Sketches {
    // id refers to the Metadata object
    private String id;
    private Set<Sketch> sketches;
}

class Sketch {
    SketchType type;
    long cardinality;
    long[] hash_values;
}

enum SketchType {
    TABLE_NAME, COLUMN_NAME, COLUMN_VALUE, FORMAT
}
```

Listing 5.2: The sketches object

5.1.2 Query Results

This object contains a list of individual entries of result, where a result maps one metadata from the query table to one matched metadata along with the similarity score of this match (listing 5.3). Results can be added into this object by providing the Metadata objects of query, matched candidate and the corresponding score. Additionally, the class provides different methods to sort the results in descending order of score, limit the results to only the top-k highest scoring ones, and filter out results whose scores are lower than the given threshold. The methods match with the options provided to the user in the query interface, so these method need to be correspondingly applied before sending the query results back to the user.

```
record QueryResults(List<SingleResult> results) {
    QueryResults sortResults();
    QueryResults limitResults(int limit);
    QueryResults withThreshold(double threshold);
}

record SingleResult(Metadata query, Metadata candidate, double
    score) {}
```

Listing 5.3: QueryResults class and the various methods to sort, filter, and limit matches

5.2 Database

In order to store the metadata and sketches, MongoDB is used. This is the database of choice due to several reasons:

- Replication: MongoDB is a distributed database, where the data are stored on multiple database servers. This redundancy increases data availability and provides fault tolerance in case a single database server fails.
- Flexible schema: MongoDB is a NoSQL database, therefore it does not store data in relational manner, but rather stored in JSON-style documents, which do not have constraint on the amount of attributes and their types. This allows us to make changes to the schema in case e.g. a new sketch type for certain types of attributes is developed without having to perform alterations to existing data to fit the new schema.
- Object mapping: The format MongoDB stores data in allow them to be easily converted to a Java object without the use of Object-Relational-Mappers (like OpenJPA [1])
- Fast read: MongoDB uses index to speed up data retrieval on indexed fields, especially on id field. Since the system mostly query on id field, and the sketches are need to be read into main memory on server startup (due to Lazo's LSH index), the database provides the needed high performance to the system.
- Unique index: Additionally, MongoDB also provides unique indexes, which ensure that the indexed fields do not store duplicate values. We use this feature to detect duplicate metadata. This happens when two machines contain the same dataset and both are trying to upload the summaries to the server. When duplicates are detected, they are grouped into the same metadata, with the set of addresses merged together.

In our database, there are two collections, one for the metadata and one for the for the sketches. The collections consist of documents, which are simply JSON strings in binary format mapped from the respective Java objects. An unique index is also set up on the fields table_name, column_name, type, size and arity of the metadata collection for reasons we mention in section 5.3.2 and 5.4.1.

5.3 Client Application

In this section, we describe the two main components of the application on the client side, which are the controller and the profiler.

5.3.1 Controller

The main job of the controller (listing 5.4) is to handle communications between user, client and server. It displays the forms to the user according to which action they want to take, retrieves the CSV files the user provide from the form, send the acquired table to the profiler to process, send the summaries to the server and displays back to the user the response that the server returned. The reading of CSV files are performed using Tablesaw [2], a third party library that provides column-wise reading and automatic type detection of attribute. The

[1] https://openjpa.apache.org/
[2] https://github.com/jtablesaw/tablesaw

controller performs REST requests to the server via the WebClient class, which is a class provided by Spring Reactive Web Framework to create HTTP requests. Upon application startup, a WebClient builder takes the IP address and port of the server that the user provided from the command line argument and create an instance of the WebClient with the server's URL as base URL.

```
@Controller
class AppController {
    // Perform REST requests
    WebClient client;
    Profiler profiler;

    @GetMapping("/upload_file")
    public String uploadFile(...) {
        // Display form to upload a single dataset
    }

    @PostMapping("/upload_file")
    public String saveFile(Form form, ...) {
        // Read file and summarize
        // Send summaries via POST request to /save endpoint
    }

    @PostMapping("/query")
    public String sendQuery(QueryForm form, ...) {
        // Read file and summarize
        // Send summaries via POST request to /query endpoint
        // Display QueryResults
    }

}
```

Listing 5.4: Preview of the controller class

5.3.2 Profiler

The profiler generates the metadata of a table by iterating over its columns to get the information in section 5.1.1. In the current implementation, the IP-Address is the local address of the machine the application is running on, which allows distinction between machines in a local area network. While table name, column name and data type are used for similarity calculation, the column size and table's arity provide additional distinguishing information to detect duplicate datasets entering the database.

As for sketches, the profiler uses three main methods: createNameSketch(), createColumnSketch() and createFormatSketch().

- createNameSketch(): used to create MinHash sketch for the table name and column name from their q-gram-sets (algorithm 1). The q-value of 4 is used, similar to [1]. If a string has a length smaller than 4, the input of the sketch generator would be an empty set.
- createColumnSketch() (algorithm 2): the method creates MinHash sketches by first converting a column to its set representation, retrieving only unique values. If a column is of type String, the values are not sketched as is, but rather transformed into a set of q-grams using the qGram() method above, which help to find fuzzy matches.

Algorithm 1 Generating the q-gram set of a string with q = 4

Input: String s
Output: Set of q-grams Q
$q \leftarrow 4$
$Q \leftarrow$ Set()
for i = 0; i < s.length - q + 1; i++ **do**
  $Q.add(substring(i, i + q))$
end for
return Q

Algorithm 2 Generating the sketch of a column's values

Input: Column C
Output: Sketch S
$S \leftarrow$ Sketch()
$C_set \leftarrow C.toSet()$
if C is of type string **then**
  for each value in C_set **do**
    $S.update(qGram(value))$
  end for
else
  $S.update(qGram(C_set))$
end if
return S

- createFormatSketch(): the method creates a sketch on the set of regexs that describe all possible values of the column. The regex grammar uses the set of patterns in listing 5.5 as the alphabet, which is the extended patterns set from [1] as we found their patterns are not listed in the correct order and are not discriminating enough. For each value represented as a string, algorithm 3 continually tokenizes the string from the beginning until the end, finding the longest starting substring to convert to a character in the alphabet. If a value matches more than one pattern, the first match from the listing is chosen. And if a character in the alphabet appears consecutively, all occurrences except the first are replaced by "+". For example, a column with the set of values $\{Alice\ C., bob, cat, user1??\}$ would yield the regex set $\{cwup, l, ao+\}$.

```
// Matches the start of a string with alphanumeric characters
a = ^(?:[0-9]+[a-zA-Z]|[a-zA-Z]+[0-9])[a-zA-Z0-9]*
// Matches the start of a string with capitalized characters
c = ^[A-Z][a-z]+
// Matches the start of a string with uppercase characters
u = ^[A-Z]+
// Matches the start of a string with lowercase characters
l = ^[a-z]+
// Matches the start of a string with numbers
n = ^[0-9]+
// Matches the start of a string with punctuations
p = ^\\p{Punct}+
// Matches the start of a string with whitespaces
w = ^\\s+
// Matches the start of a string with a character that is not
```

```
    catched by other pattern
o = .
```

Listing 5.5: Alphabet of the regex

Algorithm 3 formatRegex() algorithm

Input: String s
Output: Regex string r describing s
$r \leftarrow$ ""
while !$s.isEmpty()$ **do**
  $char \leftarrow$ longest substring that matches a pattern in the alphabet
  $r \leftarrow r + c$
  $s \leftarrow s.substring(char.length())$
end while
$r \leftarrow$ consecutive occurrences of a character in r replaced by " + "
return r

The sketch generator is a supporting class that handles the process of reading the input set and transforming it to Lazo's implementation of MinHash sketches (LazoSketch object). As the LazoSketch only hashes input values of type string, we provide a method to convert the values to be sketched to their string representation (listing 4). There are also two special cases where the input set is empty or is of type boolean. In case the set is empty, we return a LazoSketch containing the hash values of the empty string. If the set is a boolean set, then the string representation of the boolean values are 0 and 1.

Algorithm 4 Updating a LazoSketch

Input: Iterable i, LazoSketch S
Output: Updated sketch S
if $i == null$ or $i.isEmpty$ **then**
  $S.update("")$
  return S
end if
for $value$ in i **do**
  if $value == null$ **then**
    $S.update("")$
  else if $value.isBoolean$ **then**
    if $value == true$ **then**
      $S.update("1")$
    else
      $S.update("0")$
    end if
  else
    $S.update(value.toString())$
  end if
end for
return S

5.4 Server Application

In this section, we describe the components of the server. First is the controller that provides the endpoint for the save action. Then we show how the sketches are stored in the LSH indexes and how the indexes are queried to retrieve similarity scores. Next, we go into details on the calculation of the WordNet similarity score. After that, we explain the way to combine all similarity scores from different measures to one single score. Finally, the query controller is presented with an optimization technique to avoid unnecessary calculations, ensuring that the server performs efficiently when there are many datasets present.

5.4.1 Save Controller

In this REST controller class, a save endpoint is available to receive dataset summaries from clients. The controller communicates with the database using the MongoRepository interface from Spring framework, which provides a convenient way of database interaction by removing the need to write boilerplate code. To perform a query on a database, we need only to extend the interface with a method name that contains specific query keywords. For example:

```
1 @Repository
2 interface MetadataRepo extends MongoRepository<...> {
3     List<Metadata> findByIdStartsWith(String regex);
4
5     @Query("{ 'table_name': ?0, 'column_name': ?1, 'type': ?2, '
          size': ?3, 'arity': ?4 }")
6     Optional<Metadata> findByUniqueIndex(String table_name, String
          column_name, String type, int size, int arity);
7 }
```

Listing 5.6: Example of the the repository interface for collection of metadata

In listing 5.6, we extend MongoRepository with method findByIdStartsWith() to query for metadata whose id starts with the same string given in the input. Instead of having to provide an implementation of the method, MongoRepository automatically recognizes the keywords:

- *find*: Look up in the database
- *ById*: Look for field "id" in the Metadata objects
- *StartsWith*: Adds to the lookup condition that only id that starts with the string in the input are returned.

and construct a query out of them. In the case where a query is too complex to be built using only keywords, we can also provide the query manually in the @Query annotation, like in the findByUniqueIndex() method where we look for a single metadata that matches the unique index constraint by listing the fields' names as key and placeholders as values, which are replaced by the method's arguments when the query is executed.

When the list of Summaries objects make it to the save endpoint on the server, for each summaries (a column of a table), the controller first attempt to insert the metadata to the database before inserting the sketches to the database and to the indexes. If a DuplicateKeyException is thrown, then we know that

the metadata already exists due to the unique index constraint. The existing metadata is retrieved instead and the IP address field of the metadata taken from the request is added to the existing address set in the database, indicating that the dataset is available from multiple sources (algorithm 5).

Algorithm 5 Saving the summaries into the database

```
1: Input: List of Summaries objects S
2: for summaries in S do
3:   try
4:     store summaries.metadata in database
5:   catch DuplicateKeyException
6:     metadata ← findByUniqueIndex(table_name, type, size, ...)
7:     metadata.addresses.addAll(summaries.metadata.addresses)
8:     continue
9:   store summaries.sketches in database
10:  update LSH indexes with summaries.sketches
11: end for
```

5.4.2 LSH index

Once the summaries are stored in the database, they are then also immediately loaded into the LSH indexes that lie the main memory. There are currently four indexes available to correspond to the four sketch types available as seen in listing 5.2. The insertion method (listing 6) works by first recreating the LazoSketch object from the cardinality value and the hash values from the Sketch object. Then the method checks the type field from the Sketch object to see in which index should this LazoSketch be inserted to. The Lazo index also requires an object to be inserted as a key to identify the results return upon querying. Since the sketch uniquely identifies a metadata/column of the table in each index, the id of the metadata is chosen to be the key.

Algorithm 6 Inserting a sketch into LSH index

```
1: Input: Sketch id id, Sketch S
2: lazoSketch ← S.recreateSketch()
3: switch S.type()
4:   case TABLE_NAME: updateTableNameIndex(id, lazoSketch)
5:   case COLUMN_NAME: updateColumnNameIndex(id, lazoSketch)
6:   case COLUMN_VALUE: updateColumnValueIndex(id, lazoSketch)
7:   case FORMAT: updateFormatIndex(id, lazoSketch)
```

Since the indexes are stored in memory, upon starting the application, all sketches also have to be loaded from the database. While this is inefficient, we argue using evaluation statistics that the footprint of the sketches are small enough to be read all at once and the server startup is not a common operation, therefore the loading process does not compromise the usability of the system.

Now, in order to query the indexes, the LazoIndex class provides a method query() that takes the query LazoSketch as input, as well as the threshold of similarity above which another LazoSketch is considered a candidate. The

method then returns a set of LazoCandidate, which is just a wrapper object for the key that correspond to the LazoSketch that was inserted and three scores: Jaccard coefficient, set containment of the query in candidate and the set containment in the opposite direction (listing 5.7).

```
class LazoIndex {
    Set<LazoCandidate> query(LazoSketch sketch, float threshold)
}

class LazoCandidate {
    Object key;
    // Jaccard coefficient
    float js;
    // Set containment of query in candidate
    float jcx;
    // Set containment of candidate in query
    float jcy;
}
```

Listing 5.7: Candidate object returned by the query() method

When we call the query() method, for each candidate returned, we check the key (id of corresponding metadata) of the candidate to make sure the candidate is not the from the same table as the query input. This is achievable thanks to the UUID present in the id that identifies unique a table. Therefore, in order to check whether two metadata/columns come from the same table or not, we only need to check the UUID prefix of the metadata. Then, we retrieve the candidate metadata from the key of the LazoCandidate object and wrap the three scores (js, jcx and jcy) into a separate object. All the metadata are finally mapped to their corresponding scores for further query processing. The algorithm is shown in 7.

Algorithm 7 Querying the LSH index

Input: Index to be queried I, LazoSketch s, sketch identifier id, threshold t
Output: Map C of metadata candidates to its score
$C \leftarrow Map()$
for *candidate* in $index.query(s, t)$ **do**
  $candidate\_id = candidate.key$
  if $candidate\_id$ belongs to same table as id **then**
    continue
  end if
  $candidate\_metadata \leftarrow findById(candidate\_id)$
  $score \leftarrow newScore(candidate.js, candidate.jcx, candidate.jcy)$
  $C.put(candidate\_metadata, score)$
end for
return C

5.4.3 Wordnet Similarity Calculator

As mentioned before in section 4.1, while syntactic similarity can be calculated from the LSH indexes, semantic similarity calculation is achieved using WordNet database, and more specifically with the Wu-Palmer algorithm. For this, we

use a third party library [3] that provides an implementation of the Wu-Palmer algorithm as a base to compute our metric.

One problem with using WordNet is, however, the database only contains words in its base form: i.e. singular nouns, verbs that are not conjugated. As a result, performing the Wu-Palmer algorithm on any words that are not in their base form will yield a similar score of 0. Therefore, a way is needed to first transform a word back to its base form before calculation. For this task, we employ Stanford's natural language processing (NLP) library [4], which contains lemmatization algorithm. This is different from word stemming algorithms like Porter stemmer [5] since stemming may not actually return a real English word, which is necessary requirement for our application.

We then proceed to extend the semantic similarity between words to the entire table and attribute name, which can be multiple words long. Firstly, the string representation of the name is tokenized into a list of individual string by splitting it around whitespaces or underscores, which are common word separator in names (listing 5.8).

```
1 List<String> tokenize(String s) {
2     \\ regex denotes consecutive occurences of whitespaces or
          underscores
3     return Arrays.asList(s.split("\\s+|_+"));
4 }
```

Listing 5.8: tokenize() method

Then, the tokenized words are wrapped in a sentence class from NLP's library so that the words can be lemmatized. After that, the stop words (stored in a predefined list) are removed from the word list as they do not contribute to the semantic relatedness of the name. We do not remove the stop words before lemmatization since NLP library uses context of other words to determine the part of speech for correct word transformation. Finally, the algorithm from section 2.2.2 is applied. The entire process is describe in algorithm 8:

5.4.4 Measure Object

In order to differentiate the similarity scores of different similarity measurements, we wrap the score into a Measure object that contains the score, the type of measure this score represents, and the weighting of this score (listing 5.9), which is taken into consideration when the scores are aggregated together.

```
1 record Measure(MeasureType measures, double score, int weight) {}
2
3 enum MeasureType {
4     COLUMN_VALUE, COLUMN_FORMAT, COLUMN_NAME_QGRAM,
          TABLE_NAME_QGRAM, COLUMN_NAME_WORDNET, TABLE_NAME_WORDNET
5 }
```

Listing 5.9: The measure object

[3] https://github.com/dmeoli/WS4J
[4] https://stanfordnlp.github.io/CoreNLP/
[5] https://snowballstem.org/algorithms/porter/stemmer.html

Algorithm 8 WordNet similarity of table/attribute names

1: **Input**: Strings $name1$ and $name2$
2: **Output**: Similarity score $name\_sim$
3: $sentence1 \leftarrow tokenize(name1)$
4: $sentence2 \leftarrow tokenize(name2)$
5: $lemma1 \leftarrow sentence1.lemmas()$
6: $lemma2 \leftarrow sentence2.lemmas()$
7: $removeStopWords(lemma1)$
8: $removeStopWords(lemma2)$
9: $name\_sim = 0$
10: **for** $word1$ in $lemma1$ **do**
11: $word\_sim\_max = 0$
12: **for** $word2$ in $lemma2$ **do**
13: $word\_sim = wuPalmer(word1, word2)$
14: **if** $word\_sim == 0$ **then**
15: $word\_sim = levenshteinSimilarity(word1, word2)$
16: $word\_sim\_max = max(word\_sim\_max, word\_sim)$
17: **end if**
18: **end for**
19: $name\_sim+ = word\_sim\_max$
20: **end for**
21: **for** $word2$ in $lemma2$ **do**
22: $word\_sim\_max = 0$
23: **for** $word1$ in $lemma1$ **do**
24: Repeat steps 13 - 17
25: **end for**
26: $name\_sim+ = word\_sim\_max$
27: **end for**
28: $name\_sim = name\_sim$ / $(lemma1.size() + lemma2.size())$
29: **return** $name\_sim$

The score's weight of a measure depends on what type of measure it is and whether the query mode is join or union. The intuition behind the weighting system is that some types of similarity measures may play a more significant role in telling the degree of relatedness than others. For example, in the TPC-H [6] schema, the table Orders and Customer are joinable with each other through the attribute custkey. However, it is apparent that the two words orders and customers are not that related to each other. Therefore, it is sensible to not weight the similarity of the table names as high as other measures during aggregation, but rather focus more weight on the column values overlap to detect the foreign key constraint. On the other hand, imagine the Orders table belong to a company, where every year the company creates a new Orders table to keep track of orders for only that year. Then they may name each Orders table Orders_2020, Orders_2021, ... etc. Now, we want find all the Orders table to union them to one single table, then the table name would be of much more importance than column values overlap, since the order records of each year does not necessarily have any overlap between each other. For this reason, the weighting of the measures also differ in case of join or union.

For each pair of query - candidate column, to get all of the measures calculated between them, we wrap them again in the Measures object (listing 5.10):

```
1  class Measures {
2      List<Measure> measures;
3
4      void addMeasure(Measure measure) {
5          // Throws exception if a measure already exists in the list
6      }
7
8      double weightedAverage() {}
9  }
```

Listing 5.10: The Measures object

Given the list of measures, this class aggregates the scores using weighted average function:

$$score(query, candidate) = \frac{\sum_{m \in measures} w_m \cdot sim_m(query, candidate)}{\sum_{m \in measures} w_m}$$

As all calculated scores are in the range 0 to 1, the aggregated score also ranges from 0 to 1.

5.4.5 Query Controller

In main endpoint provided in this REST controller class is the query endpoint, where it receives the summaries of the query dataset along with three additional query parameters: query mode (union or join), limit (number of matches returned) and threshold at which an aggregated score is considered a match. The algorithm (9) works as follows: in order for the query dataset to be discoverable by the LSH index, we first attempt to insert the summaries into the

[6] https://www.tpc.org/tpch/

appropriate collections inside the database as well as inserting the sketches to the indexes (line 4-11). Here, it can happen that the dataset that the user is trying to query already existed in the database, so a DuplicateKeyException will be thrown. The insert attempt is stopped and the summaries from the database are used instead. To retrieve the summaries from the database, the unique index is used (line 15-18)). Notice we cannot lookup a dataset using the query table's UUID, as they are generated differently from the one in the database. After having the database and LSH indexes set up for the query, we call the queryColumn() method to find candidates for the individual columns of the table, as all of our similarity measures work on column level. The list of all candidates from all columns are then stored in a QueryResults object and depending on the query parameters, candidates with scores below threshold are removed and/or only top-k candidates are retained. The second to last step is to removed the query's summaries that might have been added to the database and LSH indexes from the beginning. Finally, the QueryResults object is then sent back to the client with the scores sorted from highest to lowest.

Algorithm 9 queryTable() algorithm

```
1:  Input: Summaries list S, query mode m, limit l, threshold t
2:  Output: Query result R
3:  existed ← false
4:  for summaries in S do
5:      try insert summaries.metadata and summaries.sketches to database
6:      catch DuplicateKeyException do
7:          existed ← true
8:          break
9:      end catch
10:     insert summaries.sketches to LSH indexes
11: end for
12: if !existed then
13:     columns ← all metadata from S
14: else
15:     columns ← Array()
16:     for metadata_request in all metadata from S do
17:         metadata_db ← findByUniqueIndex(indexed fields of metadata_request)
18:         columns.add(metadata_db)
19:     end for
20: end if
21: R ← QueryResult()
22: for column in columns do
23:     R.addAll(queryColumn(column, m))
24: end for
25: if !existed then
26:     remove query's summaries from database and LSH indexes
27: end if
28: sort R
29: return R.withThreshold(t)
```

On the column level, we differentiate between two main categories of similarity measures to perform the calculations: WordNet similarity (on table names and column names) and LSH-based similarity (on table names, column names, value sets overlap and format overlap) (listing 5.11). In any combinations of measures used, the data types is always the first filter to be used as per section 4.1.1. Since the data types of the attribute is derived automatically from CSV file by Tablesaw, and without an official schema provided, the type of an attribute is ambiguous anyway. Therefore, we define classes of data types defined by Tablesaw that are similar to each other, so that two columns with different types but belong in the same class are still considered a match:

- **"Stringy" types**: Text, String
- **Whole types**: Integer, Boolean, Long, Short
- **Decimal types**: Double, Float
- **Temporal types**: Local date, Local date time, Local time

All of the given data types correspond to a class in Java, with the Text data type also represented as Java's string type. We define boolean to belong into the whole types with the intuition that in certain datasets, booleans are not represented in its own type, but rather described by 0s and 1s. Therefore, we group them together into the same types as number, and also sketch as number as seen in section 5.3.2. While this can cause false positive matches, we hope that by incorporating multiple measures together to obtain a score, the amount of false positives are reduced.

```
1  // metadata: query column
2  // is_join: true if query mode is join, else union mode
3  // query_measures: list of similarity measures
4  QueryResults queryColumn(Metadata metadata, boolean is_join, List<
       MeasureType> query_measures) {
5      QueryResults results = new QueryResults(new ArrayList<>());
6      if (MeasureType.onlyWordNet(query_measures))
7          results.addAll(onlyWordNetQuery(metadata, query_measures,
               is_join));
8      else if (MeasureType.onlyLSH(query_measures))
9          results.addAll(onlyLSHQuery(metadata, query_measures,
               is_join));
10     else
11         results.addAll(mixedQuery(metadata, query_measures, is_join
               ));
12     return results;
13 }
```

Listing 5.11: Overview of the queryColumn() method

Now, if we wish to only find similar columns based on the WordNet similarity measures (algorithm 10), the first step is to retrieve all the metadata from the database that do not belong to the same table as the query column and share a similar types with the query column's data type. Then, for each candidate, we apply algorithm 8, once per WordNet measures present, to calculate the score, wrap the score in the Measure object with the weight assigned for that specific measure and query mode, and add them to the list of measures. Finally, the aggregated score is calculated from all the measures, wrapped in the SingleResult object, and added to the query results.

Algorithm 10 Query columns based only on WordNet measures

Input: Metadata of query column q, list of similarity measures $query\_measures$, query mode m
Output: Query results R
$table\_id \leftarrow$ first part of $q.id$
$candidates \leftarrow findByIdNotStartsWithAndSimilarType(table\_id, q.type)$

for $candidate$ in $candidates$ **do**
  $measures\_list \leftarrow Measures()$
  for $measure\_type$ in $query\_measures$ **do**
    $score \leftarrow calculateScore(q, candidate)$
    $weight \leftarrow$ weight according to $measure\_type$ and m
    $measure \leftarrow Measure(measure\_type, score, weight)$
    $measures\_list.add(measure)$
  end for
  $R.add(q, candidate, measures\_list.aggregate())$
end for

On the other hand, if a column is only queried based on only certain LSH indexes, the algorithm differs slightly (algorithm 11). First, from each index that needs to be queried, the candidates are retrieved, unioned, and filtered for only similar data types. Here, we perform an union on the different candidates and not an intersection as not to filter out any potential matches, since a candidate with zero score in one index but a high score in another one may make it above the threshold. To keep track of which candidates are from which index, we map the measure type to its respective candidates. Then, for each candidate from the union set, the score from the all indexes that take part in the query are returned. If a column is a candidate in one index A but not in the other index B, then we set the score of that column coming from index B to be 0. After that, the algorithm works the same as algorithm 10.

In the case where both categories of similarity of measures are used, then we first follow algorithm 11 to retrieve the candidates from the LSH indexes first, since they would filter out much more columns than relying solely on the data types like in algorithm 10. Then, the WordNet calculations are performed only on the indexes' candidates and added to the list of measures as usual.

While the attempt of filtering out data types in algorithm 10, or querying the LSH indexes first in algorithm 11 reduces the amount of potential candidates before having to apply the WordNet calculation, this step still requires a costly pairwise computation between the query column and all candidates, which can cause a performance bottleneck if the amount of candidates are high. Therefore, it is favorable to avoid this step when it is possible. Notice that when a query request is sent to the server, there is also a query parameter for a threshold, where a candidate becomes a match. If the user set this threshold at 1.0 for example, and a candidate column has one type of measure whose score is lower than 1.0, then it becomes redundant to consider the score of other types of measures, as this column will not become a match anyway.

To formalize this, we define M to be the set of measures used for querying, M_x

Algorithm 11 Query columns based only on LSH indexes

1: **Input**: Metadata of query column q, list of similarity measures used *query_measures*, query mode m
2: **Output**: Query results R
3: *candidates_map* ← mapping from *query_measures* to the respective candidates
4: *candidates* ← union of all candidates from *candidates_map*, filtered by data type
5: **for** *candidate* in *candidates* **do**
6: *measures_list* ← $Measures()$
7: **for** *measure_type* in *query_measures* **do**
8: $score \leftarrow candidates\_map.get(measure\_type).scoreOf(candidate)$
9: *weight* ← weight according to *measure_type* and m
10: $measure \leftarrow Measure(measure\_type, score, weight)$
11: $measures\_list.add(measure)$
12: **end for**
13: $R.add(q, candidate, measures\_list.aggregate())$
14: **end for**

the set of measures whose scores have been calculated and $M_y = M \setminus M_x$ the set of measures that have not been calculated, w_m the weight of measure m, sim_m the similarity score of measure m between the query column and a candidate, t the threshold. A candidate is not a match when the aggregated score is below threshold t:

$$\begin{aligned}
&\frac{\sum_{m \in M} w_m \cdot sim_m}{\sum_{m \in M} w_m} < t \\
\iff &\sum_{m \in M} w_m \cdot sim_m < t \cdot \sum_{m \in M} w_m \\
\iff &\sum_{m \in M_x} w_m \cdot sim_m + \sum_{m \in M_y} w_m \cdot sim_m < t \cdot \sum_{m \in M} w_m \\
\iff &\sum_{m \in M_y} w_m \cdot sim_m < t \cdot \sum_{m \in M} w_m - \sum_{m \in M_x} w_m \cdot sim_m
\end{aligned}$$

The maximum value of any similarity score is 1, so we set sim_m to 1 to find out the stop point for the algorithm:

$$\sum_{m \in M_y} w_m < t \cdot \sum_{m \in M} w_m - \sum_{m \in M_x} w_m \cdot sim_m$$

All the variables in the equation are known at the time of score calculation. We suggest the following optimization algorithm (12) to add to algorithms 10 and 11 as well as the mixed version of them, for example after line 7. When we notice the current candidate is already below the threshold, we immediately break out of the loop and not add the candidate to the QueryResults object.

Algorithm 12 Optimization algorithm

1: **Input**: List of processed similarity measures $query\_measures\_x$, list of unprocessed similarity measures $query\_measures\_x$, list of calculated scores $measures$, threshold t
2: **Output**: Boolean b indicating if the algorithm should stop
3: $total\_weight \leftarrow query\_measures\_x.sumWeights() + query\_measures\_y.sumWeights()$
4: $remaining\_weight = query\_measures\_y.sumWeights()$
5: $current\_score = measures.aggregate()$
6: **return** $remaining\_weight < t \cdot total\_weight - current\_score$

Chapter 6

Experimental Evaluation

In this chapter, we conduct a performance evaluation of the implementation, focusing on the following questions:

- How compact are the size of the datasets' summaries?
- How long does it take to generate the summaries?
- How accurate are the query results?
- What is the impact of the in-memory LSH-Indexes on startup time?
- What is the impact of the optimization algorithm?

All tests are performed on a machine running Windows 11 with 16 GB of RAM, and 11th Gen Intel Core i7-1165G7 @ 2.80GHz processor. The LSH indexes are configured with threshold of 0.7 and MinHash size of 128.

6.1 Datasets

The three datasets used are TPC-DI, WikiData, and synthesized Open Data. The first two datasets were provided by Valentine [10], while the third one by [16].

TPC-DI (∼35MB) is a synthetic dataset used for data integration benchmark. In [10], the authors created schema matching situations by taking the *Prospect* table from TPC-DI 1.1.0 with a scale factor of three, and split the table in various ways: depending on if the situations were join or union, the table is horizontally or vertically split, with varying degree of overlaps between split instances. In some variants, noises were also added to the instances and schemas, i.e, typos, additional characters added or removed from the table name or a value to differentiate the split versions from each other. For our experiments, we random select 3 split table pairs for each schema matching case presented in the paper: joinable, semantically-joinable, unionable and view-unionable. Each table varies from 11 to 22 columns and 7492 to 14983 rows.

WikiData (∼10MB) is a real world dataset that is used for Wikimedia projects. The authors focused on the table *musicians* in the dataset and manually created split pairs of the table for each matching scenarios. In order to replicate a real-life situation, column names are replaced with synonyms (e.g. partner → spouse), and values were change to alternative versions (e.g. Elvis Presley → Elvis Aaron Presley). Each table varies from 13 to 20 columns and 5423 to 10846 rows.

The synthesized Open Data dataset from [16] (∼1,1GB) is used exclusively for union scenario. The tables were generated from a base of 32 tables from Canadian and UK open government data using random projections and selections. Each table ranges from 1 to 40 columns and about 100 to 10000 rows.

6.2 Metrics

Due to varying methodologies in research papers, we employ three metrics to measure the effectiveness of our implementation. The first metric of choice is *Recall@ground_truth* from [10]:

$$Recall@ground\_truth = \frac{\#\text{ of top-k matches}}{\text{k}}$$

where k is the size of the ground truth. This metric is chosen since it shows the ranking's quality of the top relevant results with respect to the ground truth. We use this metric mainly on the first two datasets.

This metric is however not employed in [1], whose implementation we would like to compare with, as it is most similar to ours. There, the system returns ranked matches between tables and not columns. Consequently, the metrics they used: *Precision@k* and *Recall@k* are not fully compatible with ours. Nevertheless, we still measure *Precision@k* and *Recall@k* on the third dataset (which [1] also used) to show that the trends in our diagram as k grows match those in their implementation.

6.3 Single Similarity Measure

First, we compare how each similarity measure fares when being executed in isolation. The result of the experiment performed on the TPC-DI and WikiData datasets is shown in figure 6.1 as a candlestick diagram, depicting the minimum, 1st quartile, 3rd quartile and maximum value of the results. The average results are shown in table 6.1. Both table name measures that rely on syntactic and semantic similarity perform noticeably worse than other measures, recalling only 7% to 14% of matches on average for each matching scenario. This is understandable, as using table name alone to find matches would return all attributes in that table, ignoring the similarities on the column-level. Conversely, the column name measure that uses WordNet for semantic similarity show the highest result, indicating a high correlation between the attribute name and its joinablity/unionability

Measures	Average
Table name WordNet	0.14
Column name WordNet	0.65
Table name q-gram	0.07
Column name q-gram	0.31
Values set overlap	0.48
Value format overlap	0.3

Table 6.1: Average *recall@ground_truth* of individual measures

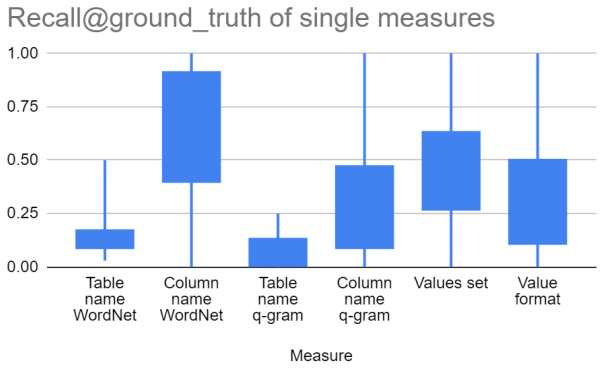


Figure 6.1: Effectiveness of similarity measures when used in isolation

6.4 Combined Similarity Measure

To evaluate the effectiveness of the similarity measures when aggregated, we measured *Recall@ground_truth* on the TPC-DI and WikiData datasets, allowing for comparison with the schema matching techniques in [10]. *Precision@k* and *Recall@k* were measured on the synthesized Open Data dataset with k ranging from 1 to 400, allowing for comparison with [1]. Precision and Recall at each k are the averages computed by perform 20 randomly selected query tables from the dataset. The similarity measures are combined in two different ways: LSH only and mixed:

- LSH only: Use only scores from the LSH indexes (table name q-gram, column name q-gram, values set, values format) to aggregate the results.
- Mixed: Instead of using table name q-gram and column name q-gram, we use WordNet for table name and column name calculation, while keeping the values set and values format from the LSH indexes like above.

Separating the combined measures this way allow us to see whether the additional process of pair-wise semantic WordNet calculation improve the outcomes or not. Additionally, we also want to test if our hypothesized weighted scheme yield better results. Therefore, for each of the two combined measures proposed above, we split the experiment into two weighting schemes: average and weighted average. Average weight means all similarity measures have the same weight when aggregated, while the measures in the weighted average scheme are provided the following weights (table 6.2), which were set up following the intuition from section 5.4.4.

The results of the experiment on TPC-DI and WikiData can be derived from figure 6.2 and table 6.3. We can see that the mixed variants perform on average 11% to 17% better than the LSH-only variants, confirming that the semantics indicate higher relatedness than purely syntax. Interestingly, the experiment

	Join mode	Union mode
Table name	0.1	0.1
Column name	0.25	0.3
Values set	0.4	0.2
Value format	0.25	0.4

Table 6.2: Weighting schema

also showed that the weightings have marginal difference on the outcomes in comparison to the average weighting scheme, performing only slightly better in the LSH-only combinations, and worse in the mixed combinations. This may be explained by the small variance between each weights, making the difference in the contributions of each similarity measure not too pronounced. Therefore, more experiments on different weights should be conducted. Nevertheless, the efficacy of our system falls in line with other schema matching techniques introduced in [10], while running on average much faster (hundreds of milliseconds to a few seconds, in comparison to hundreds of seconds).

Measures	Average
LSH average	0.51
LSH weighted	0.52
Mixed average	0.68
Mixed weighted	0.63

Table 6.3: Average *recall@ground_truth* of combined measures

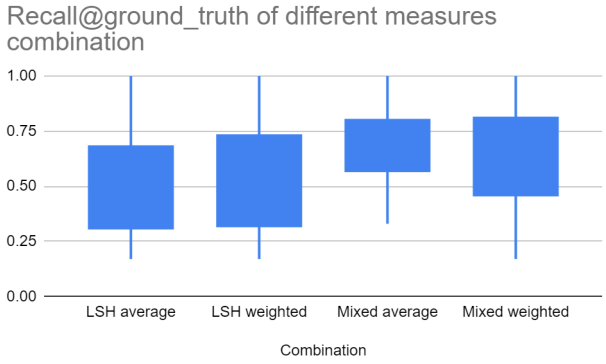


Figure 6.2: Effectiveness of similarity measures when combined

As for synthesized Open Data, we show the *Precision@k* and *Recall@k* for varying k in figure 6.3 and 6.4, where precision declines linearly from around 0.85 to 0.4 and recall grows from 0.01 to 0.7 as k goes from 1 to 400. Here, the differences between each variant of the combined measures are even less pronounced, ranging from 1% to 3%. This also showed that the weights we used

for the experiments were not effective in improving accuracy. Still, at k = 50, the system managed to return results with over 0.75 precision, suggesting that the results at the top are mostly relevant matches. Comparing our result with that of [1], their implementation displayed similar trends, but maintained a higher precision at higher k values than ours. This could be due to several factors: 1) they implements dynamic weighting system such that when calculating the score between a query and a candidate, it also considers the score of other candidates to determine if a similarity measure carries strong relatedness signal; 2) The MinHash size used for their LSH indexes were twice as large as ours (256), providing better performance at the cost of additional storage space; 3) Moreover, we also used a variant of Lazo's MinHash sketch that uses a technique called Optimal One-Hash-Permutation [19] to reduce sketching time, but also comes with the downside of lower effectiveness.

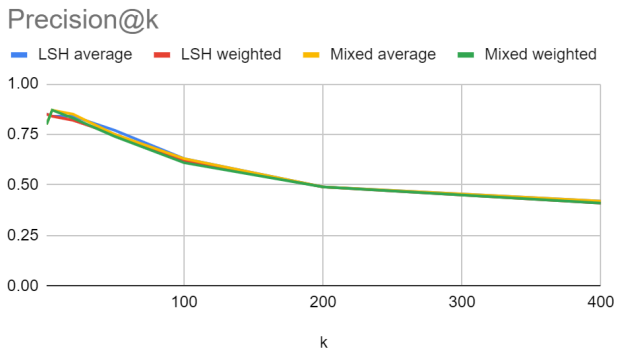


Figure 6.3: Precision at various top-k ranking

6.5 Summary Size

To measure the size of the summaries, we divide all of the datasets in 4 sections. The first section contains about 800 columns, the second 3300 columns, the third 5900 columns, and the last 6000 columns. For each sections inserted, we measure the average sketch size (figure 6.5) and the average metadata size (figure 6.6) stored in the database. With each insertion, the average size of the sketches object remain consistent at 6.63kB. This is understandable, as all sketches store the same amount of hash values. Meanwhile, the average size of a metadata object ranges from about 270B to 300B with each iterations. This is due to the fact that the information stored in the metadata is variable: one table name may be very long while other ones are shorter. In total, the metadata and sketches take up about 40MB of storage in the database, meaning the summaries are only 3,55% of the datasets' original size.

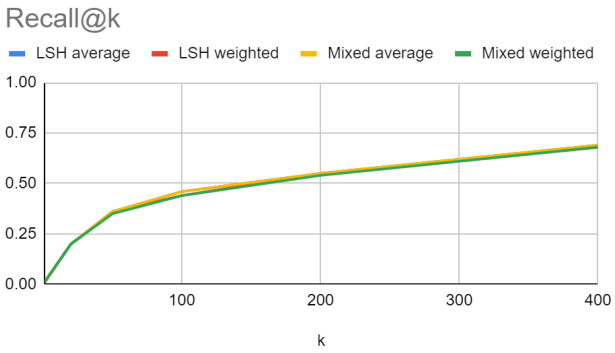


Figure 6.4: Recall at various top-k ranking

Figure 6.5: The average size of the sketches object after each insertions

6.6 Summary Generation Time

To measure the time it takes to generate summaries, we call the profile() method multiple times on the synthesized Open Data dataset, measuring the time it takes for each types of sketch to be generated, and collect the average runtime. Figure 6.7 shows that it took roughly 6 minutes to sketch all columns in the dataset. Most of the time spent were used to generate the values set and the value format, while it took only less than 2 seconds in total for the q-gram sketches of the table name and column name to be generated. Table-wise, it took on average 2 seconds for the profile() method to finish.

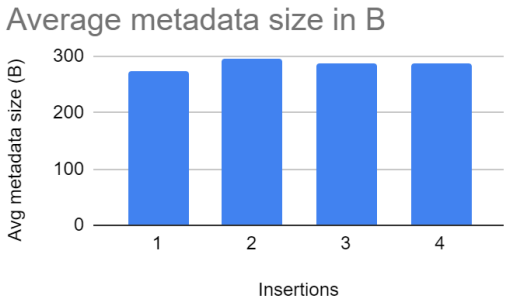


Figure 6.6: The average size of the metadata object after each insertions

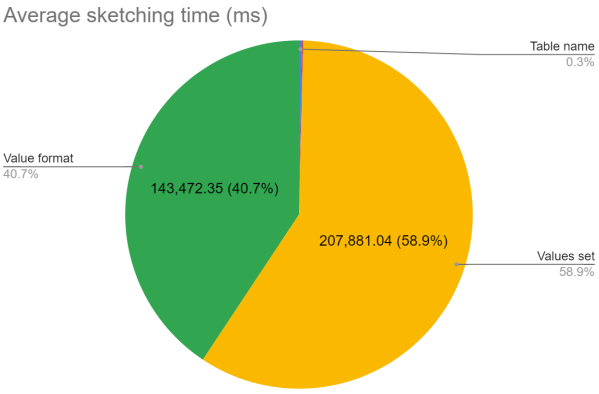


Figure 6.7: Average time taken to sketch the Open Data dataset of 1.1GB

6.7 Effect of in-memory index on startup time

As mentioned in section 4.2, the Lazo's LSH indexes run in main memory. Therefore the sketches need to be loaded from database on startup. We measure the startup time of the system by loading the datasets in 4 sections like in 6.5, restarting the server several times for each section loaded, and calculating the average startup time of each iteration. The result is shown in figure 6.8, indicating that the startup time grows linearly with the amount of sketches in the database. At the maximum database size containing about 40MB of sketches, the server took roughly 20 seconds to start. As we expect the server to not be restarted often, this time should be negligible.

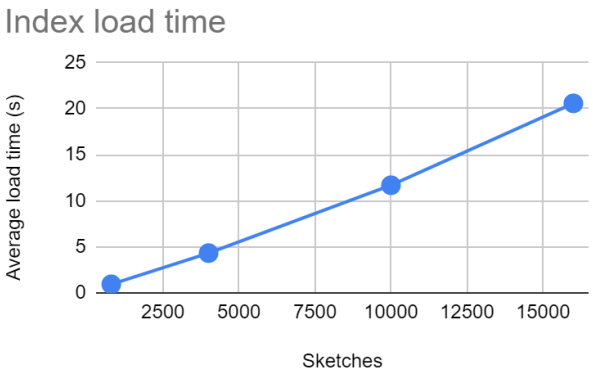


Figure 6.8: Time to startup with increasing amount of sketches

6.8 Effect of optimization algorithm

In this section, we measure the impact of the optimization algorithm 12 on the query time. We set up the experiment by first loading all three test datasets into the database, maximizing the amount of candidates that needs to be checked. Then 10 random tables are selected as query, performed at five different query thresholds: 0, 0.2, 0.4, 0.6, 0.8. The similarity measures used are the mixed weighted combinations from 6.4. The average runtime at each threshold level is shown in figure 6.9. Without any restriction on the threshold, queries take a long time to calculate. Among our 10 selected queries, there were some that take up to 3 minutes to return. The time reduces on average by 30% at the 0.4 threshold, and at 0.8 mark, an average query takes only about 10 seconds, which is 6% of the time taken compared to a 0 threshold. This proves reasonable in practical use cases, as matches with a low similarity score are unlikely to be related anyway.

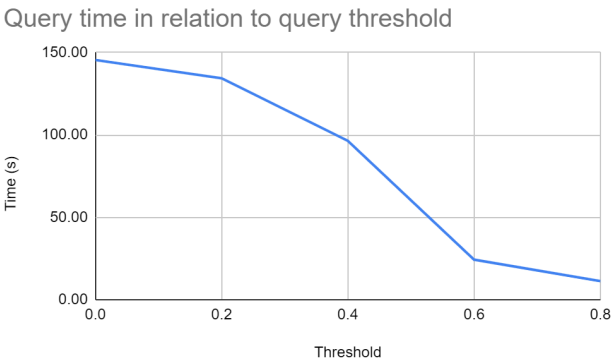


Figure 6.9: Query time at various threshold levels

Chapter 7

Conclusion

7.1 Conclusion

In this thesis, we presented an infrastructure for related dataset search that works in a distributed environment. We discussed how traditional dataset search systems do not provide enough information in order to find related datasets as well as the problem of finding datasets over wireless network. This issue is tackled by summarizing the datasets in a way that the summary is small enough to be able to be sent through the internet, while still maintaining valuable information that allows for similarity comparison. The system provided users an interface to upload datasets' summaries to the server and query for joinable and unionable datasets. Various measures of similarity were aggregated on a column-to-column basis to compare datasets, namely: WordNet semantic similarity, q-gram table and column name similarity, value sets overlap similarity, value format overlap similarity. By using MinHash sketch, we were able to keep the summaries small and still able to calculate the aforementioned measures. By inserting the sketches into modern LSH indexes, accurate results were returned without having to perform costly pairwise similarity computations. An optimization algorithm is also implemented, making use of the query threshold to quickly get rid of negative result and improve query speed.

Through empirical evaluation, we confirmed that our implementation managed to achieve comparable results to other schema matching and dataset search systems, while keeping the dataset summaries compact and transferable across network. Query results were also significantly improved through the use of the optimization algorithm.

7.2 Limitations and Future work

During our implementation process, there are the following shortcomings that arise:

- The weighting system were used to emphasize similarity measures that have higher contributions to the relatedness in different query cases. However, we found out during evaluation process that the weights did not alter the results very much. We believe that the weighting would have more impact if they are properly adjusted using a logistic regression model. However, due to time constraint and the lack of datasets with available ground truth, this was not performed.
- The Lazo's implementation of LSH index and MinHash sketch is efficient and yield relatively accurate results even when the sketch size is small. This comes with the downside however of having to store the indexes in-memory. This introduces a single point of failure in the system, as

well as preventing the system from maximizing the utility of a distributed database, since all communication has to first go through one single server. Future work should therefore look into moving the LSH index into the database itself, removing the need for a dedicated centralized server.

- Currently, the system only accepts CSV files as input for testing purpose. Clearly, datasets exist in other form other than CSV files, so allowing users to provide datasets from other sources such as XML files or importing from databases would greatly improve the usability of the system.

Finally, as mentioned in [10] and [17], schema matching algorithms are not one-size-fits-all, since they depend heavily on how the datasets look like. Our system provides a generalized approach for finding similar relational datasets, but there could be other optimized approaches if we know what kind of data we are dealing with beforehand. For example, to join datasets using geospatial data (latitude and longitude), we can cluster neighboring regions together and only search within those clusters. Also, in our implementation, we focus mostly on using MinHash sketches to find similarity. In future work, other types of sketches (like [9]) could be look into and evaluated to determine their usability and effectiveness in schema matching tasks.

Bibliography

[1] A. Bogatu, A. A. Fernandes, N. W. Paton, and N. Konstantinou. Dataset discovery in data lakes. In *2020 ieee 36th international conference on data engineering (icde)*, pages 709–720. IEEE, 2020.

[2] S. Castelo, R. Rampin, A. Santos, A. Bessa, F. Chirigati, and J. Freire. Auctus: A dataset search engine for data discovery and augmentation. *Proceedings of the VLDB Endowment*, 14(12):2791–2794, 2021.

[3] A. Chapman, E. Simperl, L. Koesten, G. Konstantinidis, L.-D. Ibáñez, E. Kacprzak, and P. Groth. Dataset search: a survey. *The VLDB Journal*, 29(1):251–272, 2020.

[4] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[5] H.-H. Do and E. Rahm. Coma—a system for flexible combination of schema matching approaches. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 610–621. Elsevier, 2002.

[6] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012. IEEE, 2018.

[7] R. C. Fernandez, J. Min, D. Nava, and S. Madden. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1190–1201. IEEE, 2019.

[8] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[9] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng. Locality-sensitive bloom filter for approximate membership query. *IEEE Transactions on Computers*, 61(6):817–830, 2011.

[10] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, and A. Katsifodimos. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE, 2021.

[11] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[12] V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[13] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *vldb*, volume 1, pages 49–58, 2001.

[14] R. Malik, L. V. Subramaniam, and S. Kaushik. Automatically selecting answer templates to respond to customer emails. In *IJCAI*, volume 7, page 3015, 2007.

[15] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[16] F. Nargesian, E. Zhu, K. Q. Pu, and R. J. Miller. Table union search on open data. *Proceedings of the VLDB Endowment*, 11(7):813–825, 2018.

[17] N. W. Paton, J. Chen, and Z. Wu. Dataset discovery and exploration: A survey. *ACM Computing Surveys*, 56(4):1–37, 2023.

[18] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10:334–350, 2001.

[19] A. Shrivastava. Optimal densification for fast and accurate minwise hashing. In *International Conference on Machine Learning*, pages 3154–3163. PMLR, 2017.

[20] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, 92(1):191–211, 1992.

[21] Z. Wu and M. Palmer. Verb semantics and lexical selection. *arXiv preprint cmp-lg/9406033*, 1994.

[22] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. Lsh ensemble: Internet-scale domain search. *arXiv preprint arXiv:1603.07410*, 2016.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Kaiserslautern, _____ _____

Date Signature