Delft University of Technology

Masters Thesis

Holistic Schema Matching at Scale

Author:
Kyriakos Psarakis

Supervisor:
Dr. Asterios Katsifodimos

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

Student number:	4909437	
Thesis committee:	Prof.dr.ir. G.J.P.M. Houben,	TU Delft, chair
	Prof.dr. A. van Deursen,	TU Delft
	Dr. A. Katsifodimos,	TU Delft, supervisor

December 3, 2020

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Holistic Schema Matching at Scale

by Kyriakos Psarakis

Schema matching is a fundamental task in the data integration pipeline and has been studied extensively in the past decades, leading to many novel schema matching methods. However, these methods do not follow a standard evaluation process, leading to uncertainty in which one performs best in matching accuracy and runtime constraints, and in which specific schema matching category, and with what hyperparameters. To clear the confusion, the need for a scalable benchmarking suite to determine the field's progress became apparent, leading to the first contribution of this work, a scalable benchmarking suite for schema matching tasks.

In the meantime, we realized that the literature lacked a scalable holistic schema matching system, leading to our second contribution. By considering the knowledge gained from our proposed benchmark, we developed a system that can incorporate any algorithm and data source while running the schema matching jobs in parallel across multiple machines in a scalable fashion.

Furthermore, we decided to give a leading role to the users of such a system. The reason behind that is that it became apparent in the benchmark that no algorithm is perfect in every situation, and in mission-critical applications, we cannot afford any mistakes. Thus, the users would have to approve the proposed matches, and we focused on making this task scalable, fast, and straightforward.

Acknowledgements

At first, I would like to thank my supervisor, Asterios, for the opportunities that he gave me to prove myself, the trust that he showed, and the feedback throughout the process of my thesis.

Moreover, I would like to thank the Delta team for all the nice moments that we had working together. Especially, Marios and Christos for always being there whenever I required guidance.

Furthermore, since I completed a part of my thesis as a research internship at ING in the AI for Fintech initiative. I would like to thank my colleagues there and specifically Jerry Brons for giving me the opportunity to learn how research can be applied in the industry.

I would also like to thank the members of the committee for giving their time to evaluate this work.

Finally, I would like to thank my friends and family for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, corporations and institutions are witnessing an unprecedented increase in data volume and variety that they need to handle to meet business needs. A recent article Reinsel et al. [2017] showed that the volume of data worldwide has increased by approximately 20 times since 2010, and it forecasts an exponential increase from then onwards. The same article predicts that corporate data will increase in share percentage, from about 30% of the total volume to 40%. Furthermore, the data variety has diversified, coming from many end-user-focused devices and systems or different teams within the same company (i.e., microservice design) in different, more flexible formats.

These predictions have led enterprises to consider adopting a more versatile system to store data. In the past, companies mostly used data warehouses as a central repository of storing data to meet their business needs. This approach's primary benefit is that data inside are curated and presented in a single data model. In such a system, schema matching can increase the data model's quality by finding links between the different databases inside the warehouse.

Although data warehouses are essential in business intelligence, they can become a bottleneck in large-scale machine learning and predictive analytics applications where the data velocity and volume make the curation process cumbersome and impractical. In that case, data lakes emerged as the preferred option. The main difference between them is that data are stored directly into the system, leaving the curation process to the end-user. Thus, in data lakes, schema matching can enhance the datasets used by the ML practitioners, increasing the feature size by finding joinable relations or increasing the number of datapoints by finding unionable relations. To add to that fact, in a recent paper from Google Sculley et al. [2015] explains that the actual ML code is only but a small part of the ML pipeline and other surrounding components, such as data integration, are more complex than credited tacking up to 21% of the development efforts Figure 1.1. Furthermore, in Nargesian et al. [2019] lists data integration as one of the biggest challenges in data lake management and underlines the need and challenge of on-demand schema matching and mapping.

The main challenges that we identified are (i) there is no way to know which schema matching method performs best, (ii) the scalability problem is getting bigger each passing year and cannot remain as a simple feature of the system, but its primary concern and (iii) matching systems tend not to use the humans in the loop efficiently.

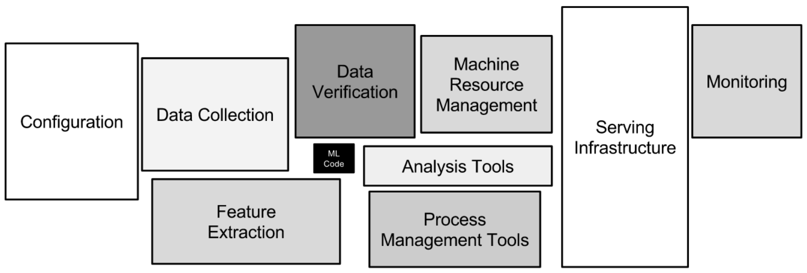


Figure 1.1: The pipeline of real-world ML systems as shown in Sculley et al. [2015].

In this work, we first create a scalable benchmark for schema matching methods that is a part of Valentine[1] Koutras et al. [2020b] to find which algorithms perform best in which matching scenario. Secondly, we integrated the best performing methods to a holistic schema matching system that operates at scale. Finally, we provide a user interface that makes the system work in a semi-automated way. We argue that this is the most sensible way to perform schema matching and aim to minimize the time spent in the match verification process.

1.1 Problem Statement

Schema matching is the process of identifying matching elements between one or multiple schemas. The field has received much attention in the past decades with three high-profile surveys showcasing the existing methods and taxonomizing them based on the techniques used to create the method and the data used (either schema level data or/and the data instances). However, the schema matching problem itself can be understood in different ways, and the exact applications of schema matching inside the general area of data integration are still undefined. Besides, a discussion on the specific use cases where schema matching can be used is also still missing. Additionally, most recent research has shifted from generic schema matching to the more specific joinable Zhu et al. [2019] or unionable Nargesian et al. [2018] attribute search, proving that the scope of the problem is essential in the method design.

Therefore, in this work, we will consider the following scenarios, where existing schema matching methods can be applied. We develop a schema matching scenario taxonomy with two fundamental categories,unionable and joinable relations, and further refine each of these categories. This taxonomy will guide our evaluation in the benchmarking system Chapter 3 as different methods can better cope with particular problem cases than with others. The general schema matching use case is the existence of relations containing data from similar or related concepts, scattered across disparate locations. Possibly, they might also use different schema conventions or encoding styles. We identify two main cases, with one subcase each, and depict them in Figure 1.2.

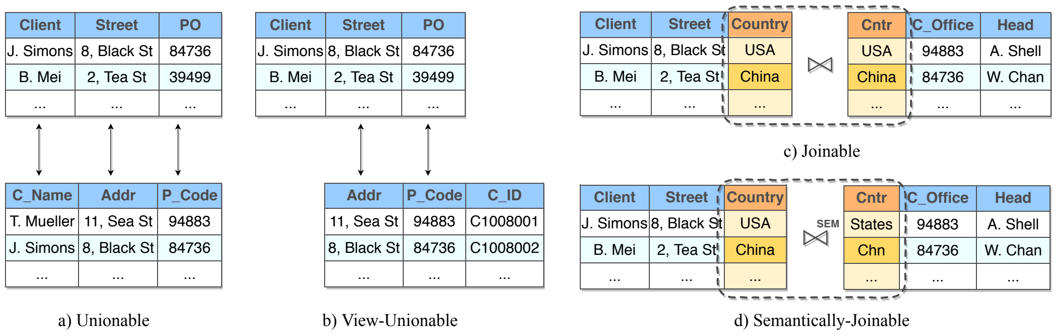
Unionable Relations. The first one is that different relations refer to the same entity type focusing on similar aspects, i.e., having similar attributes (like

[1] https://github.com/delftdata/valentine

customer data in different departments). Data is essentially horizontally partitioned or even duplicated between relations. In this case, the relations could be merged by *unioning* them, which could provide more data-points to business intelligence or machine learning applications. The subcase of unionable relations is the **view-unionable**, where the relation share attributes that correspond to each other semantically but also contain attributes that are unique to each. This could be a more typical case since data that are partitioned across different sites may be differently modeled under the respective data owner's conventions. More specifically, each such data shard may be enhanced with information (in our case attributes) relevant to each owner, making it challenging to identify the similarity between relations that refer to the same data.

Joinable Relations. The second case is relations focusing on the same entity type but different aspects, which consequently would have different attributes. Data would be vertically partitioned between relations. An example would be applicat-ion-specific customer data, e.g., customer data in the marketing department could focus on different information than data on the same customers in the billing department. In this case, relations should be merged by *joining* them, resulting in the *joinable relations* leading to more feature-rich datasets for data science teams. Its subcase is **semantically-joinable relations**, meaning that the join happens on semantically equivalent values, and it resembles fuzzy-joins. For example, a human knows that "Netherlands" and "NL" point to the same country, but these values do not exactly match. Some pioneers of the field have tagged schema matching as an AI-complete problem Bernstein et al. [2004] due to the approximate semantically equivalent relations that we discussed earlier. That fact leads to state-of-the-art systems that recommend correspondences to experienced data engineers, increasing their productivity, and not fully automated data integration.

In most practical applications, we expect to find a mix or even hybrids of both **Unionable** and **Joinable** cases.



Client	Street	PO
J. Simons	8, Black St	84736
B. Mei	2, Tea St	39499
...	...	...

C_Name	Addr	P_Code
T. Mueller	11, Sea St	94883
J. Simons	8, Black St	84736
...	...	...

Client	Street	PO
J. Simons	8, Black St	84736
B. Mei	2, Tea St	39499
...	...	...

Addr	P_Code	C_ID
11, Sea St	94883	C1008001
8, Black St	84736	C1008002
...	...	...

Client	Street	Country
J. Simons	8, Black St	USA
B. Mei	2, Tea St	China
...	...	...

Cntr	C_Office	Head
USA	94883	A. Shell
China	84736	W. Chan
...	...	...

Client	Street	Country
J. Simons	8, Black St	USA
B. Mei	2, Tea St	China
...	...	...

Cntr	C_Office	Head
States	94883	A. Shell
Chn	84736	W. Chan
...	...	...

Figure 1.2: The four cases of schema matching scenarios considered in this work.

In the next sections, we showcase the unresolved problems that we identified in schema matching literature.

1.1.1 Comparability between methods

Over the years, many schema matching methods have been proposed. However, as shown in Table 1.1, most of them use their own test data for their method's evaluation. For example, some of the most prominent papers in the field used in our first contribution Koutras et al. [2020b], Cupid Madhavan et al. [2001], COMA Do and Rahm [2002], and Similarity Flooding Melnik et al. [2002] use very simplistic schemata for evaluation drawn on each respective paper. The same behavior holds too for the Distribution Based method Zhang et al. [2011], EmbDI Cappuzzo et al. [2020], and SemProp Fernandez et al. [2018a], where although they use more complex datasets than their predecessors, the only match is between the distribution based and SemProp using the TPC-H dataset[2]. Those mentioned above create a significant issue where practitioners need to decide on which method to choose. If no direct comparison is made in the defined scenarios in Table 1.1, then there is no telling of which method performs best on what specific case. Thus, a benchmarking system is necessary to make that distinction.

	Method					
Dataset	Cupid	COMA	Similarity Flooding	EmbDI	Distribution-Based	SemProp
Simplistic	✓	✓	✓			
EmbDI datasets				✓		
TPC-H					✓	✓
SemProp datasets						✓

Table 1.1: Datasets used for method evaluation

1.1.2 Scale

The following problem is the one that we mostly focus on in this work, and it relates to the scale of schema matching problems. Firstly, we would like to address the scale of the schema matching benchmark. To start with, we would need to have multiple datasets that encompass all the separate schema matching scenarios, mentioned earlier in Figure 1.2, with varying degrees of semantic ambiguity that increase the difficulty in a gradual manner. Moreover, most methods have hyperparameters, and to get the optimal result out of them, the benchmark would have to perform a grid search. Grid search, which mathematically is the cartesian product of an array of arrays, has a complexity of *O(|first hyperparameter's distinct values| * |second hyperparameter's distinct values| * ...)* which is the precise bound. To simplify this, the array sizes could be bound by their average size m, and if the number of the arrays is n then the complexity would be $\Theta(m^n)$. This complexity is for just a grid search run on a dataset. Thus, if we want to create an extensive benchmark, its ability to scale and distribute the workload is a necessity.

Furthermore, there is little to no work done in the field of schema matching systems addressing the problem of scalability. The best open source state-of-the-art system COMA Massmann et al. [2011] is stale since 2011, and the authors themselves have listed some scalability issues. Much work is done in speeding up schema to schema matching as we showcase in Section 2.3 but, nobody addressed the elephant in the room, which is the holistic schema matching case (meaning to find matches between all the data sources of an organization). Again this is a cartesian product of an array of arrays. If k is the average size and l

[2] `http://www.tpc.org/tpch/`

the amount of the schemata, then the complexity is $\Theta(k^l)$ without considering the hyperparameter grid search that we mentioned previously. Consequently, a scalable distributed schema matching system is needed to address the holistic schema matching case.

1.1.3 Humans in the loop

The final problem that we identified in the matching systems' user interfaces is that the way of presenting the proposed matches (connected lines between elements) tends to fail with large schemata, as shown in Figure 1.3 in the work of Robertson et al. [2005]. Additionally, the previous systems cannot scale beyond one person in the match verification task, creating a bottleneck. Thus, we aim to redesign the user interface taking both of those facts in mind.

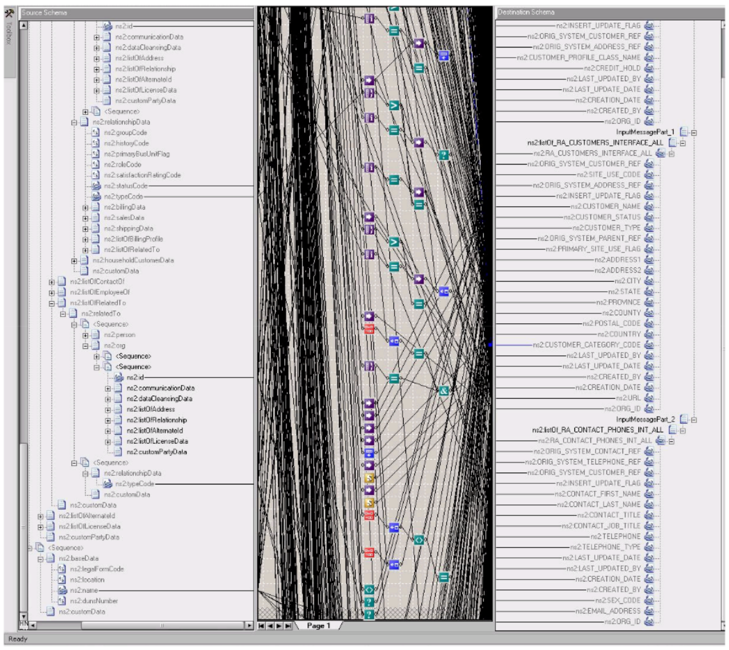


Figure 1.3: The failure of line visualizations in large scale schema matching as shown in Robertson et al. [2005]

1.2 Research Questions

Based on the problem statement, we have identified two gaps in schema matching literature. The first gap is the lack of a standardized way of testing schema matching methods (benchmark). Due to that, it is hard to tell if there is any actual progress in the field and which case (joinable, unionable), leading to our **first research question**:

> Is there any real progress in schema matching in the past two decades?

The answer to our first research question would be to create a benchmark for schema matching methods. However, one should also consider the scale of such a venture, the four different categories, as shown in Figure 1.2, the different levels of approximation in schema elements or data instances, and the number of hyperparameter combinations of each algorithm, leading to our **second research question**:

> How to construct an extensive schema matching benchmark with a primary focus on its scalability?

The second gap that we identified is the lack of a scalable holistic schema matching system. There is much work done in the field of schema matching systems, as surveyed in Section 2.2. However, none of them expands past one computing node, and the most popular solution Do and Rahm [2002] is unable to handle large schemata as expressed by its authors in Massmann et al. [2011], leading to our **third research question**:

> What should be the design of an elastic, scalable, semi-automated, holistic schema matching system?

As shown in Robertson et al. [2005], the way of showing the proposed matches to the users in the current systems does not work in large schemata or in the holistic case and does not scale beyond a single user without the need for communication between users, leading to our **fourth research question**:

> How to present the proposed matches to multiple users, increasing the speed of match verification?

1.3 Contributions

In this section, we outline the contributions that emerged in the process of answering our research questions in the same order, as expressed in section Section 1.2.

1. Scalable and extensible benchmarking system[3] Koutras et al. [2020b] that practitioners could use to test and develop new schema matching methods that improve the status quo and insights on which method to use in which schema matching case.
2. A cloud-native holistic schema matching system that can scale to multiple machines to match the workload and have the ability to support every data source.
3. A user interface that can be accessed by multiple users simultaneously, scaling the match verification process. Additionally, we display the proposed matches in a simpler way to decrease the time users spend approving matches.

1.4 Outline

The rest of the thesis is structured as follows. In Chapter 2 we conduct a literature survey on schema matching, the benchmarks conducted in the field, the academic and industrial systems that have a schema matching component, how those approach the scalability problem, and finally, since we agree with the semi-automated approach, how do they use the user input to verify the proposed matches.

Chapter 3 details the design choices made when creating a Valentine, a scalable schema matching benchmark. In Chapter 4, we showcase the design and implementation of a scalable holistic semi-automated schema matching system, using the lessons learned from the benchmark.

Following the above, in Chapter 5, we report the experiments conducted in the benchmark in the selected algorithms, both in time efficiency, matching effectiveness, and scaling. Furthermore, we conduct scalability tests in the proposed schema matching system to identify its throughput and scaling efficiency. In both sections, we also discuss the findings arising from the empirical evidence collected.

Finally, Chapter 6 concludes the thesis by summarizing our work, its limitations, and proposing directions for future research.

[3]https://github.com/delftdata/valentine

Chapter 2

Related Work

Schema matching or schema integration is a well-established research field in the database community, studied for over thirty years. The schema matching problem first appeared when organizations started to maintain multiple databases to meet their business needs. The survey of Batini et al. [1986] detailed the need to merge different schemata into a global one and surveyed the methods proposed until that time. These methods mostly focus on designing the global schema and how the distributed ones will fit in. When the design did happen, another section of methods proposes methodologies on how the global schema can be constructed manually by resolving conflicts, for example, structural or naming and so on, during the merge of the separate schema. However, the implication was that every team would follow the same database design rules and practices, something that would only work on a small scale. It is not practical, as it requires much effort from the database administrators as it focuses on the entity-relationship model, which imposes a lot of strict merging rules and cannot work in a more relaxed scenario. Finally, most of the proposed methodologies were parts of research projects with low emphasis on developing full-scale automated systems.

The first most prominent automated methods, superseding the manual era, were surveyed in Rahm and Bernstein [2001]. The methods were classified based on whether the method was a single matcher or an ensemble of combined matchers. Each matcher was classified based on the type of information it required, creating a taxonomy shown in Figure 2.1. Shema-based, for matchers that use the schema information like the similarity between the element names, constraints (keys and data-types), or the schema's structure when the database follows a hierarchical model (XML, for example). Instance-based using the data instances, for example, word frequencies, patterns, cardinalities, to name a few. In this work, the authors suggested that more methods should incorporate instance-level information because they observed that only a few used them. Additionally, they expressed the need for a system that can quantitatively compare the methods between each other since every method used a different domain, making the performance results incomparable.

Later, the survey of Shvaiko and Euzenat [2005] extended the taxonomy of Rahm and Bernstein [2001] in the schema-based methods. Firstly, the authors distinguish that methods can either be approximate or exact, where exact methods compute the absolute solution to the matching problem while approximate methods accept some errors to reduce the processing time. They then extended the taxonomy of schema matching techniques, as shown in Figure 2.2, by interpreting the input information in three categories syntactic, external, semantic,

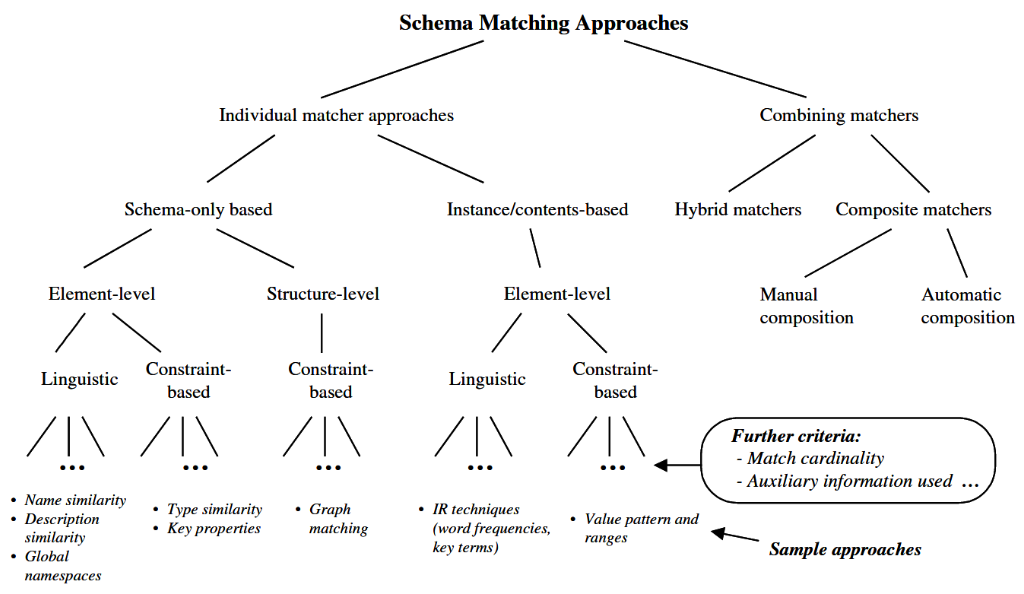


Figure 2.1: The taxonomy of schema matching methods proposed in Rahm and Bernstein [2001]

and their kind, into terminological, structural, and semantic techniques. Furthermore, they observed that most of the algorithms, in that time, used syntactic and external techniques due to the semantic ones being new. They also defended the lack of instance-based methods in their taxonomy by pointing out that it is not always feasible to get the data instances in some applications. However, they acknowledge the potential of instance-based approaches. Finally, they agree with Rahm and Bernstein [2001] that there is a need for a large-scale benchmark to measure both the matches' quality and the system's performance.

The authors of Rahm and Bernstein [2001] in their most recent position paper about developments in schema matching Bernstein et al. [2011] agreed with the updated taxonomy of Shvaiko and Euzenat [2005]. Additionally, they made the following observations that even the best schema matching algorithm can make mistakes, especially the fully-automatic. However, they argue that even erroneous results can be helpful in some cases, such as when they contribute only implicitly to some end-user task results. In the semi-automatic approaches, the authors state the importance of having excellent graphical support when there is a human in the loop. Furthermore, they pointed out that despite the progress in mapping reuse, it has not yet been integrated, up to that point, into any system. Finally, they again state the importance of a benchmark.

In addition to the previous surveys, the book Bellahsene et al. [2011] offers a comprehensive survey on schema matching till 2011. It contains three chapters. The first deals with schema matching, e.g., methods, systems. The second focuses on schema mappings and their use for schema evolution and schema merging. Finally, the third addresses evaluation and tuning of schema matching and mapping systems.

After 2011, a part of schema matching research specialized into subcategories based on the scenario, i.e., whether the two candidate tables are joinable or unionable. In the joinable subcategory, approaches are aiming to speed up the computation using the map-reduce paradigm Deng et al. [2015], others are

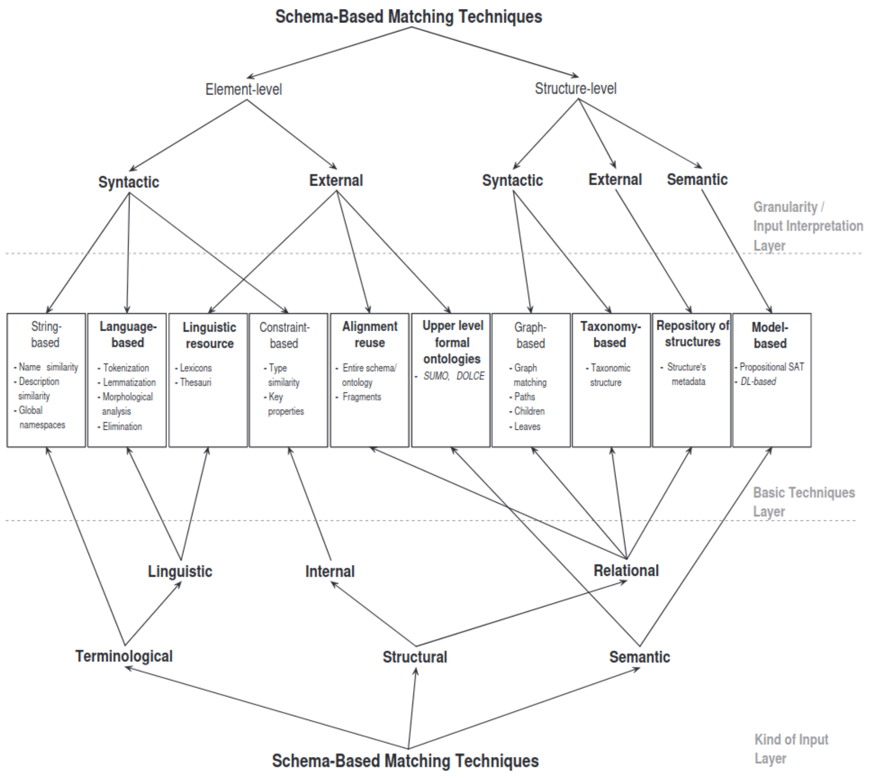


Figure 2.2: The extended taxonomy proposed in Shvaiko and Euzenat [2005]

mainly using locality sensitive hashing(LSH) Zhu et al. [2016]; Fernandez et al. [2019], and others sketches Nazi et al. [2018]. The unionable subcategory is closely linked with the joinable, meaning that if all columns from a table can join with one other from another, they are unionable. The most prominent work is Nargesian et al. [2018], which uses LSH to find the set similarity between the columns and embeddings for approximate(fuzzy) natural language matches. Finally, following the deep learning trend, the most recent schema matching methods try to work with embeddings, which are vector representations learned from input datasets Fernandez et al. [2018b]; Koutras et al. [2020a]; Cappuzzo et al. [2020]. In this case, the similarity between the two columns is high when the distance between the two vectors representing them is low.

2.1 Schema Matching Benchmarks

To answer our first research question, we investigated whether the existing benchmarks in schema matching are sufficient. The first attempt at a schema matching benchmark is XBenchMatch Duchateau et al. [2007] that introduced a preliminary system for evaluating schema matching tools, yet it showcased three schema-based methods focusing only on XML data. XBenchMatch received an update Duchateau and Bellahsene [2014] with more XML datasets, additional methods, and performance metrics, making XBenchMatch a more thorough benchmark. However, it is not complete as it again excludes instance-based methods and other data formats. XML is standard in many websites and applications, but it is not as popular in the database domain.

There is no other benchmark directly aimed at schema matching methods. However, there are works closely related. STBenchmark Alexe et al. [2008] is one of those, and it focuses on evaluating mapping systems. The difference between the two is that while schema matching systems focus on obtaining a set of mapping elements, mapping systems focus on how data in different schemata are related. iBench Arocena et al. [2015] is a metadata generator for creating data integration tasks, i.e., create mappings, schemas, and schema constraints. Additionally, it can add instances to the specific data integration task if coupled with a data generator and used as a component to create datasets used in a schema matching benchmark. Finally, eTuner Lee et al. [2007] provided a method for automatically tuning the hyperparameters of schema matching systems.

Thus we concluded that there is a gap in the literature regarding a benchmark for the following reasons:

- The related work does not provide any benchmark that encompasses the entire spectrum of schema matching problems, i.e., supports only schema-based methods with XML data as input.
- They do not explicitly consider the different matching cases expressed in section Section 1.1.
- The benchmarks that exist are not open source and cannot be used by the community to add new algorithms, datasets, or evaluate its credibility.

2.2 Schema Matching Systems

Over the years, many attempts have been made at creating schema and ontology matching systems Rahm [2011]. In this work, we will only consider those that could work with schemata of any kind (i.e., relational, XML) without the need for an external ontology and provide a user interface that allows users to verify the proposed mappings.

COMA Do and Rahm [2002] is a generic matching system. It supports the multimatcher architecture and match workflows. In its current version Massmann et al. [2011] COMA supports both instance and schema-based matching with many simple linguistic and structural matchers. In terms of scalability, it works on a single machine in multiple JVM threads, but it suffers from some memory and runtime problems for large match tasks, as reported in Massmann et al. [2011]. To cope with those performance issues, the authors reduce the problem space by using fragment matching, searching for matches between sets of high match probability elements.

Clio Miller et al. [2001], along with COMA, is one of the first schema matching tools. Its matching component Nauman et al. [2002] is relatively simple, using a naive Bayes classifier using the attributes' characters as features for the schema-based matching and instance-based when categorical data are in the column. For numerical instance data, the component uses three quantile based classification methods. Clio can also use information about the matching input from a user to infer and re-rank correspondences. Additionally, Clio provides its users with tools to reformulate queries from one schema to another or transform data from one representation to another to facilitate data exchange.

BizTalk Bernstein et al. [2006] is a product of Microsoft, and it shares many similarities with COMA. Its main improvements are in the user interface area and, more specifically, how the user will better understand the proposed matches and spend less time in the verification task.

OpenII's Harmony Seligman et al. [2010] is a hybrid matcher that combines the scores generated by a collection of schema-based linguistic matchers. The system's novelty is that it exploits textual definitions of schema elements (if they exist). Its GUI is similar to the previous ones, it does not support instance-based matching, and it can combine match results, as following the method of Mork et al. [2008]. In terms of scale, it claims to handle schemas with up to 1000 elements.

BigGorilla's FlexMatcher Chen et al. [2018]. BigGorilla is an initiative that encompasses open-source projects in the entire data integration/preparation pipeline. The project of interest to this work is BigGorilla's schema matching tool, FlexMatcher that is a reproduction of the LSD algorithm Doan et al. [2003], which uses several existing schema mappings that serve as training samples for a classifier. No attempts in scaling this system, and no UI to verify the proposed mappings are in the roadmap. However, we decided to include that since it is the latest attempt in the field.

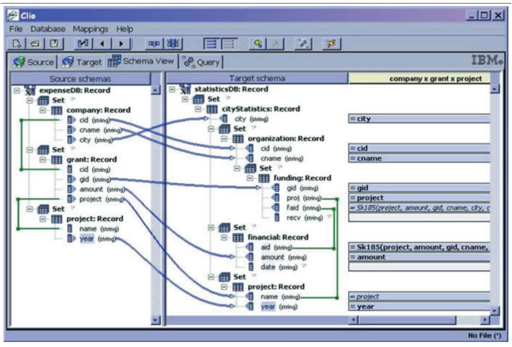
2.3 Scalability in Schema Matching

As we explained in Section 1.1.2, scalability in a holistic schema matching system is the biggest issue with complexity $\Theta(n^m)$ where n the number of elements in the largest schema and m the number of schemata that we want to integrate and, this complexity is without taking into account the hyperparameter tuning, that almost every schema matching method requires. This complexity results in long execution times and makes it challenging to find the small number of actual mappings between all the possible candidates.

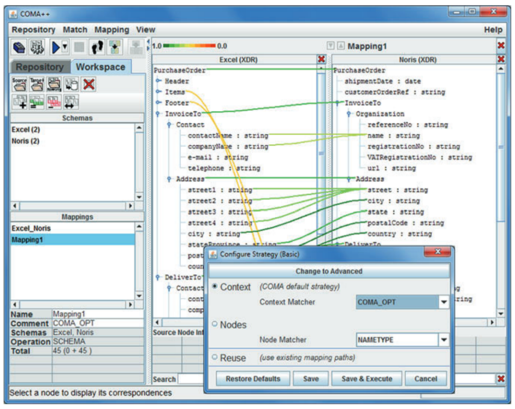
To answer our third research question, we research how the state of the art schema matching systems deal with the scalability problem in schema matching. The works of Rahm [2011]; Rahm and Peukert [2019] provide a detailed overview of how research deals with large scale schema matching, and it contains four major categories: parallel matching, reducing the search space, use of previous match results, and self-tuning match workflows.

At first, **parallel matching** is a proven concept in ontology matching. GOMMA Gross et al. [2010] and SPHeRe Amin et al. [2016] use a job queue to distribute the tasks across multiple computing nodes. However, no schema matching system supports it, and the previous works are domain-specific and hard to extend.

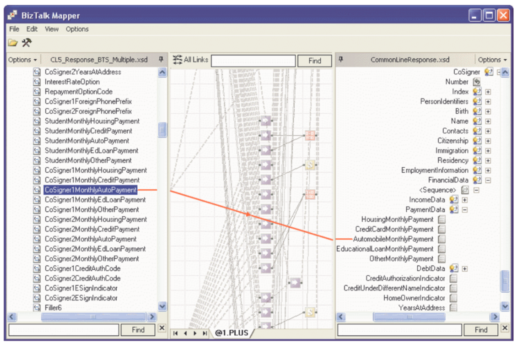
Another approach is to **reduce the search space** and is encompasses two subcategories. (i) early pruning of dissimilar element pairs and (ii)partition-based matching. In (i) research Ehrig and Staab [2004] and Peukert et al. [2010] explore whether applying simple/fast matchers in the beginning and pruning the combinations with low similarity. However, they do not prove that every actual match makes it through the pruning. These methods have not been tested in largely heterogeneous schemata or where a schema is absent (No-SQL JSON data). In (ii), systems are partitioning the input schemata to restrict the search of similar schema elements to a subset of these partitions Aumueller et al. [2005].



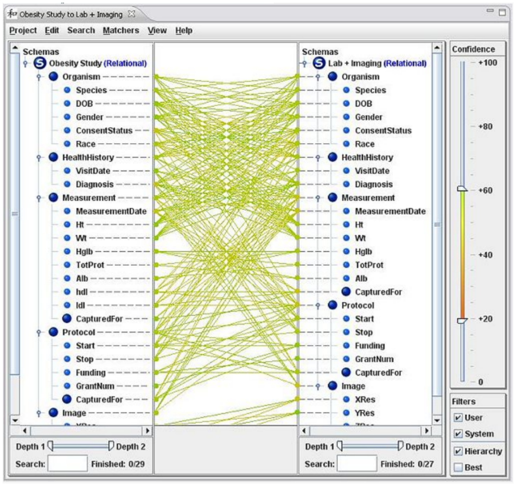
(a) Clio Miller et al. [2001]



(b) Coma Do and Rahm [2002]



(c) BizTalk Bernstein et al. [2006]



(d) OpenII's Harmony Seligman et al. [2010]

Figure 2.3: The user interfaces provided by the surveyed schema matching systems

The partitioning follows the same logic as (i), placing similar elements in the same partition and then evaluating them separately. Both (i) and (ii) are open research problems.

Other than the above, the **reuse of previous match results** can help with large scale schema matching by inferring matches. Two different styles are doing that. The first Madhavan et al. [2005] creates a domain-specific corpus, maps each match to that, and concludes that two elements match if they map to the same corpus element. The second uses results from previous runs to infer new ones where applicable. For example, if S1 matches with S2 and S2 with S3, then it is safe to conclude that S1 matches with S3 in its respective elements, without the need to run extra schema matching tasks. Both COMA Aumueller et al. [2005] and GOMMA Gross et al. [2010] use this method. The only minor drawback of this method is that it can make a system more complex due to its persistent storage requirements for keeping the verified matches or the corpus.

Finally, **self-tuning match workflows** Ehrig et al. [2005]; Lee et al. [2007] reduce the need for reruns with different method hyperparameters, which add complexity to an already complex system. They achieve that by using previously solved match tasks, with verified mappings as the training set, to find the optimal parameter settings. This approach's main issue is that it is not tested for largely heterogeneous schemata in the holistic case where the optimal parameters could be vastly different from one case to the next. To remedy that, it would require large amounts of training data, which are verified mappings in evenly distributed schemata.

2.4 Expert Evaluation

No schema matching method is perfect in every situation, and most use cases are mission-critical, making the fully-automatic approach not realistic. The status quo in the surveyed systems is a user interface showing the match results that follow a schema matching task's execution (presented in Figure 2.3). All systems show the match candidates with colored lines (green: similar, red: dissimilar). This style started from Clio Miller et al. [2001], and the others (i.e., COMA Do and Rahm [2002], BizTalk Bernstein et al. [2006], Harmony Mork et al. [2008]) followed with some improvements. This style assumes that the user who requested the schema matching task would also verify the mappings, which does not scale in large schemata or the holistic case (more than two schemata). Furthermore, this approach of showing results is proven to fail with large schemata Robertson et al. [2005] the same work proposes some enhancements to eliminate clutter and reduce the time it takes for a user to inspect the results. Although the improvements are substantial, the task time is still above 10 seconds.

There is another different approach to this problem coming from the crowd computing field. Its primary focus is to alleviate the pressure of the match verification task by distributing it in microtasks to crowd workers. These workers possessed no prior knowledge about the domain. CrowdMatcher Zhang et al. [2014] proves that this method can scale while maintaining a low cost by asking the crowd workers first to verify matches with high similarity. The work of Hung et al. [2013] also explores this notion while improving the microtask design, an example of such a task is given in Figure 2.4. While this approach could work and is extremely useful on open data, companies with sensitive data cannot distribute them to crowd computing platforms.

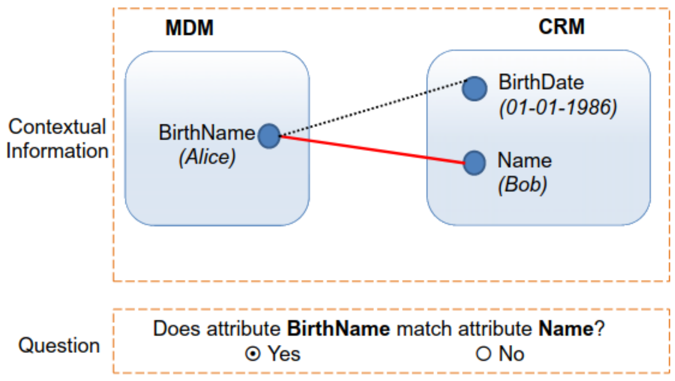


Figure 2.4: An example of a microtask given in Hung et al. [2013]

Chapter 3

Schema Matching Benchmark

This chapter goes through our first contribution, a scalable schema matching benchmarking system that is the central component of Valentine Koutras et al. [2020b] that is, to the best of our knowledge, the most comprehensive effectiveness and efficiency evaluation of schema matching algorithms for tabular data to date. The aim is to answer our first research question, "Is there any real progress in schema matching in the past two decades?". At first, in Section 3.1, we detail our initial selection of methods tested in the benchmark. Then in Section 3.2, we showcase the architecture of the system, answering our second research question, "How to construct an extensive schema matching benchmark with a primary focus on its scalability?". Following that, in Section 3.3, we express the need to parallelize the most demanding methods and the actions required to do this. Furthermore, we briefly present the dataset fabrication process in Section 3.4 and, finally, the performance metrics supported in the benchmark in Section 3.5.

3.1 Selected Methods

In this work, we chose some of the most prominent schema matching methods that stood the course of time and the most promising new ones. We made sure that our selection covers the entire spectrum of the taxonomy displayed in Figure 2.2. The first method design criterion is the information used: schema information, associated instances, or both. This one is the most critical due to the practicalities that it entails. For example, teams within a company might only be allowed to share metadata with others due to data governance rules, making the schema-based methods the only option. The remaining criteria are related to the strategy or combination of strategies that each method employs and whether they use external knowledge (i.e., thesauri, lexica, ontologies, embeddings). As shown in Table 3.1, all methods we picked rely on one or more of them and are reasonably diverse from one another. In what follows, we describe the schema matching methods that we evaluate and compare in our benchmark. Furthermore, we explicitly report any modifications or decisions we made while attempting to reproduce the original algorithms.

Cupid Madhavan et al. [2001] is a schema-based approach. The first step is to translate the two schemata into tree structures representing the hierarchy of different elements, for example, relations or attributes. The overall similarity between two elements is the weighted sum of i) the linguistic and ii) structural similarities. The first calculates the name similarity for each pair of elements from the two schemata belonging to the same category. The structural similarity

Method	**Information used**		**Matching criterion**				**External Knowl.**
	Schema	**Instances**	**Syntax**	**Constraints**	**Context**	**Distribution**	
Cupid Madhavan et al. [2001]	✓		✓	✓	✓		✓
Similarity Flooding Melnik et al. [2002]	✓		✓		✓		
COMA Do and Rahm [2002]	✓	✓	✓	✓	✓		✓
Distibution based Zhang et al. [2011]		✓				✓	
SemProp Fernandez et al. [2018b]	✓	✓	✓				✓
EmbDI Cappuzzo et al. [2020]		✓					✓
Jaccard-Levenstein [Our baseline]		✓	✓				

Table 3.1: Selected methods and their characteristics

utilizes the tree transformations of the schemata to compute the similarity between elements based on their context. Cupid is not openly-available. Thus, in our implementation, we used WordNet[1] as thesaurus, while we rely on the name similarity formula to compute data compatibility scores.

COMA Madhavan et al. [2001] combines multiple schema-based matchers. Schemata are represented as rooted directed acyclic graphs, where the associated elements are graph nodes connected by edges of different types (e.g., containment). The match result is a set of element pairs and their corresponding similarity score. COMA also supports human feedback by allowing users to indicate the correctness of the resulting matches, which is considered in the next iterations, allegedly improving general accuracy. Engmann and Massmann [2007] extended COMA also incorporates two instance-based matchers, while COMA++ Aumueller et al. [2005] provided a graphical user interface, and Massmann et al. [2011] presented a new version of the system, addressing some issues of the previous versions. In our experiments, we use the COMA 3.0 Community Edition, where we use the default schema-based and instance-based strategies.

Similarity Flooding Melnik et al. [2002] is a schema-based matching approach that relies on graphs and outputs correspondence between any category of elements (relations, attributes, data types) of two given schemata. Specifically, the schemata are transformed to directed graphs, which have as nodes every element and as edges the relationships that these elements have with each other (e.g., a relation has an attribute, which is of a specific type). The graphs are then merged into a propagation graph, where pairs of nodes having similar connections collapse into map pairs. The algorithm's intuition is that each such map pair propagates its similarity to its neighbors, causing an update in their similarity score in an iterative manner until convergence. In our study, we have implemented from scratch the original method (since there exists only an outdated Java version of it from 2003), with the only difference that we use a string similarity of our own choice, i.e., Levenshtein distance Levenshtein [1966] since there are no details on the actual function that the authors used.

Distribution Based Zhang et al. [2011] is an instance-based method where relationships between different columns are captured by comparing the distribution of their respective data values. The method computes and refines clusters of relational attributes, using the EarthMover's Distance(EMD) between pairs of columns, a measure of distribution similarity of the corresponding instance sets. In the end, some disjoint clusters are given as output, wherein relational attributes are considered to be related. We implemented the original method (which was not openly available) without any modifications except for using another software to solve the integer programming problem in the last step of

[1] `https://wordnet.princeton.edu/`

the algorithm, which decides the final clusters (we used PuLP[2] instead of IBM CPLEX).

SemProp Fernandez et al. [2018b] tries to capture relationships between schema elements beyond syntactic similarity by making use of pre-trained word embeddings Mikolov et al. [2013]. SemProp first builds a semantic matcher that, given a domain-specific ontology, links attribute and table names to ontology classes using their em-bedding representation; then, it relates disparate attributes and tables by transitively following these links. Pairs of elements that fail to be related by the semantic matcher are forwarded to a syntactic one. Our experimental evaluation uses the open-sourced code for the Aurum Fernandez et al. [2018a] dataset discovery system, which includes the SemProp matcher.

EmbDI Cappuzzo et al. [2020] is a framework facilitating data integration tasks on relational data, by building relational embeddings. The authors propose a me-thod for embedding values and attribute names of relations by training them based on the input without using pre-trained embeddings. However, the method uses external knowledge, such as synonym dictionaries or pre-trained embeddings, to deal with more challenging cases. EmbDI is eligible for schema matching tasks, where it finds relationships between the columns of two datasets by comparing their corresponding embeddings. We integrated EmbDI in Valentine by importing the code[3] accompanying the original paper.

Jaccard Levenshtein Matcher. As a simple baseline, we implemented a naive instance-based matcher computing all pairwise column similarities using Jaccard similarity. We treat two values as being identical if their Levenshtein distance is below a given threshold. The method outputs a ranked list of column pairs, along with their respective similarity score.

3.2 Benchmark Architecture

In this section, we will go through the benchmarking framework shown in Figure 3.1 used in Valentine Koutras et al. [2020a] and the system that we deployed it in Figure 3.2.

The most common benchmark architecture is mainly comprised of three main components:

1. Different scenarios that address the entire problem space and, in our case, datasets that describe the diverse matching use cases shown in Section 1.1.

2. A runtime environment that, in this work, runs schema matching methods against a selected scenario.

3. A way to evaluate the results and compare the methods against each other.

Thus, we create three modules in our proposed framework, as shown in Figure 3.1, to cover the components mentioned above.

The ***DataLoader*** module fulfills the requirement of 1 by supporting the three different kinds of input data required by schema matching methods, and those are schemata information for the schema-based methods, data instances for the

[2] https://github.com/coin-or/pulp
[3] https://gitlab.eurecom.fr/cappuzzo/embdi

instance-based, or both. Currently, Valentine supports CSV files for the data instances and JSON files for the schema information and the golden standard, the format of which are presented in the Appendix sections A.1 and A.2 respectively.

The ***Methods*** module contains all the selected methods (Section 3.1) that are integrated in the benchmark and meets the requirement 2. All the methods have to implement the *get_matches* function that takes two DataLoaders, one for the source schema and one for the target, and returns a ranked list of matches based on the similarity between elements in descending order (matches with higher confidence on top).

The last module that covers point 3 is ***Evaluation*** and is firstly used to load the golden standard of the selected dataset, which contains the actual matches of the selected schema matching scenario. Then it takes the loaded golden standard and the ranked list of matches given as output from running the method to produce the requested metrics to evaluate the method.

To glue all the modules together, we defined the abstraction of a schema matching benchmarking job that takes as input a configuration file specifying which method to test, with what parameters, and on which dataset. A job's output is a JSON file containing the job's name, the ranked list of matches, and the resulting metrics. An example of a job's output is given in Appendix A.3.

Finally, due to the abstractions in the framework, it can be extended in the following ways:

- **Integrate new methods** in the benchmark, providing a complete overview of which method is the best. Moreover, it can help researchers in the field to develop new methods that improve upon the previous state of the art by showing them in which cases their method works or not.

- **Add more datasets** to incorporate a larger area of the schema matching problem space or different use cases from other domains. For example, the matching problems in Fintech may be different from those in healthcare.

- **Provide different metrics.** For example, since the framework outputs a ranked list of matches, the users might decide first to use a 1-1 match filter to get a conventional metric like precision, or they could use the entire ranked list to create a metric like the precision at n%. Different metrics would also serve various use cases, metrics after the 1-1 filter suit the fully automated use case. In contrast, metrics that use the entire list and measure the rankings' quality are better in a standalone recommendation scenario or help users verify matches in the semi-automated case.

Now, to run the benchmark and get the best results for each method, for each dataset, we performed a grid search with the values shown in Table 3.2. This parametrization was necessary due to the sensitivity of the method parameters. The parameters that are not included are set to their default values, as described in the respective papers.

We performed two different runs for the Distribution Based method Zhang et al. [2011]. The first is based on the original paper's recommended threshold values, and the second is to help the method find more matches in columns with low overlap. Additionally, we split the proposed single global threshold in two, one for each phase, for a more thorough search. For COMA Do and Rahm [2002],

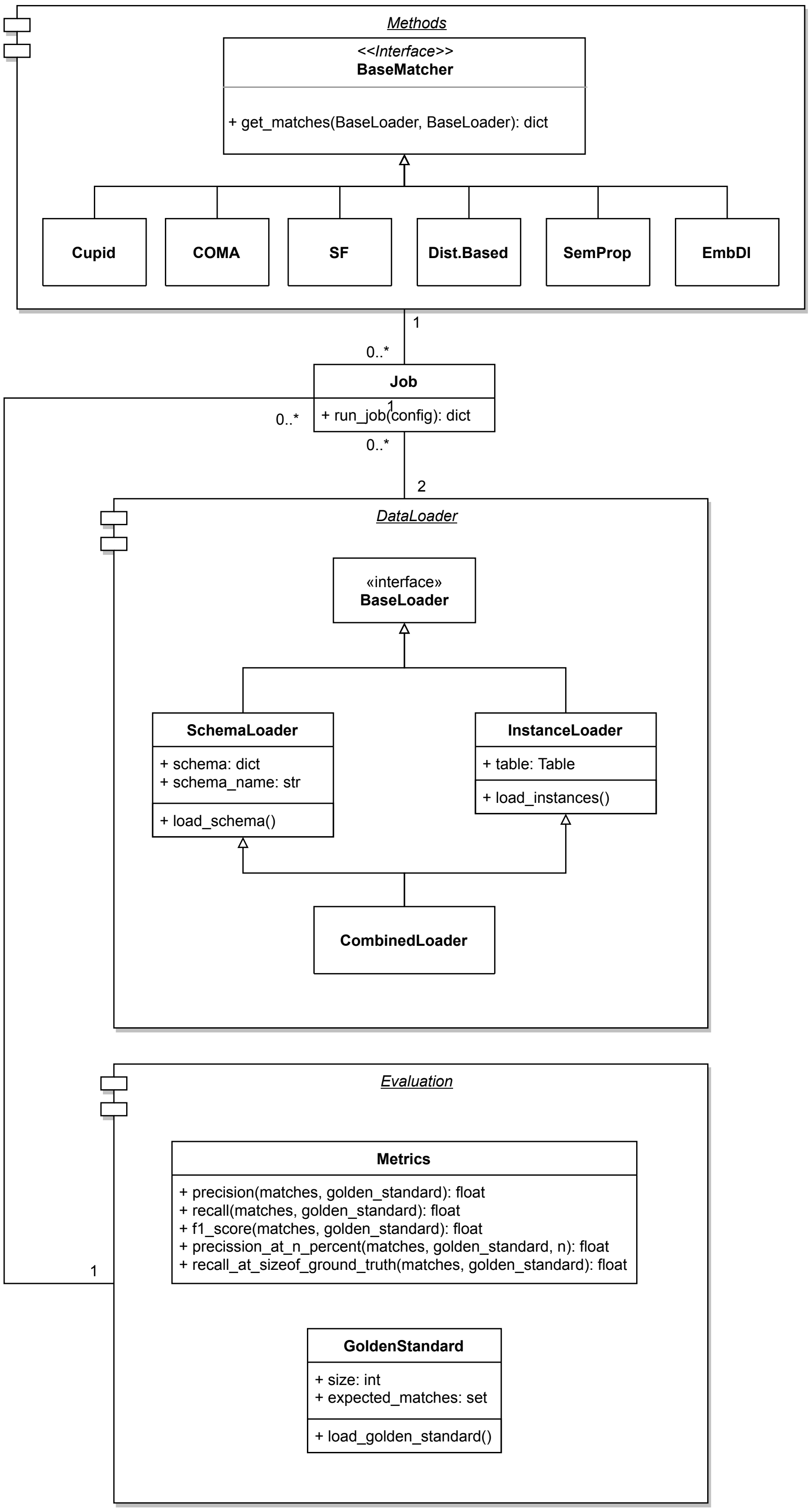
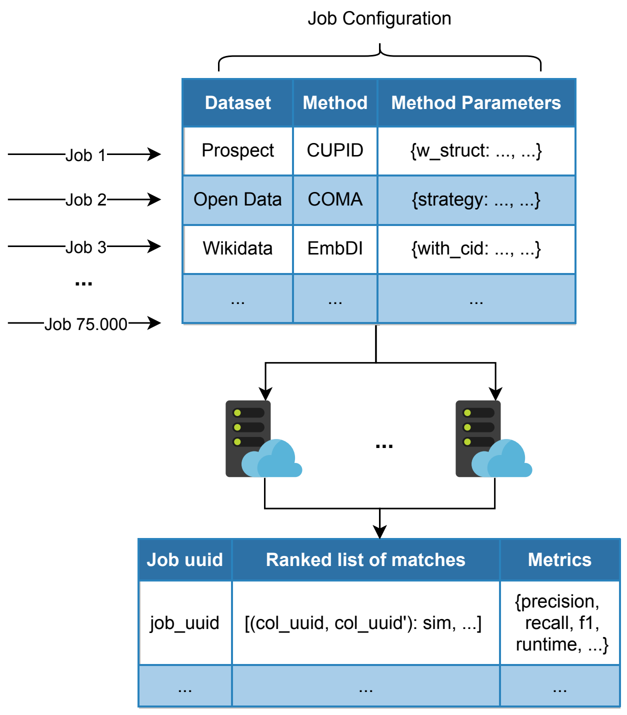


Figure 3.1: Simplified UML class diagram of the benchmarking framework



Dataset	Method	Method Parameters
Prospect	CUPID	{w_struct: ..., ...}
Open Data	COMA	{strategy: ..., ...}
Wikidata	EmbDI	{with_cid: ..., ...}
...	...	...

Job uuid	Ranked list of matches	Metrics
job_uuid	[(col_uuid, col_uuid'): sim, ...]	{precision, recall, f1, runtime, ...}
...	...	...

Figure 3.2: System that runs the benchmarking framework

we allow the output to include any found element pair, regardless of their similarity (i.e., we set the *accept similarity threshold* parameter to be 0). Finally, in Cupid Madhavan et al. [2001], we ran experiments with the weight of the structural similarity $w\_struct \leq 0.6$, since relational tables do not have the complex structure of XML schemata for which the method was designed.

An example of a method's configuration file is given in Appendix A.4. It starts with the type of the data loader (i.e., Instance, Schema, or Combined) followed by the method's parameters that can be provided in two ways, i) range for the numerical by specifying the max, min, and step and ii) values for the categorical parameters.

Following the parametrization, the problem's scale became apparent, with more than 75.000 schema matching jobs required to run all the methods against all the datasets. For example, the non-parallelized version of the Distribution Based method on the largest dataset, a small subset of the benchmark, would take around 60 days. In general, the average run time of all the methods, after parallelizing some of them Section 3.3, is around 3 minutes leading to a total runtime of more than 150 days.

This was the main reason behind our job oriented design, and with the help of GNU Parallel [4], we engineered a system that can distribute the created jobs to multiple machines and run them in parallel. A high-level overview of the system is shown in Figure 3.2, where a component at first creates the configuration files of all 75.000 jobs, then distributes them to run in parallel, and finally, gathers the output.

[4] https://www.gnu.org/software/parallel/

Method	Parameter	Values	Step
Cupid Madhavan et al. [2001]	leaf_w_struct	[0, 0.6]	0.2
	w_struct	[0, 0.6]	0.2
	th_accept	[0.3, 0.8]	0.1
Sim. Fl. Melnik et al. [2002]	prop.coeff.	inverse_average	-
	fix-point comp.	C	-
COMA Do and Rahm [2002]	strategy	[schema, inst.]	-
	threshold	0	-
Dist.#1 Zhang et al. [2011]	phase 1 θ	[0.1, 0.2]	0.05
	phase 2 θ	[0.1, 0.2]	0.05
Dist.#2 Zhang et al. [2011]	phase 1 θ	[0.3, 0.5]	0.1
	phase 2 θ	[0.3, 0.5]	0.1
SemProp Fernandez et al. [2018b]	minh.threshold	[0.2, 0.3]	0.1
	sem.threshold	[0.4, 0.6]	0.1
	coh.sem.threshold	[0.2, 0.4]	0.2
EmbDI Cappuzzo et al. [2020]	train.algorithm	word2vec	-
	sentence length	60	-
	window size	3	-
	n_dimensions	300	-
Jaccard Levenshtein (Our Baseline)	threshold	[0.4, 0.8]	0.1

Table 3.2: Parameterization of Methods

3.3 Scaling up the Instance-based Methods

As we previously discussed, the proposed benchmark is the most extensive to date, and to get timely results, we had to employ various scaling techniques. The job distribution that we performed worked well with the schema-based methods but, the instance-based methods, especially in the large datasets, remained slow. Consequently, this led us to parallelize the instance-based methods that we could.

For instance, COMA, EmbDI, and SemProp use multiple threads, and they do not have many hyperparameters to go through. Thus, there is no need to parallelize them even further. On the contrary, the Distribution based and Jaccard Levenshtein methods were not designed in a multi-threaded way, and they are each having more than double the hyperparameters of the others combined. Thus, in this section, we parallelize them and report the observed performance increase in Section 5.1.2.

Algorithm 1 Compute Distribution Clusters (**C**, θ)

```
1:  $G_D = \varnothing$
2:  for $i \leftarrow 1$ to $|\mathbf{C}|$ do
3:      for $j \leftarrow i+1$ to $|\mathbf{C}|$ do
4:          $e = EMD(C_i, C_j)$
5:          $A[C_i].insert(e, C_j)$          ▷ A is a hash table of lists of EMD/column-name pairs(e, C)
6:          $A[C_j].insert(e, C_i)$
7:      $G_D.AddNode(C_i)$
8:  for $i \leftarrow 1$ to $|\mathbf{C}|$ do
9:      $\phi_{C_i} = ComputeCutoffThreshold(A[C_i], \theta)$
10:     for all $C_j \in \mathbf{N}_{C_i}$ do
11:         $G_D.AddEdge(C_i, C_j)$
12: Return connected components of $G_D$
```

3.3.1 Distribution Based

At first, the Distribution Based method consists of two algorithms. Algorithm 1 takes as input the columns (**C**), and the global threshold (θ) finds the distribution clusters by first going through all the column combinations, calculating the EMD between each combination, and initializes the graph nodes with the columns.

Algorithm 2 Parallelized Compute Distribution Clusters ($\mathbf{C}$, θ)

```
1: $P = threadPool()$
2: $columnCombinations = generateCmb(\mathbf{C})$
3: $A = P.map(processEMD, columnCombinations)$
4: $edgesPerColumn = P.map(processCutoffThreshold, (A, \mathbf{C}, \theta))$
5: $G_D = createGraph(\mathbf{C}, edgesPerColumn)$
6: Return connected components of $G_D$
```

Then it calculates the cutoff threshold of each column with the rest and adds edges to the graph when the EMD is higher than the cutoff. Finally, it returns the connected components of the graph.

To scale this algorithm, we identified the parts that can run in parallel. At first, the logic described in lines 2-7 can be distributed to multiple threads by evenly partitioning the combinations across the threads to compute the EMD. Secondly, the cutoff threshold computation in lines 8 - 11 can again be distributed. A good thing about this approach is that no memory sharing is needed between the threads, removing the need for locks, achieving better scalability. The scaled version of this algorithm is given in Algorithm 2.

The second algorithm Algorithm 3 is used to compute the final attribute clusters taking the connected components (**DC**) from Algorithm 1 and the global threshold (θ). From lines 2-6 Algorithm 3 is almost identical to Algorithm 1 thus the same logic is applied while parallelizing it. The only difference is that instead of EMD, it has the EMD of the intersection of the two columns, and instead of a generic graph, it creates a graph of positive and negative edges. Finally, lines 7-17 do not have any significant computational cost leading us to leave them as is (in our experiments, the EMD computation takes almost 99% of the runtime). The scaled version of the algorithm is shown in Algorithm 3.

Algorithm 3 Compute Attributes (**DC**, θ)

```
1: $G_A = \varnothing, E[][] = 0, M[][] = 0$
2: for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
3:     for $j \leftarrow i + 1$ to $|\mathbf{DC}|$ do
4:         $e = EMD_{\cap}(C_i, C_j)$
5:         $I[C_i].insert(e, C_j)$    ▷ I is a hash table of lists of Intersection-EMD/column-name pairs(e, C)
6:         $I[C_j].insert(e, C_i)$
7:     $\phi_{C_i} = ComputeCutoffThreshold(I[C_i], \theta)$
8:     for all $C_j \in \mathbf{N}_{C_i}$ do
9:         $E[C_i][C_j] = 1$
10:    $G_A.AddNode(C_i)$
11: $M = E + E \times E$
12: for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
13:     for $j \leftarrow 1$ to $|\mathbf{DC}|$ do
14:         if $M[i][j] == 0$ then
15:             $G_A.AddNegativeEdge(C_i, C_j)$
16:         else
17:             $G_A.AddPositiveEdge(C_i, C_j)$
18: Return correlation clustering of $G_A$
```

Algorithm 4 Parallelized Compute Attributes (**DC**, θ)

```
1: $G_A = \varnothing, E[][] = 0, M[][] = 0$
2: $P = threadPool()$
3: $columnCombinations = generateCmb(\mathbf{DC})$
4: $I = P.map(processIntersectionEMD, columnCombinations)$
5: for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
6:     $\phi_{C_i} = ComputeCutoffThreshold(I[C_i], \theta)$
7:     for all $C_j \in \mathbf{N}_{C_i}$ do
8:         $E[C_i][C_j] = 1$
9:     $G_A.AddNode(C_i)$
10: $M = E + E \times E$
11: for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
12:     for $j \leftarrow 1$ to $|\mathbf{DC}|$ do
13:         if $M[i][j] == 0$ then
14:             $G_A.AddNegativeEdge(C_i, C_j)$
15:         else
16:             $G_A.AddPositiveEdge(C_i, C_j)$
17: Return correlation clustering of $G_A$
```

3.3.2 Jaccard Levenshtein

The second instance-based method that we parallelized is Jaccard Levenshtein (our Baseline), and since we created with scalability in mind, there exists only a parallel version shown in Algorithm 5. The method takes as input the two tables (source: **S** and target: **T**) and the threshold (θ), which bounds the Levenshtein ratio between the data entries of the two columns. Then the method generates the column combinations and calculates the Jaccard Levenshtein similarities in parallel.

Algorithm 5 Jaccard Levenshtein(**S**, **T**, θ)

```
1: $P = threadPool()$
2: $columnCombinations = generateCmb(\mathbf{S}, \mathbf{T})$
3: $similarities = P.map(processJaccardLeven, (columnCombinations, \theta))$
4: Return $similarities$
```

3.4 Datasets

The primary contribution of this work is the benchmarking system that is the component of Valentine Koutras et al. [2020b]. In addition to that, another significant contribution of Valentine is the dataset creation process, and since it plays such an integral part in the benchmark, we will go through it briefly in this section.

One of the biggest challenges in evaluating schema matching methods identified in Valentine is the lack of openly available datasets with schema matching ground truth. The strategies that we employed to solve this issue are two:

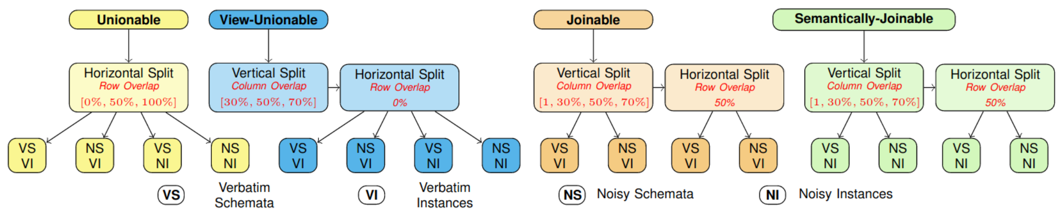


Figure 3.3: Fabrication of datasets with respect to each problem as detailed in Koutras et al. [2020b]

1. Split existing datasets horizontally to fabricate unionable dataset pairs and vertically to fabricate joinable dataset pairs, where the ground truth lies with the original table. This approach is used in recent works like Nargesian et al. [2018]; Zhu et al. [2019]. In Valentine, we extend eTuner Lee et al. [2007] that serves the previously mentioned purpose with the addition of giving us the ability to make the schema matching cases harder. It achieves that by injecting noise to both schema information (i.e., prefix column names with the table name, abbreviate the column names or drop their vowels) and data instance (i.e., keyboard proximity typos for string and distribution shifts for numeric columns) levels.

2. Curate existing datasets by determining the ground truth manually as in Zhang et al. [2011].

Finally, we split tables horizontally to create unionable pairs, vertically to create joinable pairs, and in both ways (joinable and unionable), following Nargesian et al. [2018]; Lee et al. [2007]. Figure 3.3 shows the dataset fabrication process for the four schema matching scenarios detailed in Section 1.1. All scenarios require a certain amount of actual column matches between the two datasets, i.e., *Unionable* requires all columns, *View-Unionable* a common subset of columns, and *(Semantically)-Joinable* one common column. Then, for all cases, we horizontally partition the table with varying percentages of row overlap and vertically partition all but the Unionable case. Finally, as mentioned above, all created datasets might contain verbatim schemata or noisy ones, as well as verbatim or noisy instances.

The datasets that we selected bear distinct characteristics such that they challenge all methods, coming from very diverse fields. There are two dataset categories following the two different strategies mentioned above.

The first category contains existing public datasets that go through the data fabrication process, as shown in Figure 3.3, giving more than 500 synthetic datasets. These datasets are:

The **Prospect** data of TPC-DI Poess et al. [2014] which is TPC's[5]benchmark for data integration task and the table contains hypothetical customer prospect data.

An Open Data dataset consisting of tables from Canada, USA, and the UK provided and used by the authors of Nargesian et al. [2018].

The Assays table of the open chemical database ChEMBL[6] that is closely related to the EFO[7] ontology making it one of the few datasets that come with an

[5]http://tpc.org/
[6]https://www.ebi.ac.uk/chembl/
[7]https://www.ebi.ac.uk/efo/

ontology.

The second category features real-world datasets with an inherent schema matching challenge that we curated to create the ground truth for them manually, and are the following:

We created a dataset containing US musician data from Wikidata's[8] knowledge base. The difference between the two tables that it contains is that they are represented with slightly varying schemata and instance encodings.

Magellan's Das et al. [2015] 7 dataset pairs that represent real-world data used in Cappuzzo et al. [2020] to evaluate their novel proposed schema matching method, curated mainly for Entity Matching techniques.

The final datasets that we used in Valentine to check whether the methods could perform well on the industry's use cases come from our partner ING Bank Netherlands. We were granted access to two production datasets that contain information about their SCRUM system (ING#1) and a catalog describing software applications used internally (ING#2). These two datasets cannot be made public due to privacy concerns.

Finally, in table Table 3.3, we detail the statistics of each dataset.

	Datasets						
	Synthetic			**Curated real-world**			
	TPC DI's Prospect	**Open Data**	**ChEMBL**	**Wikidata**	**Magellan**	**ING#1**	**ING#2**
# of rows	[7492-14983]	[11628-23255]	[7500-15000]	[5423-10846]	[864-131099]	937/972	1000/1000
# of columns	[11-22]	[26-51]	[12-23]	[13-20]	[3-7]	33/16	59/25
# of pairs	180	180	180	4	7	1	1

Table 3.3: Dataset statistics

3.5 Evaluation

In this section, we will go through the metrics supported in the benchmark that measure the schema matching methods' performance. These metrics can be categorized into the following two categories:

1. Classic metrics like precision, recall, and f1-score
2. Metrics that evaluate the result as a ranked list

In the first category, a one-to-one match filter needs to be applied first to get meaningful results. The reason being that many methods give as an output every possible match with varying degrees of similarity, leading to very low precision and maximum recall. In this work, the filter has the following logic: At first, it gets the median similarity of the set of the values and removes all matches that have a similarity lower than that. Then from what remained, it matches columns from the highest similarity to the lowest until the columns have, at most, one match. However, this category of metrics cannot represent the problem well since they cannot work for the 1-to-n or the n-to-m matching scenarios due to the one-to-one filter limitation.

Thus, we decided to treat the problem as a ranking problem and only focus on the second category's metrics. The reason behind that was the limitations mentioned earlier and the fact that a ranking allows users to explore and decide

[8] https://www.wikidata.org

on match candidates more efficiently. The two metrics that we selected for this are the following:

Recall@k. Measures the number of relevant matches in the top-k match pairs of the resulting ranked list where $k = |ground\_truth|$:

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

Precission@n%. Measures how many out of the top-n% items are relevant in a similar fashion as Recallk where $n \in (0, 100]$.

$$Precission@n\% = \frac{\#\ of\ top\text{-}n\%\ relevant\ matches}{\#\ of\ (top\text{-}n\%\ relevant\ matches + top\text{-}n\%\ non - relevant\ matches)}$$

Furthermore, we would like to mention that the most prominent metrics used in evaluating ranked lists from the information retrieval literature, i.e., NDCG, MAP, or MRR, cannot be used to measure the effectiveness of schema matching methods because there is no correct order in the matches. We only want the methods to produce the correct matches at the top of the list in any arbitrary order. Finally, in the experiments' section Chapter 5, we will only report Recall@k, which is essentially equivalent to Precision@n% but stricter in the sense that the perfect score is given if only all the actual matches are on top of the ranked list. On the contrary, Precision@n% could also achieve a perfect score if, for example, there is just one correct match out of the 10 top matches and n equals a small number considering only the first match from the ranked list, making it a very relaxed metric.

Chapter 4

Scalable Holistic Schema Matching System

This chapter showcases the details of our second contribution, a scalable semi-automated schema matching system. In Section 4.1, we aim to answer our third research question, "What should be the design of an elastic, scalable, semi-automated, holistic schema matching system?" by showcasing the design choices and architecture of our proposed system. In the chapters that follow, Section 4.2 shows the currently supported data sources and the abstractions that we devised so that more could be integrated in the future. Section 4.3 goes through the REST API of our schema matching microservice. Section 4.4 details why we chose a task queue approach to offload the match computation and which schema matching methods were selected after the benchmark. Furthermore, in Section 4.5, we explain which databases we used for the system and for what reason. Finally, in Section 4.6, we answer our fourth and final research question "How to present the proposed matches to multiple users, increasing the speed of match verification?" where we present the user interface of the proposed system and how it improved upon the state of the art systems.

4.1 System Architecture

To start with the system architecture, we first have to define its design specifications. This work's primary focus is **scalability**, which means the need to meet the demand for schema matching jobs without any significant runtime performance degradation while using more machines to handle the workload. Additionally, it is essential to consider the **elasticity** of such a system, and by that, we mean the ability to change the number of machines in the system to meet the demand dynamically without any downtime. For example, if we have many pending jobs, we can increase workers' numbers to achieve speedy results. On the contrary, when there is a small number of incoming jobs, we can downsize the worker pool to save on the system's expenses.

The next requirement is that we need the system to perform **holistic schema matching**, a field relatively unexplored by the literature mainly due to its scalability implications, as we mentioned in Section 1.1.2, its engineering requirements (i.e., integrate multiple data sources to the matching system) and a way to show these results to the users. Following that, we come to our next requirement: the system needs to be semi-automated. The primary reason for that is the difficulty of the schema matching problem (i.e., tagged by pioneers in the field as

AI-complete). Thus, no method could produce 100% precision in every schema matching use case. If we accepted the results in an automated way, the errors could propagate to the other data integration systems (e.g., mediated schema, knowledge base, or uniform query), making user feedback essential. However, we need to make the match verification task scalable to multiple people (that could even be non-experts in the domain), fast, and straight forward. Otherwise, it could become a very tedious task for domain experts or database administrators and falls to neglect. The last but equally important specification is that the system needs to be **expandable** in terms of both new schema matching method integration and data source support to match the new developments in the field.

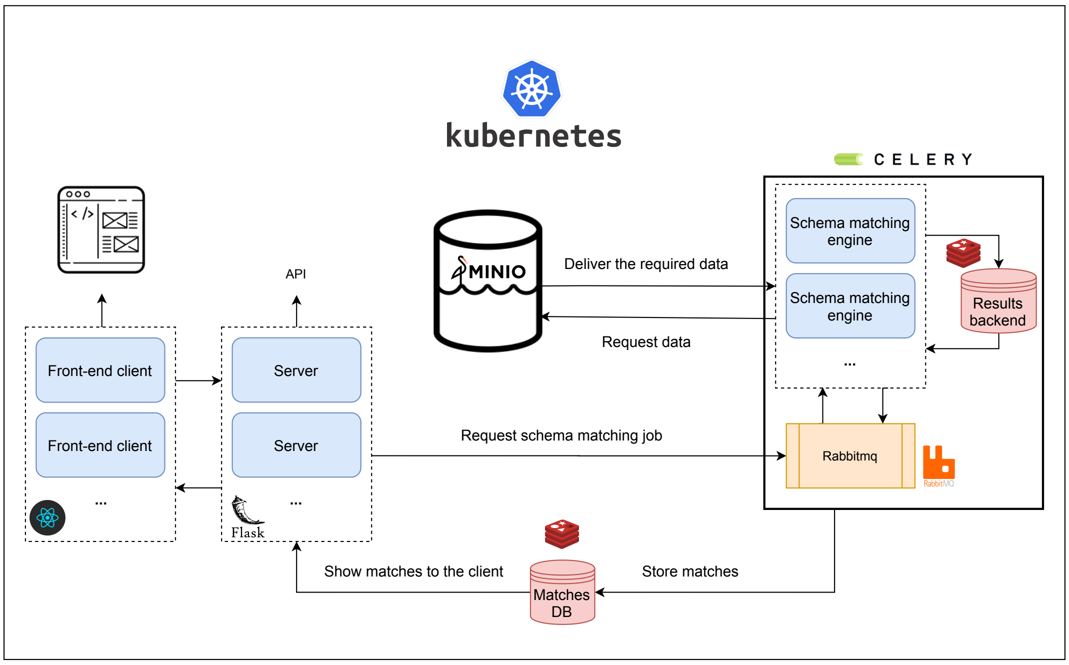


Figure 4.1: Schema Matching system kubernetes deployment

To fulfill the first two requirements, we decided to go with a container-based solution managed by Kubernetes[1], which is an open-source container orchestration system initially developed by Google. Kubernetes fulfills the requirements by providing easy-to-figure service discovery with load balancing and automated rollouts and rollbacks of deployment configurations allowing for seamless scalability and autoscaling. Thus, every component of our system needs to be containerized.

In Figure 4.1, we present our system's deployment on Kubernetes. It consists of four separate scalable components, a front-end client created with React[2] that provides the users with the UI of our system that is detailed in Section 4.6.

The next component is a Flask[3] microservice that acts as both the backend of the client and provides an independent REST API described in Section 4.3.

[1]https://kubernetes.io/
[2]https://reactjs.org/
[3]https://flask.palletsprojects.com

Now, the system offloads the actual computation of the schema matching Jobs to a Celery[4] task queue that is comprised of three parts, a message queue that receives the job information we decided to select RabbitMQ[5] because it offers better guarantees and lower latency compared to the other supported Celery message brokers. Additionally, it requires a results backend, i.e., a database that stores the intermediate results of a schema matching job that will later be combined to produce the output. Finally, worker containers that perform the schema matching logic. This part of the system is detailed further in Section 4.4.

The remaining components are a persistent No-SQL Redis[6] database that maintains the schema matching job results and the verified matches (described in Section 4.5), and the data sources. In Figure 4.1, we display the Minio object store as one of the data sources that our system supports. More information about the data source design that fulfills the expandability requirement is provided in Section 4.2.

4.2 Data Sources

For our system to hold the title of holistic schema matching, it needs to incorporate all the data sources that belong to the company or institution that uses it. Thus, in this section, we first showcase in Figure 4.2 our data source module's abstractions that allow for data or schema extraction feeding the schema matching methods.

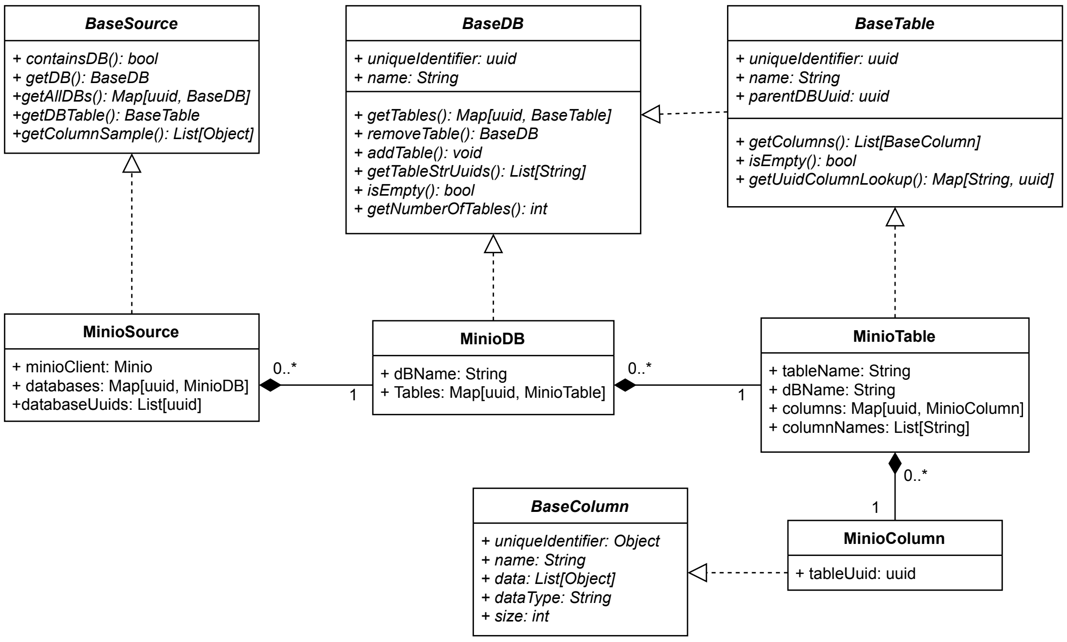


Figure 4.2: UML class diagram of our data sources module

This logic resides within the Celery worker nodes of our task queue. The top-level abstraction is the data source itself called *BaseSource* (i.e., a Minio cluster, a PostgreSQL server, an Apache atlas server, and so on) supporting lookup and

[4]https://docs.celeryproject.org
[5]https://www.rabbitmq.com/
[6]https://redis.io/

retrieval operations for the databases that lie within, their tables and column samples of said tables. The one following is the *BaseDB* describing a database within the source (i.e., a Minio bucket, a PostgresSQL database, e.t.c.) providing functions for table handling within the database. The most interesting one is the *BaseTable* because it inherits the *BaseDB* the use case for that is that a table can also be modeled like a single table database giving the users more flexibility while using the API. This abstraction supports functions related to column retrieval. The final one is the *BaseColumn* that holds the column information. In Figure 4.2, we also give an example of the Minio data source, extending those abstract classes.

We will go briefly through the data sources currently supported in this work in the sections that follow.

4.2.1 Minio

The first source incorporated into the system is Minio, an open-source distributed object store with an API compatible with Amazon's S3. The main reason we chose Minio is its resemblance to S3, the foundation of Amazon's data lake enterprise solutions, providing a realistic use case for our system. The way we mapped Minio to our data source abstractions is the following: The data source is the Minio server, the databases are the Minio buckets, and the data inside the tables (currently, only CSV and XLSX files are supported).

The Minio architecture is shown in Figure 4.3, and it provides the following features: i) Uses erasure coding for data protection ii)It is highly available iii) Follows strict read after right.

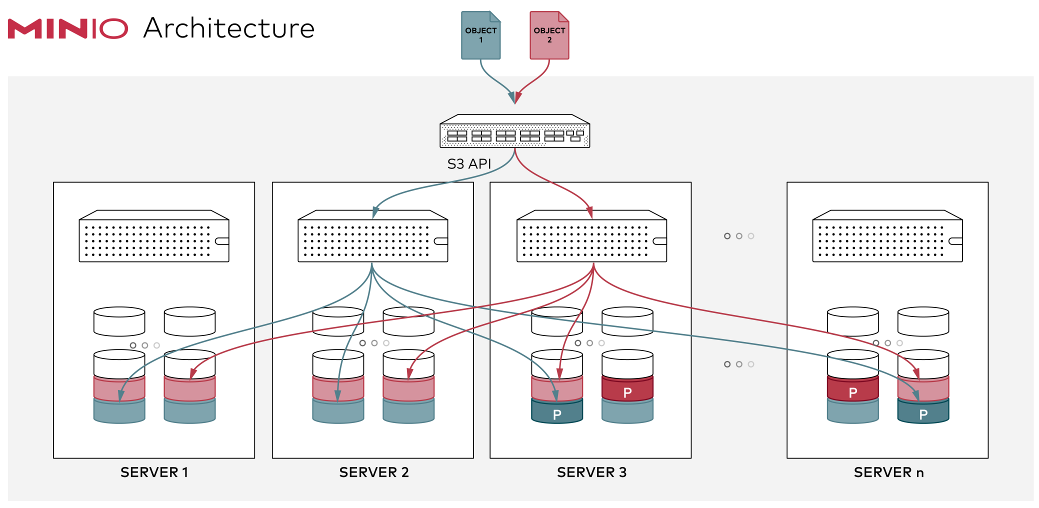


Figure 4.3: Minio's architecture as shown in the project's official page

4.2.2 Apache Atlas

Apache Atlas is the go-to open-source data governance system with one part that we are mostly interested in, its metadata management capabilities. We connect Atlas to our system by requesting the metadata information via its REST API and identifying the kind of information via the response because it returns entities that can represent databases, tables, columns indexes, e.t.c. However, the problem with that is that we need to know the custom classifications that the administrators have used for their data sources internally, making a very tedious integration to the system. Atlas's positive thing is that it is very scalable with the JanusGraph graph database as a backend.

4.2.3 Local Filesystem

The final data source supported is mainly for development and testing purposes and is the local filesystem. This data source is handy because the developer that creates a schema matching method would not want to containerize the method and deploy a data source during the development and testing phases. Additionally, we can create test suites for all the methods with this data source without any additional infrastructure. The way it is structured is to have a root folder, which is the data source that contains subfolders (the databases) and CSV files as the tables. Finally, it supports both Windows and UNIX based systems.

4.3 Schema Matching API

In this section, we will go through the API specification of our Flask microservice that its function is to serve the following purposes:

1. Dispatch schema matching jobs to our Celery distributed task queue
2. Return the match results to the front-end when they arrive
3. Allow the user to verify/discard matches or delete a job's output that is not useful anymore or
4. List the available data sources with their databases and tables
5. Provide with column sample data that help with the match verification (when the data source supports that)

In the following sections, we will go through the endpoints in more detail.

4.3.1 Schema Matching

At first, the primary endpoint that the front end uses for the systems matching purposes is the ***/matches/submit_batch_job*** and supports the following logic:

The users will select the tables that they are interested in finding matches (source tables), the tables that they will comprise the search space (target tables), and the schema matching methods (with their parameters) that will try to find these matches these three comprise of the input information of the endpoint. The service then finds all the unique table combinations, submits a job to the task queue for each schema matching method, and returns its universally unique identifier to the users. However, since we are dealing with tabular data, it is safe to assume that we can partition the submitted job into sub-tasks that run in parallel because of the structural similarity within the database, that some methods like Cupid Madhavan et al. [2001] use, would always be the same. Thus, these sub-tasks comprise a single source/target table combination providing only a part of the proposed matches, which is stored in a cache, and when they all finish, they are combined, sorted from the match with the highest similarity to the lowest and finally stored into a persistent No-SQL Redis database awaiting verification.

The API also provides some other specific endpoints that can be used to be the users. They are six, three for Atlas, and three for Minio. The input they receive is which schema matching method to use, its parameters, the source table, and some configuration information about the data sources. The logic that the endpoints follow is that they run the method at different scopes. The first searches in the entire search space (i.e., holistic schema matching). The second takes an additional input argument from the user specifying which database to search for matches, limiting the search space. The last one searches for matches within the same database, trying to find the functional dependencies. All the endpoints work asynchronously, and thus, they return a UUID of the job submitted to the task queue, which the user can later poll to get the ranked list of matches.

4.3.2 Results and Auxiliary

The server's API also supports endpoints related to the functions that surround the schema matching process. A group revolves around managing the schema matching jobs like retrieval of finished jobs, their details, and proposed matches. Another group around the match verification process, verifying or discarding a proposed match that a job proposed, the retrieval of the verified matches, and when a user finishes inspecting a job about verified matches, the ability for job deletion. Finally, it also provides endpoints that list information about the connected data sources, in the case of Minio, the buckets with their contents, and column samples from a selected dataset. The retrieval of column samples comes in hand during the match verification process, where the users can take a peek inside the data to get a better feel whether the two suggested columns are actual matches.

4.4 Task Queue

As we identified in Section 2.3, this approach has been explored previously in the ontology matching literature in works like GOMMA Gross et al. [2010], and SPHeRe Amin et al. [2016] where they use a job queue to distribute the tasks across multiple computing nodes, and it is proven to scale well. However, in these works, an HPC cluster has been used, something that not every company or institution can access, while our cloud-native Kubernetes approach is a lot easier to acquire. Additionally, this has not been done before to address the schema matching problem. That might be due to the engineering complexity of integrating various data sources and different methods in such a system.

The task queue component that forms the backbone of our system functions in the following way: At first, schema matching job requests are received in the queues message broker detailing the job's parameters. We use RabbitMQ for this work since it is very mature within the Celery ecosystem and is proven to be faster and more robust than its counterparts.

The first design decision that we made was on which task queue framework to use. We opted for Celery because it allowed for straight forward Job partition to multiple subtasks executed in parallel. This partitioning was useful because we could split a schema matching job consisting of multiple table combinations to individual tasks. To achieve that, Celery requires a results backend to store the intermediate results before merging them later. We decided to use Redis for this purpose mainly due to its speed as an in-memory cache. Finally, when all the results are merged and sorted to form the ranked list of matches, the task queue saves them into a persistent external database, ready for inspection.

In the next section, we will go through the supported schema matching methods of the proposed task queue.

4.4.1 Schema Matching Engine

The schema matching engine that resides in every worker node's core in our task queue consists of the methods that made it through the benchmarking process described in Chapter 3 based on their performance presented in Chapter 5.

The methods that made it through are:

COMA Do and Rahm [2002] as its instance-based variant scored the highest on average among all other methods in every matching scenario. Additionally, it is reasonably fast compared to the other instance-based methods by coming second to the Distribution Based method.

CUPID Madhavan et al. [2001]. Although CUPID did not score among the highest in the schema-based category, we decided to include it mostly because the reason for that was the lack of user-defined external knowledge that it requires, impairing its performance, and in a realistic deployment scenario, such knowledge would not be too troublesome to acquire. Furthermore, after the benchmark, we improved our implementation of CUPID, wherein the linguistic similarity part, we use Levenshtein distance between the elements instead of n-gram, observing a substantial improvement in accuracy.

Distribution Based Zhang et al. [2011] might have scored second in our synthetic benchmark. It is the fastest instance-based method (after our parallelization presented in Section 3.3) and scored the highest in our industry based scenario. Moreover, COMA's issue with n-m matches discover in our benchmark makes this the more robust instance-based method.

Similarity Flooding Melnik et al. [2002] was the highest performing schema-based method in our synthetic benchmark and second fastest while working on a single thread while COMA's schema-only variant that is the fastest uses as many as are available.

Jaccard Levenshtein [Our Baseline] scored third in terms of performance of the instance-based methods and was considerably slower than the rest. However, as we previously mentioned, it is less than 100 lines of code. Thus, it could be very quickly be improved with a different similarity metric between the elements and approximation techniques that in works like Zhu et al. [2016]; Flajolet et al. [2007] has proven to increase the runtime performance.

The methods that did not make it to the benchmark are:

SemProp Fernandez et al. [2018b] was not integrated into the system due to its requirement for external services (i.e., elastic search) and an ontology related to the schema matching task, which is not always available. Furthermore, its performance on both accuracy and runtime was among the lowest. Thus, the only option that remained was to exclude it.

EmbDI Cappuzzo et al. [2020]. The main reason for EmbDI's exclusion was the mediocre results that could not beat the schema-based methods in conjunction with a runtime of three orders of magnitude greater, while also running on multiple threads during the match detection phase.

4.5 Databases

To meet the system's needs for caching and persistent storage, we employ two databases. The first one is a Redis used by Celery as an in-memory cache to store the intermediate task results (i.e., table-to-table matches) that are later combined and inserted into the next database of the system that is responsible for the match results. This database is set to contain the ranked lists of matches, JSON files keyed by the UUID of the corresponding schema matching job. That fact made us chose a No-SQL document store, and we chose Redis due to its speed, persistence options that it provides, and ease of use. The persistence options that it provides are point-in-time snapshots at specified intervals and logging every write operation so that if a failure happens, it can load everything on startup.

This database fulfills two more requirements. The first one is to maintain the chronological order of the submitted matches. Due to the way No-SQL databases maintain their records in different partitions, there is no way to tell which came first, something coming out of the box in traditional databases. To solve this, we created a Redis list functioning like a queue maintaining the order this way, avoiding the need to create a large document with timestamps that would need to be accessed simultaneously by multiple tasks. The final requirement is to store the verified matches persistently, and we solve it by creating a different namespace inserting them there.

4.6 Expert Evaluation

This section goes through our system's UI design that focuses mostly on practicality and scalability rather than looks. At first, we want out UI to serve the following purposes: Provide an interface such that the users can perform holistic schema matching. To do that, the UI should list all the databases and tables from the connected data sources, allowing the users to select which tables to run against which. To clarify this, the users will select some source tables or databases (those that they want to find their matches) and some target tables or databases (those that will form the search space). The next action that this UI needs to facilitate is the method selection by displaying the integrated methods and allow for their parameter configuration. After these two actions are

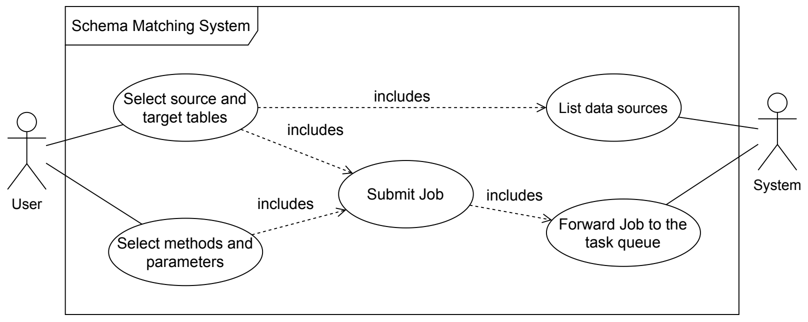


Figure 4.4: The use case diagram of our UI's schema matching page

performed, a schema matching job is formed and ready to be submitted for computation to our task queue. The UML use case diagram illustrating this logic is provided in Figure 4.4, and the resulting UI page of our system is given in Figure 4.5.

When the task queue finishes with the schema matching job, the UI needs to fulfill the semi-automated requirement by enabling the user to verify or discard the recommended matches. As we investigated in the related work Section 2.4, there is a shift towards formulating this as a microtask. Thus, we took this into account. This UI component displays the proposed matches by showing the columns matched in the format "tableName.ColumnName" for both source and target columns. The similarity is displayed with a gradient color bar moving from red to green as the similarity increases. We made that choice to make it easier to compare the different proposed match similarities.

Additionally, to not overwhelm the users, we paginated the resulting proposed matches, and since they are sorted by similarity with the highest on top, all the results with the highest probability of being matches are shown first. Moreover, to give more information about the two matching columns to the users when they are not sure whether they should verify a match or not, we give the option to click on the proposed match, and then the UI displays sample data out of both

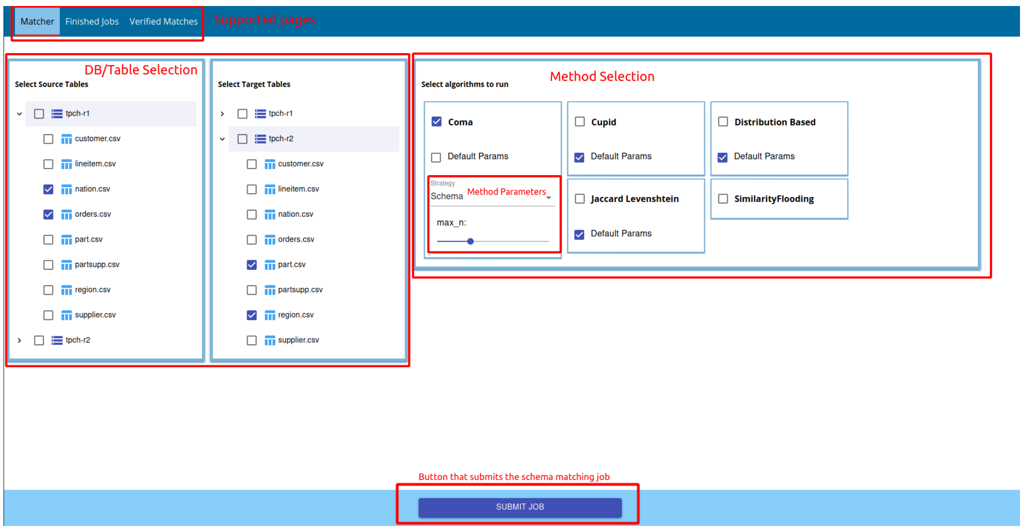


Figure 4.5: The resulting page in the UI of our system after facilitating the use case of Figure 4.4

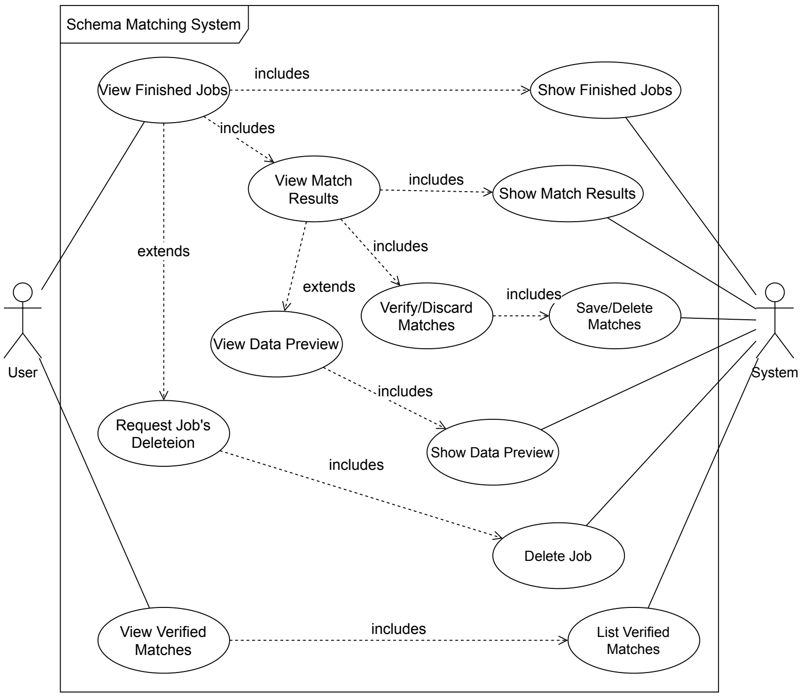
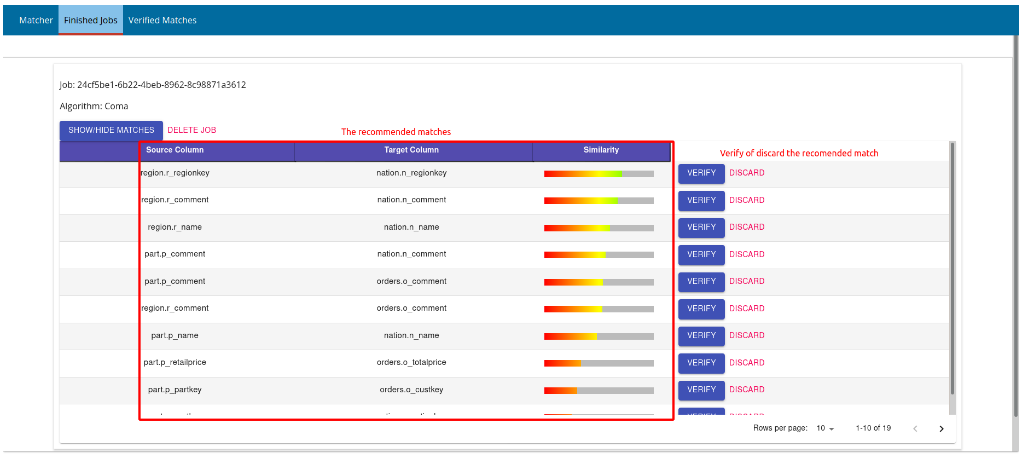


Figure 4.6: The use case diagram of our UI's match verification page

columns as shown in Figure B.1. Finally, to make this scale, we allow multiple users to view this page, verifying matches in parallel.

The UI's final page produces us the verified matches page displayed in Figure B.2, which shows the detailed information about all the verified matches. Namely, the database and table that they reside in with their universally unique identifiers can later be used to create a knowledge base, a mediated schema, or even give information to data scientists looking for related datasets.

The UML use case diagram illustrating this logic is provided in Figure 4.6, and the resulting UI page of our system is given in Figure 4.7.



Job: 24cf5be1-6b22-4beb-8962-8c98871a3612

Algorithm: Coma

Source Column	Target Column	Similarity	
region.r_regionkey	nation.n_regionkey		VERIFY DISCARD
region.r_comment	nation.n_comment		VERIFY DISCARD
region.r_name	nation.n_name		VERIFY DISCARD
part.p_comment	nation.n_comment		VERIFY DISCARD
part.p_comment	orders.o_comment		VERIFY DISCARD
region.r_comment	orders.o_comment		VERIFY DISCARD
part.p_name	nation.n_name		VERIFY DISCARD
part.p_retailprice	orders.o_totalprice		VERIFY DISCARD
part.p_partkey	orders.o_custkey		VERIFY DISCARD

Figure 4.7: The resulting page in the UI of our system after facilitating the use case of Figure 4.6

Chapter 5

Experiments & Evaluation

In this chapter, we go through the experimental evaluation of this work, split into two parts. In Section 5.1, we detail the evaluation process for the benchmarking system of Chapter 3. Following that, in Section 5.2, we evaluate the scalability and efficiency of the proposed holistic schema matching system of Chapter 4. The developed system's source code is openly available on Github.

5.1 Schema Matching Benchmark

In this section, we present the work done for the schema matching benchmark described in Chapter 3. Its evaluation consists of two subsections. Section 5.1.1 showcases the accuracy results produced in Valentine Koutras et al. [2020b] by running the selected methods described in Section 3.1 against the created datasets defined in Section 3.4. Section 5.1.2 displays the runtime efficiency evaluation for the schema matching methods, the benchmarking system's scalability, and the parallelized instance-based methods' efficiency.

5.1.1 Effectiveness results

To measure the performance of the selected schema matching methods of Valentine Koutras et al. [2020b] detailed in Section 3.4 we performed an extensive evaluation as explained in Section 3.2 with more than 75.000 benchmarking jobs that is, to the best of our knowledge, the most extensive to date for such a task. In all the effectiveness measurements that follow, we use $Recall@|GroundTruth|$ as detailed and explained in Section 3.5 mainly because we want to treat the methods' output as a ranked list and thus, evaluate it as such. Finally, we report the aggregated results over all the datasets (540 in total) and configuration pairs in all graphs that follow.

At first, in Figure 5.1, the schema-based methods' accuracy results are displayed against our synthetic/fabricated datasets. These methods are Cupid, Similarity Flooding, and the schema-based variant of COMA. We decided to display only the noisy schemata in our evaluation since all algorithms had perfect scores if the column names were the same. Furthermore, we see that the results we get for both joinable scenarios are almost identical since schema-based methods ignore the noise in instances, which separates the two scenarios. The small differences we observe are due to the stochastic nature of our fabrication process. In the conducted test, none of the schema-based methods managed to get higher $Recall@|GroundTruth|$ than 0.65. On average, Similarity Flooding performed better than the rest in every scenario, with the schema-based version of

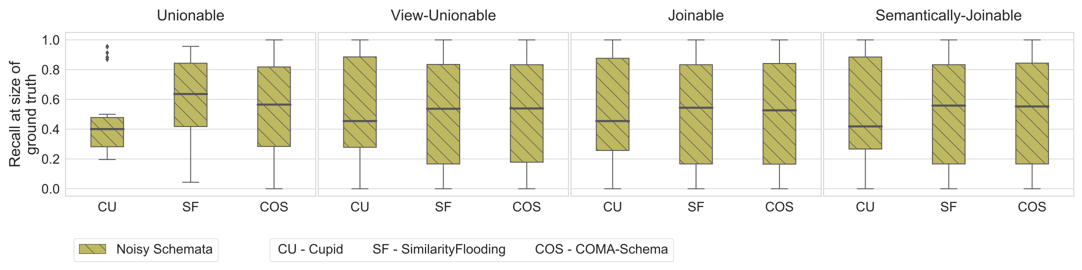


Figure 5.1: Recall at ground truth results of the schema-based matching methods for our fabricated datasets

COMA coming in second but very close. Cupid comes last by more than 0.5 in every scenario. Additionally, since the data are tabular and do not have a detailed structure that the methods can exploit, the scenarios are even harder for CUPID and Similarity Flooding. Consequently, this test proves that with the devised variations in the column names, schema-only methods struggle to perform well with our evaluation metric.

Naturally, the next experiment focuses on the selected instance-based methods that use only the rows of each dataset. These methods are the Distribution Based, the instance-based variant of COMA, and our baseline Jaccard Levenshtein. In Figure 5.2, we observe that, on average, the instance-based methods outperform the previously evaluated schema-based in all cases except the semantic-joinable, which is expected due to the induced changes in the data. In the joinable case with verbatim instances, both the Distribution Based and COMA get close to perfect scores, with Jaccard Levenshtein close behind. Furthermore, in both unionable cases, COMA seems to outperform in both noisy and verbatim instance cases. Also, the view unionable case seems harder for all methods because the datasets share less common values. Finally, an interesting outcome is that our simple baseline comes close to the other two sophisticated methods beating the Distribution Based in the view unionable case and staying close to COMA when no noise is applied to the data.

The final method category that we experimented with our fabricated datasets is the hybrid methods that utilize both schema and instance-level information, EmbDI, and SemProp. We have to note that for SemProp, we use only the ChEMBL dataset due to the methods requirement on an external ontology while EmbDI runs on all. In Figure 5.3, we observe that SemProp performed poorly, being the worst out of all the methods so far, in all schema matching scenarios. Therefore, making it hard to recommend. On the contrary, in the joinable and

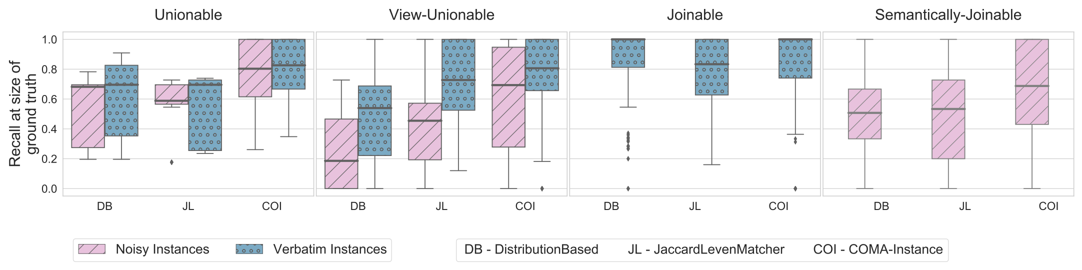


Figure 5.2: Recall at ground truth results of the instance-based matching methods for our fabricated datasets

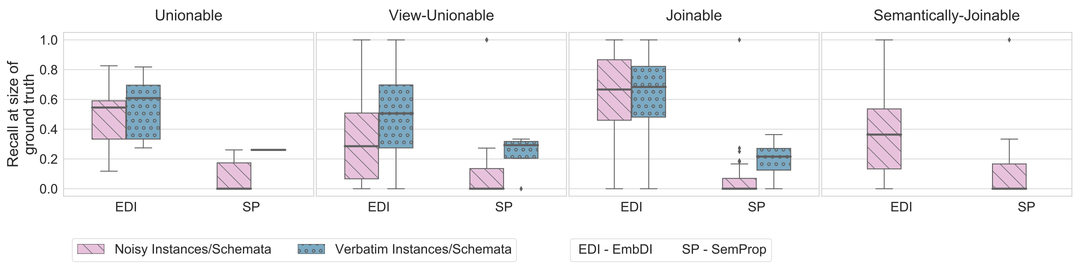


Figure 5.3: Recall at ground truth results of the hybrid matching methods for our fabricated datasets

unionable cases, EmbDI performed better than the schema-based methods but worse than the instance-based. Additionally, in the view-unionable case, it performed similarly to the Distribution Based method. However, we expected both methods to perform better or at least equally well to the rest since they use both instances and schema information, but we were proven wrong. For Semprop, the reason for this behavior could be that its pre-trained word embeddings are not representative of such a specialized case of a chemical dataset. For EmbDI, we believe that the randomness in the training data generation, as well as its dependence on overlapping values, result in a low recall.

In the final part of the effectiveness results, we ran the methods against the real-world datasets that we have previously, manually curated their ground truth. Figure 5.4 shows the results for the Wikidata dataset where again, the four instance-based methods have better performance than the schema-based ones in the joinable and unionable scenarios. This makes sense since we have not tampered with the data instances in any way, but the datasets' schema elements are quite different. An expected finding is that in the cases where the distributions vary a lot, for example, the view-unionable and the semantically joinable, the Distribution Based performs worse than the other instance-based methods. On the other hand, the schema-based methods performed similarly as in our previous experiments, which is average, close to 0.6 $Recall@|GroundTruth|$. In this scenario, COMA's instance-based variant is the winner, with our simple baseline coming up second. Overall, these results again prove that the instance-based methods perform substantially better than their schema-based counterparts.

In Table 5.1, we present the remaining results for the Magellan and ING datasets. At first, the Magellan datasets fall into the unionable category, and the column names of every dataset are exactly the same. Thus, as expected, the

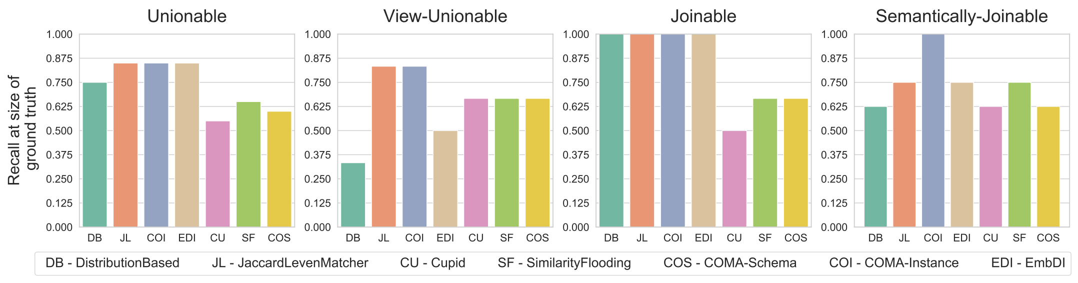


Figure 5.4: Recall at ground truth results on the Wikidata dataset

schema-based methods have a perfect score. On the other hand, the instance-based methods perform well, with COMA having a perfect score followed by EmbDI and our baseline. However, the Distribution Based method had the lowest score. This can be attributed to the fact that Magellan datasets contain multi-valued attributes (e.g., lists of actors for movie datasets), making the column's distribution very hard to calculate.

In the first ING dataset (ING#1), we observe that, except for Similarity Flooding that placed many false positives in the top ranks, all other schema-based methods reached a performance over 0.7. This performance improvement was expected because the corresponding/matching columns between the two tables share identical or very similar names. The best performing method was the Distribution Based, which was expected due to the tables containing many almost-identical values in the matching columns leading to very similar distributions. COMA and our baseline are coming in second, with a few more false positives ranked on top.

The second ING dataset (ING#2) is where things get interesting as it is the only one having the requirement of 1 to m matches from a small table to one with more than 60 columns. At first, we observe COMA's abysmal performance in what we believe to be a bug in its current version when it comes to supporting this schema matching scenario. Moreover, the schema-based methods missed many matches due to column name differences (the target table contained suffixes in its column names), leading to worse performance than in the previous ING dataset. The clear winner in both ING datasets is the Distribution Based method for the same reason as in the previous one, with our baseline coming in second. Finally, EmbDI's embedding could not accurately capture the relationships between the columns leading to its worst performance of 0.22. A reason for that might be the vastly different size of the two tables.

Methods	**Maggelan**	**ING#1**	**ING#2**
CUPID Madhavan et al. [2001]	**1**	0.71	0.5
Similarity Flooding Melnik et al. [2002]	**1**	0.36	0.44
COMA Schema-based Do and Rahm [2002]	**1**	0.79	0.12
COMA Instance-based Do and Rahm [2002]	**1**	0.79	0.14
Distribution Based Zhang et al. [2010]	0.54	**0.86**	**0.88**
Jaccard Levenshtein	0.79	0.79	0.62
EmbDI Cappuzzo et al. [2020]	0.82	0.71	0.23

Table 5.1: Recall at ground truth for the Magellan and ING Data

5.1.2 Efficiency results

The efficiency evaluation of the benchmark is split into three parts. In the first, we display and explain the efficiency of all benchmarked algorithms. Secondly, we present the scalability results of the parallelized instance-based methods of Section 3.3. Finally, we evaluate the scalability of the benchmarking system as a whole regarding how efficiently it distributes the benchmarking jobs.

Method runtime results

In Table 5.2, we present the average runtime results with standard deviation for all the methods in every dataset variant that produced the results we saw previously in Section 5.1.1.

Methods	Dataset	Average Runtime	Standard Deviation
CUPID Madhavan et al. [2001]	Prospect	16.23	8.71
	Open Data	34.61	26.57
	ChEMBL	10.65	4.66
	Wikidata	17.74	5.84
	Magellan	3.83	0.09
Similarity Flooding Melnik et al. [2002]	Prospect	24.76	30.98
	Open Data	48.16	78.15
	ChEMBL	3.99	2.76
	Wikidata	0.60	0.39
	Magellan	0.07	0.05
COMA schema-based Do and Rahm [2002]	Prospect	6.24	5.9
	Open Data	4.51	4.1
	ChEMBL	1.20	0.43
	Wikidata	2.23	0.95
	Magellan	0.59	0.09
COMA instance-based Do and Rahm [2002]	Prospect	1948.7	2540.2
	Open Data	1005.6	1168.52
	ChEMBL	45.47	26.14
	Wikidata	301.95	183.21
	Magellan	28.33	13.09
Distribution Based Zhang et al. [2011]	Prospect	79.65	32.83
	Open Data	490.12	219.84
	ChEMBL	56.51	17.36
	Wikidata	74.18	40.03
	Magellan	11.76	1.36
SemProp Fernandez et al. [2018b]	Prospect	-	-
	Open Data	-	-
	ChEMBL	735.25	234.15
	Wikidata	-	-
	Magellan	-	-
EmbDI Cappuzzo et al. [2020]	Prospect	6219.1	2354.98
	Open Data	4124.89	3915.78
	ChEMBL	2709.89	822.2
	Wikidata	3959.04	1146.91
	Magellan	5140.52	7268.44
Jaccard Levenshtein	Prospect	2350.68	1830.69
	Open Data	550.68	976.58
	ChEMBL	1008.62	822.67
	Wikidata	308.44	129.63
	Magellan	219.13	75.30

Table 5.2: Average runtime with standard deviation (in seconds) for all the methods in every dataset

All experiments were distributed as explained in Section 3.2 in two 80-core Linux virtual machines, with 320 GB of RAM each. However, those on the ING datasets ran on our partner's in-house machines for privacy reasons and are hence excluded. As expected, the schema-based methods are the best performing, with the schema-based variant of COMA being the fastest follower by CUPID and Similarity Flooding, which makes sense because it does not build complex structures like the other two and runs on the JVM instead of Python. We witness the same expected behavior from the instance-based, which was that they are going to be orders of magnitude slower, with the scaled Distribution Based method running on four threads being the fastest by far, but we have to mention that we calculated the global ranks required by the method offline, which was not taken into account for the runtime. COMA's instance-based variant also performed on par with the Distribution based method in terms of efficiency. EmbDI performed the worst, which could be attributed to its inefficient way of performing random walks (something that does not scale in our larger datasets) in addition to the embedding training phase, which is costly. The other two methods, SemProp and Jaccard Levenshtein, performed as we expected in the middle because SemProp uses pre-trained embeddings, and an external ontology is considerably faster than EmbDI. Simultaneously, Jaccard Levenshtein does pairwise checks within each column's row elements instead of matching distributions, which is a lot slower than the Distribution Based's approach.

Instance based methods scaling

After witnessing the higher runtime of the two instance-based methods that we parallelized in Section 3.3, compared to the rest as shown in Table 5.2 even after scaling them (the runtime results shown of both are while running on four threads), we believe that our parallelization efforts were justified. In Figure 5.5 and Figure 5.6, we show what level of parallelization we achieved for both methods. At first, we selected two tables from Open Data of 26 columns (676 combinations) and 11629 rows each and ran the scaled methods while increasing the number of threads used. This way, we will see the efficiency achieved.

In Figure 5.5, we see that the Distribution Based method achieves close to ideal scaling. A reason for not achieving perfect results could be because we write the intermediate results to disk to compensate for large columns that do not fit in memory. Thus the I/O times gradually impede performance more and more while the runtime improvement seems to start to plateau at six threads.

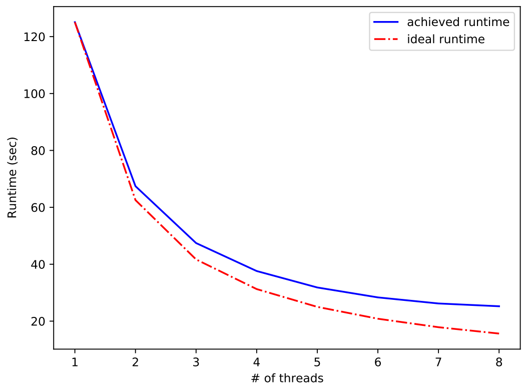


Figure 5.5: Scalability test of the parallel Distribution Based method

The Jaccard Levenshtein method shows more peculiar results in Figure 5.6, reaching a strict plateau a lot faster at four threads. The main reason for this is that we have 17 out of the 26 columns containing integers or single-character strings in the selected datasets, where the Levenshtein distance could be calculated very fast. Consequently, when we have many threads, the work finishes very fast for those that get the columns mentioned above and end up waiting on the few that have the columns with larger strings. A way to solve this would be to create a cost model for each batch, evenly distributing the more expensive columns to all threads but, since this is not as trivial as it sounds and out of the scope of this work, we decided to leave it for the future if deemed necessary.

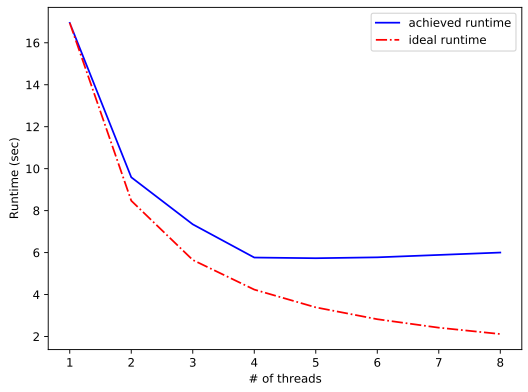


Figure 5.6: Scalability test of the Jaccard Levenshtein method

Benchmarking System's Scalability results

Now that we saw how well the two scaled instance-based methods work, we perform a test on the entire benchmarking system to evaluate the job distribution efficiency. At first, we selected 100 configuration files of two methods from both the schema and instance-based categories. The reason behind that is that the schema-based methods need to read and process a smaller amount of information and, thus, have lower I/O requirements, and we wanted to see whether that impedes the instance-based scalability efficiency. For the schema-based methods, we selected CUPID Madhavan et al. [2001], and in Figure 5.7, we show the achieved job parallelization, which is very close to ideal as expected. For the instance-based, we selected the Distribution Based Zhang et al. [2011] method as indicative of the category, and in Figure 5.8, we observe that it too reached ideal performance.

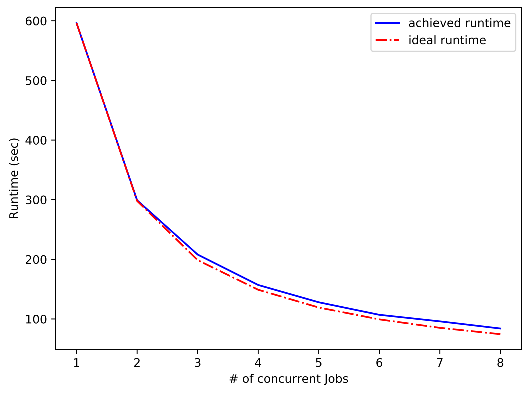


Figure 5.7: Efficiency of CUPID jobs' parallelization

Based on our rough estimates, without the development of such a system, it would take close to 150 days to perform such an extensive benchmark as the one we did in Valentine Koutras et al. [2020b] where we finished them within a week while running on two 80 core virtual machines. The reason that the math does not add up since 150 days divided by 160 cores would be less than a day, is that as we mentioned before, the Distribution Based and Jaccard Levenshtein ran on four threads, each requiring more resources than the rest. Additionally, the SemProp Fernandez et al. [2018b] and EmbDI Cappuzzo et al. [2020] methods taken as-is from their respective authors' repositories had some design nuances that limited their job distribution potential. For example, SemProp required an external elastic search dependency that needed to be separated between each job instance, and EmbDI had a Deep Learning framework that used up all available resources while running, making the scheduling of the rest harder.

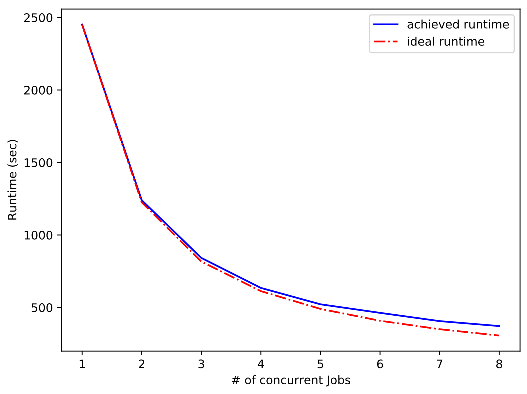


Figure 5.8: Efficiency of Distribution Based jobs' parallelization

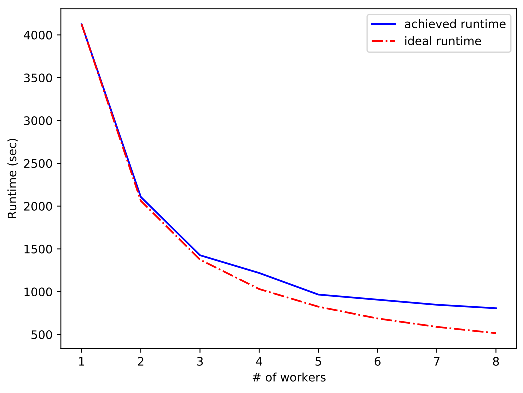
5.2 Schema Matching System

5.2.1 Scalability tests

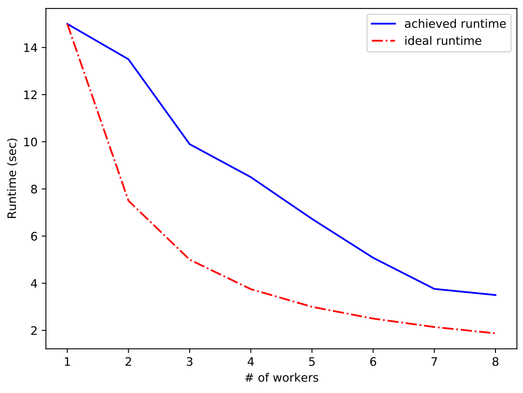
In this section, we evaluate the performance of the proposed schema matching system of Chapter 4. Since the methods are the same as in the benchmark, their efficiency is displayed in Table 5.2. Thus, here we explore how well the system scales by increasing the number of workers that tackle a set schema matching problem. In our tests, each worker has 2 CPUs and 8 GB of memory.

The first schema matching scenario we selected is finding all the related attributes within the TPCH[1] dataset consisting of eight tables of varying size that describe a webshop. This scenario contains 64 schema matching tasks (one for each table combination). In Figure 5.9, we present the scalability results achieved in this problem for two schema matching methods, Distribution Based and Cupid. We select one descriptive method from each category (schema or instance-based) because, as we saw in Table 5.2 the instance-based are close to three orders of magnitude more demanding. Figure 5.9a shows that the system achieves close to linear scalability for the Distribution Based method while taking into account that this is a distributed system with messaging networking and I/O costs. The performance degradation witnessed after the third worker results from TPCH's characteristics rather than the system's poor scalability because the problem has eight costly tasks related to the largest table out of the eight. Meaning that while four workers deal with them, the others have finished their work and wait.

[1] http://www.tpc.org/tpch/



(a) Distribution Based Zhang et al. [2011]



(b) CUPID Madhavan et al. [2001]

Figure 5.9: Scalability test for the TPCH scenario

In Figure 5.9b we run the same scenario using CUPID, and the scalability results are non-descriptive, as expected, due to the small size of the scenario (64 table pairs) and the fast speed of the schema-based methods (runtime of less than a second) we observe that the scheduling, networking, and I/O costs are more significant than the performance gains of the task distribution. Thus, to evaluate the system's scalability for the schema-based methods, we created a different scenario consisting of all the fabricated joinable datasets of Nargesian et al. [2018] (Open Data), used in the Valentine benchmark Koutras et al. [2020b] detailed in Section 3.4, of more than 9000 table pairs. In Figure 5.10, we observe the previously described scenario's scalability results. They match our expectations by showing linear scalability, and also, it is better than that of the Distribution Based's scenario due to the uniformity of the cost across all tasks (i.e., all the tasks cost the same to compute). To sum up, from the displayed experiments, it is safe to conclude that the developed schema matching system achieves close to linear scalability.

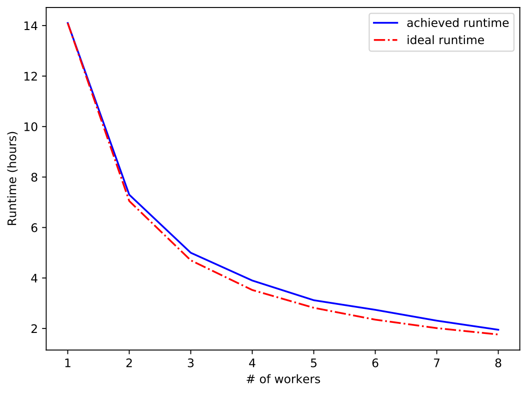


Figure 5.10: Scalability test of CUPID in the joinable case of Open-Data

Chapter 6

Conclusion

6.1 Summary

This thesis aims at solving the scalability problem in holistic schema matching, which came in two parts. At first, we wanted to investigate which schema matching methods could fit such a system leading to our first contribution, the scalable benchmarking component used in Valentine Koutras et al. [2020b] while answering our **first research question**:

Is there any real progress in schema matching in the past two decades?
As we saw in our effectiveness and efficiency evaluation (Section 5.1), the answer is complicated. While COMA was taking a substantial lead in almost all of our devised schema matching scenarios, our finding that COMA does not work with 1 to m, n to 1, or n to m matches while running the ING#2 dataset made it hard to declare it as the winner. Additionally, from the efficiency standpoint, other instance-based methods performed better, and again in the ING case in both datasets, the Distribution Based algorithm was the best performing method. This fact leads us to integrate a selection of methods to our scalable holistic schema matching system, as explained in Section 4.4.

However, to get the previous findings, we needed an extensive schema matching benchmark, which we created while answering our second research question:

How to construct an extensive schema matching benchmark with a primary focus on its scalability?
We explored every possibility in everything related to scale (scaling up to multiple threads in the same machine and scaling out to multiple machines), starting with parallelizing the two slowest instance-based methods, as shown in Section 3.3 and achieving good scalability results. Then we created a batch job system that achieved linear scalability (Section 3.2) that allowed us to finish the method evaluation in a week that would have taken almost half a year (if it was running sequentially).

Having all the needed information about the schema matching methods from our previous contributions, we answered our third research question:

What should be the design of an elastic, scalable, semi-automated, holistic schema matching system?
With those requirements in mind, we built a system that fulfills them all, works on the popular Kubernetes framework on the cloud, and is thoroughly detailed in Chapter 4.

Finally, we wanted to make our semi-automated schema matching system more user-oriented, answering our fourth research question:

How to present the proposed matches to multiple users, increasing the speed of match verification?
We created a more user-friendly UI, as presented in Section 4.6, by making the match verification task a microtask, inspired by the related work that we found on crowdsourcing, making the process far simpler and scalable to multiple users.

To sum up, this work was a first attempt at solving such a vast and complicated problem with open issues that can be addressed as future work to make such a system more complete and fine-tuned.

6.2 Future Work

In this section we propose some improvements that can be made in the future in both the holistic schema matching system and benchmark.

6.2.1 Schema Matching Benchmark

In the case of the schema matching benchmark, in the future, we could improve its accessibility to the community by creating an open portal where researchers in the field can easily compare their methods and compete with each other on all schema matching scenarios. This competitiveness is a proven progress driver in many fields but not yet in schema matching.

Another obvious improvement would be to make the benchmark more extensive with more methods and datasets.

Finally, on the engineering side, the benchmarking system could be made to work on the cloud, similar to the developed holistic schema matching system making its lifecycle a lot more manageable.

6.2.2 Schema Matching System

For the proposed schema matching system, we could do a few performance optimizations like caching the data received frequently from the data sources to not substantially increase their load whenever the system wants to perform a schema matching job.

Furthermore, we could add more features like job priority, making the jobs run when the data sources are not heavily used, job progress reports, reuse previously computed jobs when applicable, and more.

Additionally, we could add a search space reduction module that excludes schema matching tasks with a low probability of containing matches as we found in Section 2.3.

Finally, we can combine multiple matchers. For example, enhance some of the best performing instance-based methods with the best schema-based to make use of all the information available at our disposal.

6.2.3 Expert evaluation

For the UI part of our schema matching system where the users request jobs and evaluate the proposed matches, a noticeable improvement would be to prettify the UI. Show the verified matches with a better, graph-like visualization. Finally, the most crucial future improvement would be to measure how much our design improved the users' productivity in the match verification process and how well it scaled.

Appendix A

Schema Matching Benchmark

A.1 JSON schema file

```
{
    "musician": {
        "type": "text"
    },
    "birthDate": {
        "type": "text"
    },
    "familyNameLabel": {
        "type": "text"
    },
    "givenNameLabel": {
        "type": "text"
    },
    "numberOfChildren": {
        "type": "integer"
    },
    "websiteLabel": {
        "type": "text"
    },
    "residenceLabel": {
        "type": "text"
    },
    "ethnicityLabel": {
        "type": "text"
    },
    "religionLabel": {
        "type": "text"
    },
    "activityStart": {
        "type": "text"
    },
    "twitterNameLabel": {
        "type": "text"
    },
    "geniusNameLabel": {
        "type": "text"
    },
    "recordLabelLabel": {
        "type": "text"
    }
}
```

Figure A.1: Schema file example on Wikidata's Musicians dataset

A.2 JSON golden standard file

```
{
    "matches": [
        {
            "source_table": "musicians_joinable_source",
            "source_column": "musician",
            "target_table": "musicians_joinable_target",
            "target_column": "musicianID"
        },
        {
            "source_table": "musicians_joinable_source",
            "source_column": "birthDate",
            "target_table": "musicians_joinable_target",
            "target_column": "birthDate"
        },
        {
            "source_table": "musicians_joinable_source",
            "source_column": "familyNameLabel",
            "target_table": "musicians_joinable_target",
            "target_column": "familyName"
        },
        {
            "source_table": "musicians_joinable_source",
            "source_column": "givenNameLabel",
            "target_table": "musicians_joinable_target",
            "target_column": "forename"
        },
        {
            "source_table": "musicians_joinable_source",
            "source_column": "numberOfChildren",
            "target_table": "musicians_joinable_target",
            "target_column": "NChildren"
        },
        {
            "source_table": "musicians_joinable_source",
            "source_column": "websiteLabel",
            "target_table": "musicians_joinable_target",
            "target_column": "webpage"
        }
    ]
}
```

Figure A.2: Golden standard example on the Wikidata's Musicians dataset

A.3 Schema matching job's output

```
{
  "name": "Musicians_viewunion__Coma{'max_n': 0, 'strategy': 'COMA_OPT_INST'}",
  "matches": {
    "(('musicians_viewunion_source', 'birthDate'), ('musicians_viewunion_target', 'birthDate
        '))": 0.67182,
    "(('musicians_viewunion_source', 'familyNameLabel'), ('musicians_viewunion_target', '
        familyName'))": 0.56238514,
    "(('musicians_viewunion_source', 'musician'), ('musicians_viewunion_target', 'musicianID
        '))": 0.53427684,
    "(('musicians_viewunion_source', 'numberOfChildren'), ('musicians_viewunion_target', '
        NChildren'))": 0.45804533,
    "(('musicians_viewunion_source', 'givenNameLabel'), ('musicians_viewunion_target', '
        forename'))": 0.3680327,
    "(('musicians_viewunion_source', 'websiteLabel'), ('musicians_viewunion_target', 'webpage
        '))": 0.32513022,
    "(('musicians_viewunion_source', 'twitterNameLabel'), ('musicians_viewunion_target', '
        motherName'))": 0.311179,
    "(('musicians_viewunion_source', 'ethnicityLabel'), ('musicians_viewunion_target', 'city
        '))": 0.28124666,
    "(('musicians_viewunion_source', 'recordLabelLabel'), ('musicians_viewunion_target', '
        kind'))": 0.24016242
  },
  "metrics": {
    "precision": 0.5555555555555556,
    "recall": 0.8333333333333334,
    "f1_score": 0.6666666666666667,
    "precision_at_10_percent": 1.0,
    "precision_at_20_percent": 1.0,
    "precision_at_30_percent": 1.0,
    "precision_at_40_percent": 1.0,
    "precision_at_50_percent": 0.8,
    "precision_at_60_percent": 0.8333333333333334,
    "precision_at_70_percent": 0.7142857142857143,
    "precision_at_80_percent": 0.625,
    "precision_at_90_percent": 0.5555555555555556,
    "recall_at_sizeof_ground_truth": 0.8333333333333334
  },
  "run_times": {
    "total_time": 292.26641100700004,
    "algorithm_time": 292.26639625700005
  }
}
```

Figure A.3: Job's output while running COMA on WikiData's Musicians dataset

A.4 Method’s Configuration

```
"JaccardLevenMatcher": {
  "data_loader_type": "InstanceLoader",
  "args": {
    "threshold_leven": {
      "type": "range",
      "max": 0.8,
      "min": 0.4,
      "step": 0.1
    },
    "process_num": {
      "type": "values",
      "data": [
        4
      ]
    }
  }
}
```

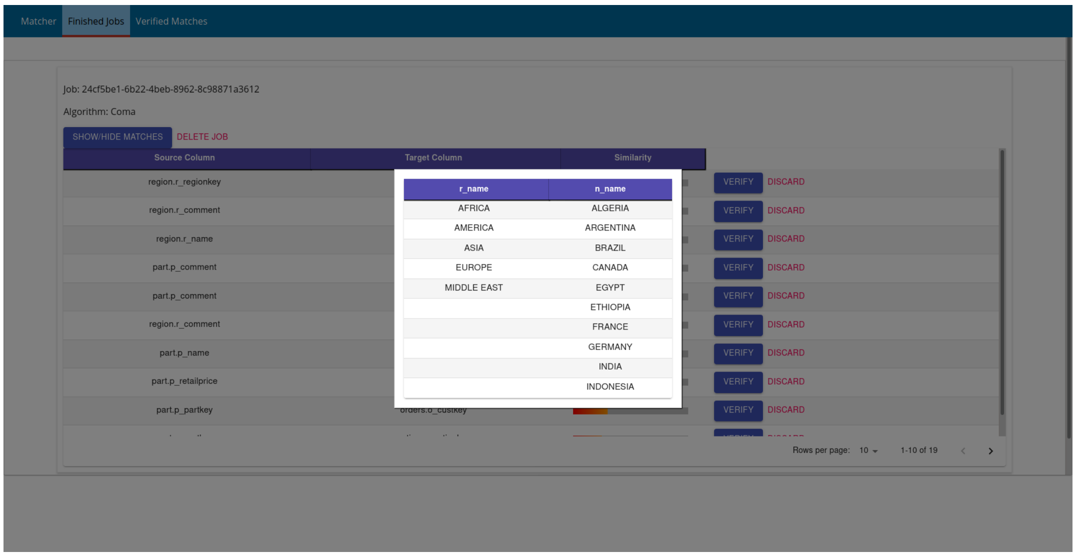
Figure A.4: Our baseline matcher’s configuration file

Appendix B

Schema Matching System

B.1 UI Components

B.1.1 Data Preview Page



Job: 24cf5be1-6b22-4beb-8962-8c98871a3612

Algorithm: Coma

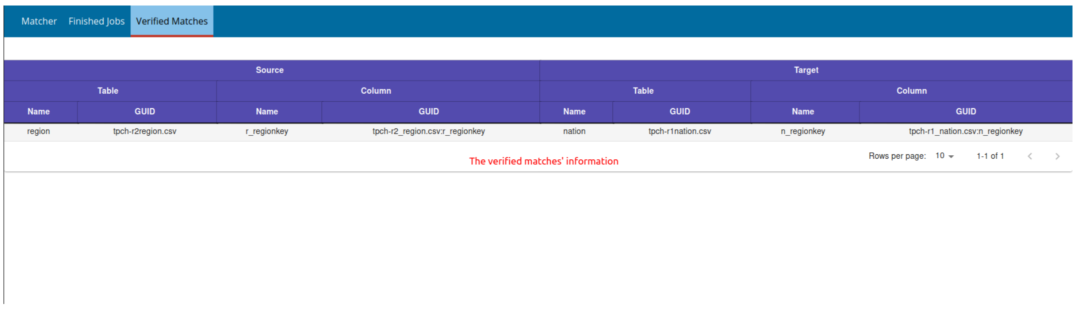
Source Column	Target Column	Similarity		
region.r_regionkey			VERIFY	DISCARD
region.r_comment			VERIFY	DISCARD
region.r_name			VERIFY	DISCARD
part.p_comment			VERIFY	DISCARD
part.p_comment			VERIFY	DISCARD
region.r_comment			VERIFY	DISCARD
part.p_name			VERIFY	DISCARD
part.p_retailprice			VERIFY	DISCARD
part.p_partkey	orders.o_custkey		VERIFY	DISCARD

r_name	n_name
AFRICA	ALGERIA
AMERICA	ARGENTINA
ASIA	BRAZIL
EUROPE	CANADA
MIDDLE EAST	EGYPT
	ETHIOPIA
	FRANCE
	GERMANY
	INDIA
	INDONESIA

Rows per page: 10 1-10 of 19

Figure B.1: The data preview component of the match verification page

B.1.2 Verified Matches Page



Source				Target			
Table		Column		Table		Column	
Name	GUID	Name	GUID	Name	GUID	Name	GUID
region	tpch-r2region.csv	r_regionkey	tpch-r2_region.csv:r_regionkey	nation	tpch-r1nation.csv	n_regionkey	tpch-r1_nation.csv:n_regionkey

The verified matches' information

Rows per page: 10 1-1 of 1

Figure B.2: The verified matches page of our UI

Bibliography

B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *Proc. VLDB Endow.*, 1(1):230–244, Aug. 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453886. URL https://doi.org/10.14778/1453856.1453886.

M. B. Amin, W. A. Khan, S. Hussain, D.-M. Bui, O. Banos, B. H. Kang, and S. Lee. Evaluating large-scale biomedical ontology matching over parallel platforms. *Iete Technical Review*, 33(4):415–427, 2016.

P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The IBench integration metadata generator. *Proc. VLDB Endow.*, 9(3):108–119, Nov. 2015. ISSN 2150-8097. doi: 10.14778/2850583.2850586. URL https://doi.org/10.14778/2850583.2850586.

D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with coma++. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908, 2005.

C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4): 323–364, 1986.

Z. Bellahsene, A. Bonifati, and E. Rahm. *Schema Matching and Mapping*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 9783642165177.

P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *ACM Sigmod Record*, 33(4):38–43, 2004.

P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental schema matching. In *VLDB*, volume 6, pages 1167–1170. Citeseer, 2006.

P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *Proceedings of the VLDB Endowment*, 4(11):695–701, 2011.

R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1335–1349, 2020.

C. Chen, B. Golshan, A. Y. Halevy, W.-C. Tan, and A. Doan. Biggorilla: An open-source ecosystem for data preparation and integration. *IEEE Data Eng. Bull.*, 41(2):10–22, 2018.

S. Das, A. Doan, C. G. PSGC, P. Konda, Y. Govind, and D. Paulsen. The magellan data repository, 2015.

D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment*, 9(4):360–371, 2015.

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.

A. Doan, P. Domingos, and A. Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Machine Learning*, 50(3):279–301, 2003.

F. Duchateau and Z. Bellahsene. Designing a benchmark for the assessment of schema matching tools. *Open Journal of Databases*, 1(1):3–25, 2014.

F. Duchateau, Z. Bellahsene, and E. Hunt. Xbenchmatch: a benchmark for xml schema matching tools. In *The VLDB Journal*, volume 1, pages 1318–1321. Springer Verlag, 2007.

M. Ehrig and S. Staab. Qom – quick ontology mapping. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *The Semantic Web – ISWC 2004*, pages 683–697, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30475-3.

M. Ehrig, S. Staab, and Y. Sure. Bootstrapping ontology alignment methods with apfel. In *International semantic Web conference*, pages 186–200. Springer, 2005.

D. Engmann and S. Massmann. Instance matching with coma++. In *BTW workshops*, volume 7, pages 28–37, 2007.

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, 2018a.

R. C. Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 989–1000. IEEE, 2018b.

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.

P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

A. Gross, M. Hartung, T. Kirsten, and E. Rahm. On matching large life science ontologies in parallel. In *International Conference on Data Integration in the Life Sciences*, pages 35–49. Springer, 2010.

N. Q. V. Hung, N. T. Tam, Z. Miklós, and K. Aberer. On leveraging crowdsourcing techniques for schema matching networks. In *International Conference on Database Systems for Advanced Applications*, pages 139–154. Springer, 2013.

C. Koutras, M. Fragkoulis, A. Katsifodimos, and C. Lofi. Rema: Graph embeddings-based relational schema matching. In *EDBT/ICDT Workshops*, 2020a.

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, 2020b.

Y. Lee, M. Sayyadian, A. Doan, and A. S. Rosenthal. ETuner: tuning schema matching software using synthetic scenarios. *The VLDB Journal*, 16(1):97–122, Jan. 2007. ISSN 1066-8888. doi: 10.1007/s00778-006-0024-z. URL https://doi.org/10.1007/s00778-006-0024-z.

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

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

J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *21st International Conference on Data Engineering (ICDE'05)*, pages 57–68. IEEE, 2005.

S. Massmann, S. Raunich, D. Aumüller, P. Arnold, and E. Rahm. Evolution of the coma match system. In *Proceedings of the 6th International Conference on Ontology Matching-Volume 814*, pages 49–60. CEUR-WS. org, 2011.

S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th International Conference on Data Engineering*, pages 117–128. IEEE, 2002.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. Howard Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *ACM Sigmod Record*, 30 (1):78–83, 2001.

P. Mork, L. Seligman, A. Rosenthal, J. Korb, and C. Wolf. The harmony integration workbench. In *Journal on Data Semantics XI*, pages 65–93. Springer, 2008.

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.

F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data lake management: Challenges and opportunities. *Proceedings of the VLDB Endowment*, 12 (12):1986–1989, 2019.

F. Nauman, Ching-Tien Ho, Xuqing Tian, L. Haas, and N. Megiddo. Attribute classification using feature analysis. In *Proceedings 18th International Conference on Data Engineering*, pages 271–, 2002.

A. Nazi, B. Ding, V. Narasayya, and S. Chaudhuri. Efficient estimation of inclusion coefficient using hyperloglog sketches. *Proceedings of the VLDB Endowment*, 11(10):1097–1109, 2018.

E. Peukert, H. Berthold, and E. Rahm. Rewrite techniques for performance optimization of schema matching processes. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 453–464, 2010.

M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield. Tpc-di: the first industry benchmark for data integration. *Proceedings of the VLDB Endowment*, 7(13): 1367–1378, 2014.

E. Rahm. Towards large-scale schema and ontology matching. In *Schema matching and mapping*, pages 3–27. Springer, 2011.

E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

E. Rahm and E. Peukert. Large-scale schema matching., 2019.

D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The evolution of data to life-critical. *Don't Focus on Big Data*, 2017.

G. G. Robertson, M. P. Czerwinski, and J. E. Churchill. Visualization of mappings between schemas. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 431–439, 2005.

D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015.

L. Seligman, P. Mork, A. Halevy, K. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. Openii: an open source information integration toolkit. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1057–1060, 2010.

P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. In *Journal on data semantics IV*, pages 146–171. Springer, 2005.

C. J. Zhang, Z. Zhao, L. Chen, H. V. Jagadish, and C. C. Cao. Crowdmatcher: crowd-assisted schema matching. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 721–724, 2014.

M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3 (1-2):805–814, 2010.

M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 109–120, 2011.

E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. Lsh ensemble: internet-scale domain search. *Proceedings of the VLDB Endowment*, 9(12):1185–1196, 2016.

E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*, pages 847–864. ACM, 2019.