Journal on Data Semantics manuscript No.
(will be inserted by the editor)

On Generating Benchmark Data for Entity Matching

Ekaterini Ioannou · Nataliya Rassadko · Yannis Velegrakis

Received: date / Accepted: date

Abstract Entity matching has been a fundamental task in every major integration and data cleaning effort. It aims at identifying whether two different pieces of information are referring to the same real world object. It can also form the basis of entity search by finding the entities in a repository that best match a user specification. Despite the many different entity matching techniques that have been developed over time, there is still no widely accepted benchmark for evaluating and comparing them. This paper introduces EMBench, a principled system for the evaluation of entity matching systems. In contrast to existing similar efforts, EMBench offers a unique test case generation approach that combines different levels of types, complexity and scales, allowing a complete and accurate evaluation of the different aspects of a matching system. After presenting the basic principles of EMBench and its functionality, a comprehensive evaluation is performed on some existing matching systems that showcases its discriminative power in highlighting their capabilities and limitations.EMBench has all the characteristics of a benchmark, and can serve as a standard evaluation methodology provided that it gains popularity and wide acceptance.

Keywords Data Integration, Matching Benchmark, Entity Matching

Ekaterini Ioannou
Technical University of Crete, Greece.
E-mail: ioannou@softnet.tuc.gr

Nataliya Rassadko
University of Trento, Italy.
E-mail: rassadko@disi.unitn.it

Yannis Velegrakis
University of Trento, Italy.
E-mail: velgias@disi.unitn.eu

1 Introduction

A fundamental problem in every information integration and data cleaning application is the ability to identify whether two different data structures are modeling the same real world object, i.e., an event, a location, a book, a conference, a person, etc. The task is typically known as *entity matching*, but is also found in the literature as record linkage, deduplication [72], entity resolution [80], merge-purge [44], entity identification [58], reference reconciliation [29], and entity linkage [45, 46]. What makes the task a challenge is the heterogeneity that naturally exists in the data because they have been produced by different applications, because errors and inconsistencies occur naturally in the data, or even because the different data engineers that designed and developed the respective systems did so having in mind different requirements.

Matching has been used in many different aspects of data management, such as ontology matching [40], and schema matching [12]. Entity matching is a similar concept in which the fundamental structure of the data model is an *entity*, i.e., a structure that models a real world object. An entity is different from schema structures, such as tuples or tables whose design is often driven by performance or application specific requirements. Due to this, they may only partially describe a real world object or may represent a combination of more than one conceptually different objects in the same structure, maybe in order to avoid costly operations like joins. Furthermore, schemas describe the structures to which a collection of data elements is supposed to conform, while entities are the actual instances of these structures. Entity matching is also different from ontology matching since the ontology contains schema information, such as classes, and isA relationships. Entity matching is what typically meant by the term *instance matching*, i.e., the matching at the individual (instance) level [60]. Entities have become the fun-

damental structures in Semantic Web applications, which has boosted the interest of the research and industrial community on entity matching [25, 79, 47] for heterogeneous web data integration and for query answering. In the latter, a structure describing the specifications set by a user query is matched across the structures stored in a database to identify those that best satisfy these specifications [56, 77, 10]. Entity matching has also become important in entity evolution [21] where the identification of similarities across different data structures may signal that the two structures represent different phases of the lifespan of a single entity [79].

Matching approaches are typically based on some similarity function that measures syntactic and semantic proximity of two structures. Depending on the results of this comparison, it is decided whether the two structures are matching or not. The similarity function may additionally take into consideration auxiliary information, such as online dictionaries, log files, co-reference, or other forms of existing knowledge. Apart from atomic similarity methods that compare strings or sets of strings [23], other techniques that have been proposed are collective matching [13, 29], blocking [64, 80], and solutions exploiting the schema structure in general [35, 40].

Unfortunately, despite the many different techniques for entity matching there is no evaluation methodology that is widely accepted or used. Most matching techniques have followed their own ad-hoc evaluation approach, tailored to their own specific goals. Comparison among entity matching systems and selection of the best system for a specific task at hand is becoming a challenge. Developers can not easily test the new features of the products they develop against competitors, practitioners can not make informative choices for the most suitable tool to use, and researchers can neither compare the techniques they are developing against those already existing, neither identify existing limitations that can serve as potential research directions.

There have been efforts towards the creation of such an evaluation methodology though. A large portion of these efforts is based on the creation of test cases extracted from real world situations. This has the advantage that the matching system is tested against situations that are actually met in practice, but suffers from three main limitations. The first is that the extracted scenarios may have been created through the convolution of more than one kind of heterogeneities to a degree that it is not clear even to an expert user what these heterogeneities are. Knowing that a system cannot support such a scenario is not informative enough to pinpoint the limitation of the under evaluation system, i.e., the kind of heterogeneity it cannot handle. For this reason, it is important for an evaluation methodology not only to report the unsuccessful scenarios, but the heterogeneity that each such scenario was supposed to test. The second limitation is that the testing scenarios are restrictive to only those that are currently met in practice. Nowadays, we are all witnessing an unprecedented rate of data generation, not only in terms of size but also in terms of structural and semantic complexity. As such, an evaluation methodology should be able to create new test cases that may not be currently present in existing data sets, yet are very likely to be found in the near future through the integration or processing of the existing datasets. The third limitation is that most of the test data are static. The evaluation scenarios should include not only cases of different heterogeneities, but also scenarios with varying data sizes that can test how well the under evaluation matching system can cope with data at different scales [32].

In this paper, we present a principled, systematic, and consistent evaluation framework for entity matching systems and its implementation in a tool called EMBench. We have studied many practical situations and many existing test scenarios of entity matching, such as those offered from OAEI [60], the largest entity matching evaluation initiative nowadays, but instead of taking these examples as they are, we have analyzed them and identified the actual heterogeneities that characterize them. For each such heterogeneity we allow the creation of a test case that evaluates whether and to what degree a matching tool can successfully perform its task in the present of the respective form of heterogeneity. The characteristics of each test case are fully configurable through a set of parameters controlling aspects like size, distribution, and complexity. The way the test case generation is achieved is by exploiting a large set of raw real world data we have collected. The data is first combined to form an initial collection of entities I_o. Some heterogeneity of interest is then injected into this collection, leading to a modified collection I_h. A test case is formed by an entity e_o selected from the original collection I_o, its heterogeneity injected form e_h in the collection I_h and the collection I_h itself. For the evaluation, the matching system is provided with the collection I_h, and asked to find the entity that best matches e_o. If successful, the entity e_h will be returned. EMBench can generate multiple test cases to provide a complete understanding of the capabilities and limitations of the under evaluation matching system. EMBench is designed in a way that different metrics can be plugged in to measure the success rate of the matching system from different points of view. After presenting the different features of EMBench and explaining its design principles, we illustrate how these features can be used for designing and performing evaluations of entity matching tools.

We need to emphasize here that this work is not an evaluation study and our goal is not to designate the best entity matching tool. Different tools perform better in different situations, so there is clearly no single winner as many similar efforts have already demonstrated [30, 31, 51, 53, 81]. Our goal is to illustrate the way EMBench works and its powers in highlighting the capabilities and limitations of the entity

matching systems. An effort similar to ours is the SWING benchmark [37], however, the examples generated with this approach for testing are not as diverse as ours, neither the user has the flexibility and the control that our approach is offering. The extensively used OAEI initiative [60] on the other hand, is still lacking the dynamism and scaling capabilities of EMBench, but the ideas proposed here can be easily incorporated into OAEI and strengthening it in the entity matching evaluation arena.

Note that our evaluation framework has all the characteristics of a benchmark. It is based on the same design principles, e.g., TPC-H[1], or STBenchmark [2]. Yet, we avoid calling it a benchmark since it is original work and is not yet standardized. Doing so is something that is outside our control and orthogonal to the purpose of this work. It is expected that if widely adopted it can become a standard or implemented into one.

The main contributions of the paper can be summarized as the designed and implementation of a system for benchmarking entity matching systems in a generic, complete, and principled way. The system provides a series of test cases that cover the majority of the matching situations that are met in practice and which the existing matching systems are expected to support. In contrast to other similar proposals, even in different areas, our system is fully configurable and allows the dynamic (i.e., on-the-fly) generation of the different test cases in terms of different sizes and complexities both at the schema and at the instance level. The fact that the entity matching scenarios are created in a principled way, allows the identification of the actual type of heterogeneities that the under evaluation matching system does not support. To illustrate the functionalities and capabilities of our system we use it to evaluate some known entity matching systems and describe our experience.

The remainder of the paper is organized as follows. Section 2 provides an overview of the related work and explains how our proposal differs from similar efforts. Section 3 models formally the entity matching problem, and Section 4 discusses the requirements of an entity matching evaluation framework. Section 5 defines formally what an evaluation scenario is and presents the types of evaluation scenarios we consider. Section 6 presents the implementation of the evaluation framework. Section 7 discusses the usage of our evaluation framework, and Section 8 reports the results of our experimental evaluation on a number of different aspects. The current work concludes with Section 9 that wraps-up the main contributions and possible extensions.

[1] http://tpch.org

2 Related Work

2.1 Entity Matching

To better understand the aspects of matching system that a benchmark should evaluate, it is important to understand the way entity matching operates [26, 32, 39, 38]. Entity matching techniques can be grouped into five main categories.

The first category consists of those approaches that use similarity techniques at the atomic level to decide how close two structures are. They handle cases like “John D. Smith” vs. “J. D. Smith”, and “Transactions on Knowledge and Data Engineering” vs. “IEEE Trans. Knowl. Data Eng.”. Differences in strings are a typical consequence of misspellings or naming variations due to the use of abbreviations, acronyms, etc. Matching is performed by detecting the resemblance between the text values found in the entities. There are already interesting surveys that cover in details the majority of these techniques [15, 23].

The second category contains techniques that view entities as a set of atomic values. A classical example is a relational record representing an entity. To reduce the problem into the one of comparing atomic values, two of the very first techniques proposed in the area [22, 52] concatenate all the atomic values of each entity into one string which is then used in atomic value comparisons as a representative of the entity. The approaches proposed in [75] and [27] aim at matching entities by discovering possible mappings from one entity to another. More specifically, in [75] a mapping is found by applying a collection of *transformations*, such as abbreviation, stemming, and the use of initials. For the same purpose, Doan et al. [27] apply *profilers*, which are described as predefined rules with knowledge about specific concepts. Profilers are created from domain experts, are learned from training data, or are constructed from external data.

The third category focuses on collective matching. Instead of using the information from two entities only, the collective matching suggests the exploitation of the information about the matched entities between two sets. As an example consider co-authorship in publications. By knowing that a publication has α, β, and γ as authors, and another publication has β’, and γ as authors, one can say with higher confidence that β describes the same author as β’. Such information is captured by existing relationships between entities. The approach in [4] uses dimensional hierarchies to model the collection of entities, while the approaches introduced in [13] and [50] use graphs. Ananthakrishna et al. [4] exploit dimensional hierarchies to detect fuzzy duplicates in dimensional tables. The hierarchies are built by following the links between the data from one table to the data from other tables. Entities are matched when the information across the generated hierarchies is found similar. An-

other well-know algorithm is the *Reference Reconciliation* [29] where one starts by identifying possible relationships between entities by comparing their entity literals. The information encoded in the matches between the literals is propagated to the rest of the parent entities which in turn increases the confident for the matching of the children. The approach introduced in [46] models the data into a Bayesian network, and uses probabilistic inference for computing the probabilities of entity matches and for propagating the information between matches.

Blocking based methods form the fourth category. The idea of blocking is that instead of comparing each entity with all other entities, the entities are separated into blocks and each entity is compared only with the entities inside the same block. The challenge is to create blocks of entities that are most likely to refer to the same real world objects. Many techniques are typically associating each entity with a value summarizing the values of selected attributes and then operate exclusively on it. For instance, the *canopy clustering* [55] employs a string similarity metric for building high-dimensional overlapping blocks, whereas the *suffix arrays* approach [1] considers the suffixes of the block value. More recent blocking methods focus not only on scaling the matching process to large datasets, but also on capturing additional issues related to entity matching. For instance, [62, 64] investigate how to apply the blocking mechanism on heterogeneous semi-structured data with loose schema binding. They introduced an attribute-agnostic mechanism for generating the blocks and explained how efficiency can be improved through scheduling the order of block processing and identifying when to stop the processing. The approach introduced in [80] processes iteratively blocks in order to use the results of one block in the processing step of another block. The idea of iteratively block processing was also studied in [67]. It provided a principled framework with message passing algorithms for generating a global solution for the entity resolution over the complete collection.

The last category contains techniques that exploit schema information. Entity matching highly overlaps with schema [40] and ontology matching [35], yet it is not the same. In schema matching, the goal is to identify correspondences between attributes or structures of the schema that model the same concept. This means that the correspondences generated by the schema matching task do not necessarily connect entities. This is actually one of the main reasons why schema matching is not enough for query answering or data exchange and needs to be followed by a schema mapping task [36]. The same applies also to ontology matching. Although ontologies are much closer to the concept model of entities [73] than schemas, an ontology may model additional concepts that are not necessarily stand alone real world objects. An ontology, for instance, may contain a vocabulary of concepts, or some terminologies, and the results of an entity matching task may be correspondences among attributes that simply describe characteristics.

2.2 Benchmarking Data

[Static Datasets] A common approach that many matching proposals have followed in order to evaluate their techniques is to use some test dataset and measure the degree of success of their system on it. Cora[2] is one of these testing datasets and was used for evaluating various algorithms, such as [55], [24], [14], [65], [29], [5], and [46]. The dataset contains 1,295 scientific publications from the Cora Computer Science Research Paper Engine, and more specifically it includes around 9,700 descriptions for 2,800 real world author entities. Accuracy is typically measured in terms of precision and recall based on the given mappings between the different author descriptions. Since Cora is a small size dataset, it has not been used for scalability experiments. Scientific publications are popular datasets for evaluating matching algorithms [49, 50]. The datasets are generated by capturing a fraction of the data from an existing application, such as Citeseer[3] or DBLP[4]. Other types of datasets that have been used are products [7], such as CDs or DVDs extracted from Yahoo! Shopping services, travel data from various travel web sites, and movies [45] extracted mainly from IMDB[5]. Another interesting kind of scenarios [57] is those derived from Personal Information Management (PIM) systems. There are examples with data describing up to 3,000 people [29, 46]. Other well known collection is the one provided by the University of Texas[6]. Unfortunately, this pluralism of datasets is naturally leading to a number of issues related to the repeatability of the experimental evaluation and the direct comparison among matching systems, unless the exact datasets that were used in the evaluation are available and applicable to other systems, and there is a common agreement on the steps to follow for the evaluation procedure.

Having in mind this limitation, among others, the OAEI [60] initiative has been founded. It is a global effort of evaluation campaigns from many researchers in the ontology matching community. In each campaign, a set of test cases is provided to the participants that use their matching tools to generate matching results and then report them back into some specific format. Then, the results are evaluated based on some metrics, which are typically the precision and recall. The testing sets provided by OAEI are very rich and

[2] http://www.cs.umass.edu/~mccallum/data.html
[3] http://citeseer.ist.psu.edu/
[4] http://www.informatik.uni-trier.de/~ley/db/
[5] http://www.imdb.com/
[6] http://www.cs.utexas.edu/users/ml/riddle/data.html

cover many cases, yet, they are static, in the sense, that an evaluator does not have the capability to create her own test cases.

[Benchmarking Systems] Several systems have been developed to generate benchmarking data in many different fields, with TPC-H query engine benchmark being probably the most well known. In contrast to query answering engines that benefit from the format semantics of the query language in describing the expected correct answer, benchmarks on schema, ontology and entity matching are more challenging since the semantics involved in the process may cause a number of issues regarding what the actual correct answer is. For this reasons, test cases in these benchmarks are coming with some expected answer used as the ground truth. Evaluation techniques for schema mapping systems have been studied and described in some of our own previous work in books [6, 17] and tutorials [18, 42]. Furthermore, many of the design principles of our benchmark are based on the design principles of the TPC-H and STBenchmark [2, 3] mapping evaluation system that we have previously developed. For the area of schema and ontology matching [74], there is already a plethora of evaluation efforts [81, 30, 31, 78, 70] which can serve as the basis for a benchmark. A comprehensive description of what such a benchmark should contain can be found in a dedicated to the topic recently published book chapter [35].

[Synthetic Data Generation] The schema and ontology matching systems operate on data that are mostly for the schema level [34]. However, the entities, on which the entity matching systems operate, are at the instance level, and, due to the heterogeneity they have, they may not even conform to some schema. Thus, it is important to be able to generate test data containing entities, i.e., instance level data. As of 2009, the OAEI campaign included a special track on instance matching. In this track, the participants were asked to apply various transformations to over 4,000 individual entities retrieved from Freebase, and to match around 10,000 RDF entities (i.e., people, organizations, locations) with data integrated from different external sources, such as the New York Times. The limitation of this approach is that the evaluator does not have the ability to control the structure and format of the entities that are created, rather she will have to accept them the way they have been extracted from the sources. Our evaluation framework, instead, generates the entities synthetically, in which the evaluator has the flexibility to select the attributes that she is wishing the generated entities to have, their domains, and the distribution of the values of the attributes within the generated dataset. This way, the evaluator has the ability to test the under evaluation entity matching systems in different situations that best fit those with which the matching system is intended to be used in practice.

In schema matching, performance may not be a critical factor since the schemas are typically small related to the size of the instances. However, in entity matching that is mostly about instances, performance is becoming an important factor, especially given the exponential growth on the size of the data that we nowadays observe. As already explained, entity matching may be used for run-time integration of web data, or for query answering in an entity repository. Thus, the size of the data will largely affect the entity matching task. For this reason, evaluating the matching tools with data of different sizes is of major importance in order to understand how they scale in terms of time. In this context, the OAEI [33] instance matching track has also this limitation since its dataset is practically static[7]. Instead, in our tool we have the possibility to control the size of the dataset we generate. This is in line with other benchmarks, such as TPC-H and STBenchmark[2], and stress test tools, such as Siege[8]. The latter is an HTTP load testing and benchmarking utility, but these are benchmarks for application domains different to entity matching.

The only other work very close to ours that we are aware of and is synthetically generating data for benchmarking entity matching systems is the SWING system[9] [37], which focuses on providing an infrastructure for evaluation of semantic technologies and has been used for evaluating the techniques in the recent OAEI campaigns. Data acquisition is based on retrieving data from existing linked data repositories that are then transformed to serve as test data. Similar to our approach, the designer specifies the transformation that needs to take place and the system is then generating the transformed data to serve for testing. One difference between SWING and EMBench is that SWING is based on Description Logics thus, the expressive power of the transformation specification is the one of the Description Logics used. Instead, our tool uses datalog extended with operation like aggregation and grouping. The second difference is that SWING does not offer any control on the distribution of the attribute values. In particular, the distributions of the values in an attribute will typically follow the one in the sources, and the distribution in the transformed data will depend only on the original distribution and the transformation that was applied to it. However, SWING, similar to us, offers the ability to create datasets of different sizes. In short, despite the fact that the SWING design principles and goals are similar to EMBench, EMBench has more expressive power and offers more flexibility in the specification of the testing data. Finally, the LINQS group at the University of Maryland pro-

[7] The dataset can be downloaded as is from the OAEI web site.
[8] http://www.joedog.org/siege-home/
[9] http://code.google.com/p/swing-generator/wiki/OntologyGenerator/

vides a data generation tool[10] for entity matching but it is aiming noisy references with co-occurrence relationships.

Very recently, the importance of dynamically generated data for the OAEI was presented [71]. However, the method was used to regenerate the dataset that OAEI is using for testing (called Benchmark) and not to allow evaluators of matching systems to tune at run-time the test cases and the data they want to run. In that sense, although the specific effort is heading towards the right direction, it is still generating a static dataset.

3 Entity Matching Systems

Our work is based on the *concept model* [25], a model that is gaining popularity in many different areas, including the Semantic Web [19] and dataspaces [28], and is reinforced by the recent trends towards a Web of data [16, 25, 41, 43]. Its basic data unit is the *entity*. An entity is a data artifact that models a real world object. It consists of a unique identifier and a set of attributes. Each attribute has a name and a value and describes some characteristic of the real world object. The value of an attribute can be an atomic value or an entity identifier. The latter allows the modeling of relationships among entities.

More formally, we assume the existence of an infinite set of entity identifiers $\mathcal{O}$, an infinite set of names $\mathcal{N}$, and an infinite set of atomic values $\mathcal{V}$.

Definition 1 *An* attribute *is a pair* $\langle n,v\rangle$*, with* $n\in\mathcal{N}$ *and* $v\in\mathcal{V}\cup\mathcal{O}$*. Attributes for which* $v\in\mathcal{O}$ *are specifically referred to as* associations*. Let* $\mathcal{A}=\mathcal{N}\times\{\mathcal{V}\cup\mathcal{O}\}$ *be the set of all the possible attributes. An* entity *is a tuple* $\langle id, A\rangle$ *where* $A\subseteq\mathcal{A}$ *is finite set of* attributes*, and* $id\in\mathcal{O}$*. The latter is referred to as the* entity identifier*. The set of all possible entities is denoted by* $\mathcal{E}$ *and a finite subset of it* $E\subset\mathcal{E}$ *that is closed in terms of associations and contains no two entities with the same identifier is an* entity collection ∎

Since each entity in an entity collection is uniquely identified by its identifier, we will often use the terms *entity* and *entity identifier* equivalently.

Note that an entity is an instance level artifact. It describes some instance level structure and should not be confused with schemas that describe the structure of a collection of objects. The term "attribute" used in the entity definition is similar to the term attribute used in instance objects in object oriented databases, attributes of XML elements and properties of RDF data.

Entity matching is the task of determining whether two entities e_1 and e_2 can be linked one to the other to represent the fact that they refer to the same real world object. If this is the case the two entities are said to *match*, denoted as $e_1\equiv e_2$.

[10] http://www.cs.umd.edu/projects/linqs/projects/er/index.html

Deciding whether two entities match is a challenging task mainly due to the heterogeneity that may be present among them. Another reason is that different representations may be used for the same fact or the same fact may be represented in different forms. Entity matching systems are using the degree of similarity of the two entities to guide their decision on whether they match or not. For the similarity they exploit various techniques, such as syntactic comparison, semantic equivalence, lexical variations, etc. Using this information they compute a score that indicates their belief that the two entities match.

An *entity matching system* is a system that given a collections of entities and an entity e, finds the entity in the collection that best matches the entity e or produces a ranked list of entities based on the matching score with entity e. Entity matching systems can be used for many different purposes. One is *deduplication*, i.e., detecting in an entity collection different representations of the same real world object and merging these representations in one. To do so, the system finds the matching score of each entity with every other entity in the repository. It can then consider as matches those with a score higher than a specific threshold, or produce a ranked list of entity pairs in a decreasing order of their score and let the end user decide how to use that information [45]. Another application is to find structures in two different databases that model the same real world object and use this information for database merging or data exchange [18].

Example 1 *Consider a repository containing an entity collection, a portion of which is illustrated in Figure 1 (ignore for the moment the entity* e_n*) and assume that some deduplication process needs to be run on it. Considering first entity* e_1*, it computes the similarity score with all the other entities. As explained before, the similarity is not only structural but also semantic, syntactic, or may involve any other form of complex reasoning and auxiliary information support. In the specific case, the system will see that entities* e_1 *and* e_2 *are different enough and will produce a low score for that pair, while* e_1 *and* e_4 *are very similar since they have many attributes in common and will assign to that pair a high score. Entity* e_1 *is also similar to entity* e_5 *but the similarity is not that high, thus, a score less than the latter but higher than the first will be produced. Having the scores, it may be decided that only the score between* e_1 *and* e_4 *is high enough to indicate a duplicate representation and perform a merge of the two entities* e_1 *and* e_4 *into a single representation to eliminate the redundancy.*

A third application of entity matching is query answering. A user query is a list of specifications describing the desired characteristics of the entity(ies) that the user is looking for in the form of attribute name-value pairs. Of course, it is not always sure that all these characteristics can be found in

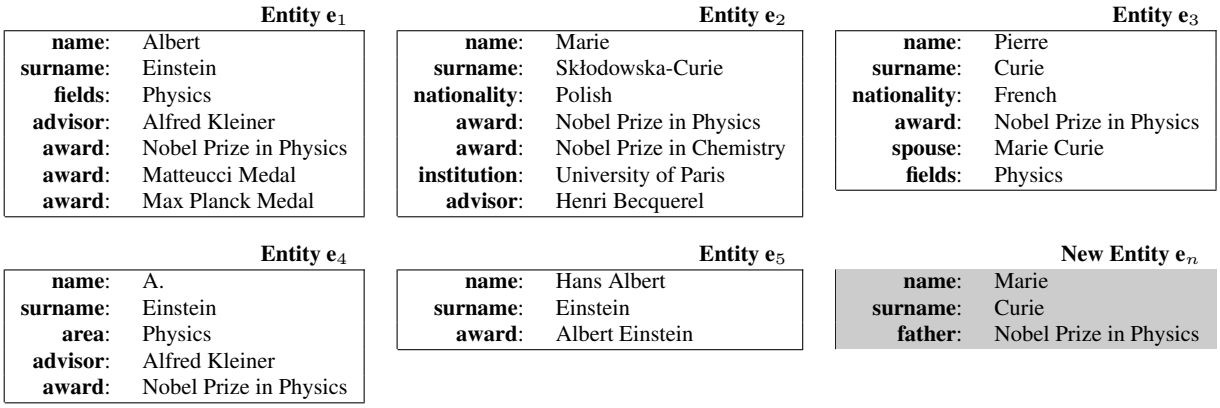


Fig. 1 A fraction of the entity collection of an entity system (entities $e_1, \ldots, e_5$), and an entity e_n modeling the query given by a user.

one entity in the database. Given the fact that users may not have a complete knowledge of the data in the database, or may not be sure of what exactly they are looking for, their queries are often under or over specified [8]. As such, an entity that only partially satisfies the query conditions can be in the answer set, with a score indicating the belief that the specific entity is the one that the user is actually looking for. The answer set is then a ranked list of entities in decreasing order based on the computed score [9, 11]. In some sense, this is based on the foundations of information retrieval, where the user specifies a set of keywords and the system returns the documents that contain as many of these keywords as possible. Despite the many similarities, entity matching cannot be handled as an information retrieval problem. Documents are related to a topic and the keywords are indications of what the topic is. Attributes, on the other hand, cannot be handled as a flat list of keywords. Each attribute has some specific semantic (specified by the attribute name) and a role in specifying the entity it belongs. This means that a different handling is required.

To use entity matching for query answering, we need to be able to reduce the query answering problem into it. The next corollary states that this is possible and its proof explains how.

Corollary 1 *Approximate entity search is reduced to an entity matching problem.*

Proof. Let I be an entity collection and $s_1, \ldots, s_n$ be a set of specifications included in a query q given by the user. A dummy entity e_n is created that contains an attribute a_i for each specification s_i, with i=1..n. The entity is then matched against the entities in the collection I generating a ranked list of entities in decreasing order of their matching score, which reflects the belief that they represent the same real worlds entity as e_n, which actually means the belief that the entities model the real world object the user is looking for. Thus, the results of the entity matching can serve as an answer to the user query. ∎

Due to this equivalent representation of a query as an entity, in the rest of the paper we may use the term entity or *entity request* to refer to a user query. Note that this form of query answering on entity systems is what is already in place in the Syntice Semantic Search Engine [77] and various other applications, e.g., [56].

Example 2 *Consider a user posing the query* $\{\langle$*name, "Marie"*$\rangle, \langle$*surname, "Curie"*$\rangle, \langle$*award, "Nobel Prize in Physics"*$\rangle\}$. *To answer this query the entity* e_n *indicated in gray in Figure 1 is created and matched against the entities* $e_1, \ldots, e_5$ *that are already in the system. The matching task finds that entity* e_2, *although not exactly the same as* e_n, *looks very similar to it and most likely represents the same real world as the one* e_n *represents, thus,* e_2 *gets the highest matching score among the other entities already in the system. Since* e_n *models the user requirements expressed in the query,* e_2 *is returned to the user as an answer.*

4 Entity Matching Evaluation Requirements

A benchmark is defined as "a standardized problem set, or tests that serve as a basis for evaluation or comparison"[11]. Thus, a first step towards the creation of a benchmark for entity matching systems is the creation of a series of test cases that could be used to evaluate and compare the these systems. To understand what these test cases should be, it is important to carefully consider the matching task. According to the previous section, the matching of an entity towards a collection of entities returns a ranked list of matches (or simply the top one). When a matching is performed, the right (expected) answer should be known in advance so that the results produced by the matching system can be evaluated based on how successfully that expected answer was

[11] Merriam-Webster Dictionary

returned. Thus, a test case for a matching system should consist of the input to be provided to the matching tool and the *ground truth*, i.e., the expected correct answer. If the right response is returned, it means that the matching system is capable of dealing with the heterogeneities that exists between the entities in the collection and the entity that it was asked to match.

The more heterogeneities a matching system can successfully handle, the better. Hence, the set of test cases should be *rich* enough to allow the evaluator of the matcher to understand the heterogeneities the under entity matching system can handle, and discriminate it from other similar matching systems.

Of course the test cases should also be *correct*, i.e., the ground truth that is accompanying them should correctly include the matching entities.

The set of test cases should have *no redundancy*. There is no reason of having multiple tests for the same heterogeneity. One is enough to identify whether the entity matching system can successfully deal with it. On the other hand, the test cases should be *well-justified*, meaning that no test case should be present unless it is for testing some specific entity matching situation.

Furthermore, the series of test cases should be as *complete* as possible. Definitely, there are countless cases that may be met in practice, and a system cannot cover them all. This is the same like a good query language, which although cannot cover all the possible queries that one may want to ask, it is important that its expressive power is such that it covers the majority of the queries of interest.

The test cases should allow the evaluator not only to realize the heterogeneities that the matcher can handle, but also to comprehend in what degree of heterogeneity this handling is possible. Thus, it should be able to *configure and scale up* the various test cases.

Finally, in order to be able to compare different systems, it is natural to require that they are all tested on the same test cases. This is not an issue if the test cases are static. However, when the test cases are dynamically generated, it is important that they are the same independently of the time they were generated or the hardware that was used to generate them, assuming of course that the configuration parameters, if any, are the same. This guarantees a fair comparison among entity matching systems that have been evaluated in different architectures, at different times and by different persons. We refer to this desired property as *consistency*.

5 Entity Matching Scenarios

To generate in a systematic way test cases with the properties mentioned in the previous section, we introduce the notion of a *scenario*. A scenario consists of some input to an entity matching tool alongside the ground truth. Recall that given a collection of entities, and an entity e_r, an entity matching system can find the entity in the collection of entities that is best matching e_r.

We have decided to keep the information in a scenario to the minimum, i.e., not to include any additional meta information in order to increase the *discriminating power* of our evaluation mechanism. Consider, for instance, a scenario in which the best match for an entity e_r is requested among the entities in a collection I. Let e_c be one of the entities in the collection with the attribute values of e_c being synonyms of the attribute values of e_r. Clearly e_c is a good match for e_r, and is the ground truth for the scenario. Assume that the collection and the entity e_r are provided as input to two different matching systems, one that has the ability to use WordNet as auxiliary information and another that has not. The first will successfully recognize the match, while the second will not, indicating the higher capabilities of the first regarding synonyms. If the synonym information had been provided as part of the input, the second system would have been able to exploit it and also successfully identify e_c as a match.

Definition 2 *An* entity matching scenario *is a tuple* $\langle I, e_r, e_c\rangle$ *where* I *is an entity collection,* e_r *is an entity and* $e_c \in I$ *referred to as the* ground truth*. The scenario is said to be* successfully executed *by an entity matching system if the system returns the entity* e_c *as a response when provided as input the pair* $\langle I, e_r\rangle$*, i.e., returns* e_c *as the best match of* e_r *in the entity collection* I*.* ■

To guarantee the *correctness* property of the scenarios we consider, i.e., to make sure that the entities e_r and e_c are indeed representing the same real world object, we are creating the scenarios by starting with an entity collection, select an entity, introduce some form of heterogeneity in the entity collection and then test whether the entity matching system can identify as the best match to the entity we had initially selected, its modified version. Of course, we are not stopping with one entity only but we do the above modification to a large part of the entity collection. To implement the above task we introduce the notion of a *modifier*. A *modifier* is a transformation function $f|\mathcal{E}\rightarrow\mathcal{E}$, which is typically an implementation of a certain form of heterogeneity. By applying a modifier on an entity e, we know for sure that e should match $f(e)$, and that any entity matching system that successfully matches them, supports the type of heterogeneity the modifier f introduced.

Driven by this, our proposal for generating an entity matching scenario is to first generate an entity collection, select one (or more entities), and consider it as an entity request. Then apply a number of modifiers to the data such that the structure of the entities is modified. Then consider as an entity collection the modified one and as an answer to the

entity request the entity produced by applying the modifiers on the one that was initially selected as an entity request.

Let I_o be an entity collection of an entity system. A *modified entity collection* is a collection $I_m{=}f(I_o)$. In other words the modified entity collection is the set of entities generated if a modifier f is applied on the entities on I_o. This operation is denoted as $I_o \xrightarrow{f} I_m$. Note that we made no assumption that the modifier f is a total function. This means that there may be entities in I_m that are exactly the same as in I_o. We say that an entity $e_o{\in}I_o$ is the *origin* of the entity $e_m{\in}I_m$, and denote it by $e_o \xrightarrow{f} e_m$, if $e_m{=}f(e_o)$.

Since the result of a modifier is also an entity collection, it can also be used as an input to another modifier. Thus, a modified entity collection may be the result of a series of different modifiers applied on an original collection. The idea of the origin of a modified entity is extended accordingly.

All the above lead to the following steps for generating an entity matching scenario:

1. Consider an entity collection I_o
2. Select an entity $e_r{\in}I_o$
3. Select a series of modifiers $f_1, f_2, \ldots, f_n$
4. Generate the modified entity collection using the modifiers $I_n{=}f_n(f_{n-1}(\ldots f_2(f_1(I_o))))$
5. Select the entity $e_n{\in}I_n$ such that $e_r \xrightarrow{f_1} e_1, \xrightarrow{f_2} \ldots \xrightarrow{f_n} e_n$
6. Generate as a scenario the triple $\langle I_m, e_r, e_n \rangle$

As previously mentioned, not all the test cases can be considered since they are infinite, but only those that have some particular importance. Thus, we focus our attention to those modifiers describing heterogeneities that are commonly used in practice and for which the research community has expressed interest. To do so, we studied the related literature on schema matching [66], mapping [36], information integration [54] investigated in many real applications, and we studied various benchmarks from different areas, such as TPC-H[12], XBenchMatch [31], and most importantly our own STBenchmark [2]. From all these cases that we found in our study we selected a set in a way that we did not choose the same heterogeneity twice, neither heterogeneity that is not met in practice, and at the same time we tried not to leave any common type of heterogeneity out. As a result, we guarantee that out test scenarios have *no redundancy*, are all *well-justified*, and are as *complete* as possible. Specifically, the fact that we tried to include as many modifiers as possible has also helped in boosting further the *discriminating* power of our evaluation methodology.

To ensure that the scenarios are *highly configurable*, we have identified for each scenario as many as possible critical parameters that an evaluator may be interested in controlling and in the implementation, and although they have some default values, we allow them to be modified. This applies also to the creation of the original entity collection. To ensure higher flexibility on the datasets and the scenarios we will generate, we assume the existence of a set of domains $\mathcal{D}$, with each domain $D{\in}\mathcal{D}$ associated to a name $n{\in}\mathcal{N}$, where $\mathcal{N}$ is an infinite set of names. The entity collection I_o is then created by considering a set of entities E such that for each attribute $(n, v){\in}A$ of an entity $\langle id, A\rangle{\in}E$, the $v{\in}D$ and $D{\in}\mathcal{D}$ with n being the name of the domain D. The values and the attribute names are selected randomly through a random selection function, but according to some distribution (explained in Section 6.2).

Finally, to guarantee the *consistency* of the scenarios generated dynamically by the system, since the generation algorithms are deterministic, we have to make sure that any random selection will generate the same random sequence if run on different machines and/or at different times, provided of course that the configuration parameters are the same. This is achieved in the implementation by using random generation functions that are known to generate the same random sequence (given the same seed).

5.1 Entity Modifiers

For the heterogeneities we have identified in our study, we have created the respective modifiers. We present next the types of heterogeneities that the modifiers implement.

A. Syntactic Variations. This category includes variations in the syntax of the actual value of an attribute or the attribute name. They are a consequence of the different ways that a value can be written in real life, without any alteration of its meaning, or a result of human errors. In particular, the category includes:

1. *Misspellings*. A common situation met in practice, especially when humans have manually entered the data, for instance, in the case of web forms.
2. *Word permutations*. When a value consists of more than one words and the order of the words is not critical for the meaning of the value, the words can be found in different order. A classical example is the last/first name values, e.g., “Barack Obama” vs. “Obama Barack”.
3. *Aliases and Different Standards*. Different standards in the way values are represented, is causing this large group of variations. For instance, “George Bush” vs. “George W. Bush”, or “2005 - 30 Charles Str.” vs. “30 Charles Str., Apt 2005”.
4. *Acronyms, Initials, and Abbreviations*. The modern user is overwhelmed with information both in his professional and private life. Acronyms, initials, and abbreviations are a convenient mechanism to speed up the data communication process. The examples are count-

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

less. Conference names are a classical one, e.g., "ISWC" instead of "International Semantic Web Conference".
5. *Homonymity*. A very different problem is this of homonymy. An attribute may have the same name or value but this does not necessarily mean that they are the same. Thus, in entity matching, the system should be able to see through these similarities and distinguish the two entities. As an example, drawn from DBLP, there are many authors with an identical name, and the available attributes about the authors in DBLP are not enough to distinguish between then. Often, exploiting their collaboration network is an alternative that would allow their disjunction (i.e., collective matching, discussed in Section 2.1).

B. Structural Variations. Variations may also exist among the attributes of the entity. This makes two entities that represent the same real world object to vary significantly. Reasons for variations are:

1. *Use of multiple attributes*. Some entities may use a set of attributes to describe some information while others use one attribute. Classical examples are the human names, that may be split into first name and last name, or may not, and the addresses that may be stored as one string, or may have the street, city, zip and country code as separate attributes.
2. *Missing values*. In certain cases, the users may want to specify that an entity has some characteristic but the value of this characteristic is not known or important, and vice-versa. This is usually done by having attributes with no value or no name. For instance, one may know that an entity representing a person has a husband attribute, but is not known who that husband is. Similarly, we may know that a person John is related to a person Mary, but the exactly relationship is not known. In that case, the entities representing the two persons will be associated through an attribute (on John) that has a value (Mary) but no name.
3. *Underspecified entities*. Due to lack of information or any other reason, certain entities may have very few attributes to a point that it is hard to identify them, since they have no attributes to satisfy the entity request attributes.
4. *Overspecified entities.* In entity matching it is important to choose the entity that best matches the expectations as described in the entity request given by the user. However, it may be the case that more than one entity satisfy the request conditions, maybe all of them. In this case, the entity matching technique should be able to choose the entity that most likely is the one that the user is looking for.

C. Semantic Variations. Even if the values of the attributes are the same, their meaning may not be. This is happening because the same word is often used to represent different things. The same applies in the opposite direction. Different words may represent the same concept. More specifically, we consider the following:

1. *Synonyms*. Synonyms are inherently present in every text. The size and popularity of WordNet[13] is a clear testimony of the pervasiveness of synonyms in real life.
2. *Multilingualism*. The globalization has reached unprecedented levels and people of different cultures and languages are often required to collaborate or communicate. Unavoidably, their backgrounds leads to the use of different words (i.e., from different languages) for describing the same thing [76]. For example, it is not rare the case of finding some product over the Internet where part of the vendors describe its color as "black" while others as "noir".

D. Evolution of entities. Entities do not remain static in general. They do evolve. The evolution may involve changes in their attribute values, elimination of attributes, addition of new attributes, etc. As explained in [61, 63], Web 2.0. applications especially focus on enabling and encouraging users to constantly contribute and to modify existing content. An analysis of DBPedia revealed that the data describing the entities were modified in time, with only some of the data remaining the same. Furthermore, they may also split or merge with other, a form of semantic evolution [69].

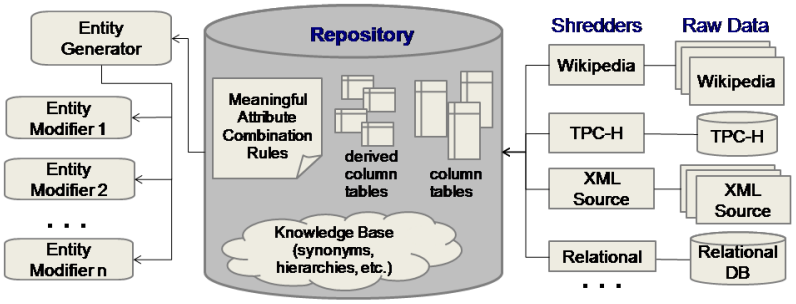
E. Association Network Variance. Entities are connected to each other forming a network of associations. To identify (and distinguish) one entity from another, their network plays an important role. There are already numerous studies of this kind [29, 46, 49].

When this method is used simply as described, it will not take into consideration real world knowledge. This means that algorithms that take real world knowledge into consideration will not have a special advantage when compared with the goal of testing other features. However, when the evaluator wants to test how a matching algorithm performs with respect to real world knowledge, she has the ability to do so by adjusting the entity generation rules accordingly.

6 The EMBench System

We have built a system, called EMBench, that implements the ideas of the previous section. It accepts as an input a set

[13] http://wordnet.princeton.edu



Column Tables

Disease	4,003	Company	96,609
Software	7,768	Band	2,888
Film	52,979	FeminineName	49,350
Mountain	9,811	MasculineName	74,161
Song	110,434	Surname	325,809
University	11,817	Newspaper	3,338

Derived Tables

FirstName = $FeminineName
FirstName = $MasculineName
Name = $FirstName + " " + $Surname

Fig. 2 The general architecture of EMBench alongside the flow of the data among the various components (left), and part of the data in the column tables alongside some rerived table definitions as found in its current implementation (right).

of configuration parameters and generates a series of test-cases for evaluating an entity matching system. The general architecture of the system is illustrated in Figure 2. It consists of three main components: (i) A data repository to be used in the construction of the entity collections of the scenarios, alongside the components (called shredders) that populate this repository from various sources of known importance and quality; (ii) An entity generation engine that composes the data in the repositories to formulate an entity collection; (iii) A set of modifiers that modify in various ways the data in the entity collection and construct a new entity collection with a high degree of heterogeneity.

6.1 Repository

The first part of EMBench is the part dealing with the collection of the data. We do not want the synthetic data we are generating to be completely random strings but we want them to be real world values. For this reason we have introduce the so called *shredders*. A shredder is a software component that takes a database (i.e., relational, XML) and shreds it into a series of column tables. There are general purpose shredders for relational or XML databases, but there are also shredders specifically designed for many popular database that are freely available, like Wikipedia[14], DBPedia[15], Amazon[16], IMdb[17], DBLP[18], OKKAM[19] and Lyrics. The evaluator has the ability to select what databases to be shredded, and add additional databases if desired by supplying the respective shredder, or by using the general purpose that comes with the system. The outcome of the shredding process is a set of column tables that are actually representing the attribute domains (ref. Section 5).

The column tables may have repetitive, overlapping, or complementary information, thus, there is a need for some cleaning or management. For instance, it is possible that two different sources have city names information. Through the different shredders, they will end up into the creation and population of two different column tables. However, it is desired that we have only one domain for city names, thus, it is desired that these two column tables be merged (union) into one. This will also eliminate duplicate values that may exist among them. It is also possible that the values of one column table are separated into different tables, e.g., in the case of the name that is stored as a concatenation of the first and the last, one may want to extract the first name into one new column table and the last name into another. To deal with these issues, EMBench is equipped with a set of predefined rules that can be further extended by the user. These rules specify how the values of the column tables are to be combined together or modified and guide the creation of a new set of column tables, referred to as the *derived column tables*. Note that a derived column table may be created through an identify function rule, meaning that it is considered a derived table without any modification.

Of course, there is no need to shred the original sources, or to create the derived column tables every time the benchmark needs to be run. Once they are created, they remain in the repository until deleted or overwritten. In Section 8 we provide a list of some of the column tables that are included by default in the current version of the EMBench.

Example 3 *Entities describing people are frequently maintained in systems. For generating such entities, we need to populate our repository with related data. This is achieved using the shredders. The result is a set of column tables with individual atomic values related to people, such as* $FirstName$, $Surname$, $Occupation$, *etc. According to a set of predefined combination rules, EMBench then populates the derived tables. For instance, rule* $Name$=$\$FirstName$+" "+$\$Surname$ *populates table* $Name$ *by concatenating values from* $FirstName$ *with values from* $Surname$. *Value selection is explained in the following paragraphs.*

[14] http://www.wikipedia.org/
[15] http://dbpedia.org/About
[16] http://www.amazon.com/
[17] http://www.imdb.com/
[18] http://www.informatik.uni-trier.de/~ley/db/
[19] http://www.okkam.org/

6.2 Entity Generator

Having the domains created, i.e., the derived tables, the initial entity collection can be produced. The entity collection is generated by creating entities with attributes selected from the derived column tables (the attribute name is the name of the derived column table and the value is one of its values.) However, we would not like to generate completely randomly the entity collection, but would like to have some control over it. For this reason we do the following. We decide the number of entities that we need to create, say N, and the maximum number of attributes we expect them to have, say M. We now make M random selections from the pool of derived column tables, each time simply selecting one derived column table. If we do not want to allow the entities to have multi-value attributes in the entities, e.g., multiple attributes with the same name but different value, we make sure that the M selections we make have no repetitions, i.e., we never select the same derived column table twice. In the sequence, we construct a relational table $R_o[A_0, A_1, \ldots, A_M]$, where the attribute A_k is the name of the derived column table we selected in our k-th of the M selections. Next we populate the table R_o with data, by selecting for each attribute A_k, N values from the respective derived column table, and inserting them in attribute A_k of the table R_o. At the end of this process, the table R_o has N tuples. When selecting the N values from the derived column table, we have two options. Either we always do a random selection, meaning that there may be some repetitions, or we do a random selection without repetitions, or we can select values that follow the Zipfian distribution. The way this is achieved is through a small pre-processing of the column tables. Basically, assuming that a column table has K tuples, we generate K integer values that are following the Zipfian distribution and we assign them to the respective K tuples of the column table. We refer to this number as the *repetition number*. Then during the random selection of the values from the derived column tables, when a value is retrieved that has the number j assigned to it from the previous step, it populates j tuples in the R_o table instead of only one.

We can now generate an entity collection I_o of N entities by constructing an entity for every tuple of the populated table R_o. Each such entity will have M attributes, one for every of the M attributes of the table R_o. This task is performed by the *entity generator* module in Figure 2.

Having all the entities in the entity set having M attributes, and actually all of them with the same name, is not very natural. For this reason, before generating the entity collection I_o, we perform an iterative step in which we nullify a number of attribute values of the tuples in R_o. When a i-th tuple has a null value in its j-th attribute A_j, it is considered that the entity e_i generated by that tuple has no A_j attribute. The number of values that are nullified is specified by the evaluator in the configuration parameters of EMBench. If not, a 30% default value is considered.

Example 4 *Consider that we now wish to generate entities representing people based on the column tables mentioned in Example 3. Let us now suppose that each entity will need to contain an occupation attribute (with values taken from the corresponding table and following a Zipfian distribution. The latter implies that the majority of the attributes would appear few times (e.g., "pilot", "parliament member") and only a small number of the attributes would appear many times (e.g., "administrative assistant"). Zipfian distribution is also applied on rules, for instance over the $Name$ rule (Example 3) it would generate names in which the values from $FirstName$ would appear many times (e.g., "John", "Marie") and only a small portion of the values would be rare (e.g., "Odysseus").*

Note that the use of relational tables is only an implementation choice that does not affect in any way the expressiveness of the data model of the entities.

6.3 Entity Modifiers

With the original entity collection I_o been created, the modified dataset I_m is produced next. This is achieved by running the table through a series of *modifiers*. A modifier is a software component that implements a modifier function as explained in the previous section. The input to a modifier is a table like the R_o and the output is a table with the same schema and number of tuples but with modified values. At the end of the process, the modified table R_m is used to generate an entity collection which plays the role of the modified entity collection I_m described in the previous section.

What modifiers are used, in what order and how much each will modify the table is something that is specified by a set of configuration parameters. These parameters have some default values in the system but can also be modified by the user.

With the original and the modified entity collections I_o and I_m in place, the scenarios can be created. To do so, an entity e_r is selected from the former collection, say the one corresponding to the i-th tuple of the table R_o, alongside the entity e_c which is the one corresponding to the i-th tuple of the modified table R_m. The generated scenario is the $\langle I_m, e_r, e_c \rangle$. The above steps are repeated to create multiple scenarios for the same entity collection. If more scenarios need to be constructed for different entity sets, the whole process can be repeated from the beginning. The number of times the whole process is to be repeated and the number of scenarios that are to be generated each time, is again specified in the configuration file of EMBench by the user otherwise some default values are used.

The generation of scenarios that test only one specific type of heterogeneity can be achieved by allowing only one modifier to be applied to the original table R_o each time.

Each modifier needs its own specific implementation. Modifiers can be easily added in the system as independent Java classes, and the user needs to specify their configuration parameters in the global configuration file of EMBench. We describe here a few details on how the modifiers already included in the system have been implemented.

Introducing Contradictory Features. For the misspellings introduction task, we assume the existence of set A, which includes all letters of the English alphabet in upper and lower case. Let symbol |world| denotes the length of a word, and that $random(1, l)$ is a function that returns a random position from 1 to l. If i is the position of a character that should be misspelled, then word$[i]$ corresponds to the character itself. The possible misspelling transformations are:

1. *insertion* of a character randomly selected from the alphabet (i.e., $A[random(1, |A|)]$) into a randomly selected position in the word (i.e., word$[random(1, |\text{word}|)]$),
2. *deletion* of a randomly selected character from the word,
3. *substitution* of a randomly selected character from the word with a character randomly selected from the alphabet, and
4. *permutation* of existing neighbor characters that are randomly selected from the word.

Note that for misspellings, the user also provides a the misspelling rate m. This means that the selected transformation is not applied only once on the given word, but a total of m times.

For the word *permutation* task, the steps are similar to the permutation of characters within a word. We first randomly select two words from the text, and swap them. For introducing *abbreviations* we randomly select a position in the word, and then delete all characters from that position until the end of the word, and replace them with a "." character. A similar process is followed for *acronym* generation, but in this case we delete all characters after the first character of the randomly selected word(s).

For *synonyms* and *multilingualism* we use dictionaries and thesauri. For this, EMBench requires two mappings: one from the words to the synonyms, and another from the words to the word in the other language. This information is maintained in a relation database. Once the process randomly selects a word from the text, then this mapping is used for identifying the replacement, and then we substitute the picked word with the one derived from the mapping.

Introducing Structure Variations. To achieve this kind of data variation, the modifier is first selecting randomly the entities in the entity collection on which it will apply the changes. The number of entities that are selected is a configuration parameter. For each of the entities, the modifier then chooses whether it will perform an attribute deletion, or an attribute split/merge. The attribute deletion task is similar to the word deletion described previously. It randomly selects an attribute among all the attributes of the selected and nullifies it. For the case of *merge* the process first selects two different attributes from the entity merges them under a third one, and for *split* the process selects one attribute into two.

Introducing Variation between Entities. Variations between entities are realized through the *underspecified* transformation and the *overspecified* transformation. The first creates entities that do not contain enough information (i.e., attributes) for allowing high effective matching. The latter creates entities containing information that is not really needed and which should be detected and "ignored" during the matching process.

The process for both transformations uses a variation percentage parameter that is given by the user. This percentage is multiplied with the number of attributes of the given entity, resulting in number x that is between 1 and the maximum number of the entity attributes. The underspecified transformation, randomly selects x attributes and nullifies them, i.e., removes them from the entity. For the overspecified transformation, EMBench generates x new attributes that are included in the entity. This is implemented by simply extending the table R_o described in the previous section with new attributes.

Dealing with underspecified or overspecified entities, is quite a challenging task for entity matching [59], and especially for entity search systems as these allow users to include queries (describing entities) which in most situations correspond to underspecified entities. It is therefore not really required to combine underspecified or overspecified transformation with any other transformations, although in EMBench this is possible.

Monitoring Entity Evolution. The entity evolution modifier sees evolution in two levels. One is the attribute level in which an entity updates its values. For instance, the person named "Jacqueline Lee Bouvier"[20] was later known as "Jackie Kennedy", and then as "Jacqueline Onassis". To do so, the modifier selects randomly an entity and an attribute within, and modifies its value. It may also nullify it instead, to model the case that an entity had some characteristic that it lost.

The second level is the conceptual evolution, basically the split/merge operations. For this the modifier selects an entity, removes it and inserts two new. Each of the two new entities has a subset of the attributes of the one that was removed. For the case of the merge, two entities are removed from the entity set and a new one is inserted, the attributes of which are the union of the attributes of the two entities that

[20] http://en.wikipedia.org/wiki/Jacqueline_Kennedy_Onassis

were removed. In the case in which there are conflicts, i.e., two attributes with the same name but different values, one of the two values is randomly selected. One difference from existing terminology evolution systems [79] and versioning systems [68] is that we do not maintain any timestamp information to achieve conceptual evolution [20].

Supporting Association Network Variance. Recall that an association between entities is modeled by using the id of the second as a value in one of the attributes of the first. EMBench supports modifications to the way the entities are associated between them. The way it does so is by selecting the values that are ids and shuffling them around alongside the elimination of some and the introduction of some new. However, entity matching techniques that use this information are based on the assumption that the data is pretty stable [29, 46, 49, 50]. If the associations are modified, then these techniques will not work. What EMBench can contribute in this part is to measure the sensitivity of these techniques on the data modifications.

Example 5 *Consider the entity collection shown in Figure 1, and now assume that the abbreviation modifier is applied on it. A possible output over the attribute* ⟨*institution,"University of Paris"*⟩ *is* ⟨*institution,"Un. of Paris"*⟩*. Applying the structure variation modifier over attributes* ⟨*name,"Pierre"*⟩ *and* ⟨*surname,"Curie"*⟩ *may result to the attribute* ⟨*name,"Curie Pierre"*⟩*. The latter can would also result to* ⟨*name,"C. Pierre"*⟩ *if the acronym modifier is also applied on it.*

7 Usage of EMBench

EMBench offers three main functionalities. The first is to create a source repository by importing data using shredders. The second is to generate entity datasets using the data from the source repository. The third functionality is to evaluate matching algorithms. To ease the use of these functionalities, EMBench is in general fully parametrized through a configuration file. In addition, EMBench is accompanied with a user interface that allows the specification of the parameters that build the configuration file on-the-fly and run EMBench. In this section we elaborate on each of the three functionalities, and the following URL provides an online access to the system as well as the binary file for a local execution:

`http://db.disi.unitn.eu:8282/embench/`

Note that although the main purpose of EMBench is of course generating synthetic data for evaluating entity matching algorithms, as a by-product, it can be used for two additional purposes. One is the creation of a data warehouse from real data sources, which can be done using the shredders. The second is the generation of large synthetic entity collections, i.e., sets with various entity types, on which no modifiers are applied.

Importing Data in the Repository. The first task is to import data in the repository. For this task, users can either utilize the data that are induced in the default implementation of EMBench (Figure 2), or collect data from the existing sources. The latter can be achieved using the EMBench's shredders. In some sense, the shredders create an image of the real database in a column-store format. By selecting which shredders to activate, the EMBench users can control which sources should be copied and which not. The generic shredder that the EMBench comes with can be easily extended to support additional sources. In addition, EMBench comes with a small collection of shredders, which include a generic shredder for a relational database, and shredders for known systems (described in Section 6.1).

To perform the data collection and shredding part, assuming that the shredders are implemented and in place, the user needs to simply select from the GUI what shredders should be used and the parameters of the sources from where they will retrieve the data, such as the communication protocol and the Internet address.

Configuration and load of the source repository will be typically performed only once, as the data stored in the repository are not altered by the other functionalities of EMBench. And if the users are satisfied with the data incorporated in EMBench's default implementation (discussed in Section 8), then the efforts are really minimal.

Generating Entity Collections. Once the collected data is in place the user has to specify how the column tables should be combined to generate the initial and the modified entity collection. Figure 3 shows a snapshot of the interface with the configuration parameter fields. Two configurations are required for generating the entity collection. The first involves defining the schema for the entities to be generated; in particular the maximum number of entity attributes, the distribution of the values, the percentage of non-existing attributes, etc. All these are done by the configuration interface. For example, as shown in the figure, the first set will contain from 45,000 to 50,000 "publication" entities. Each entity will have the following attributes: a title, a conference, from 2 to 5 authors, and a year. The second part of the configuration involves the modifiers that are to be used and the parameters of each one. As also shown in the figure, users first select a modifier and then specify its parameters.

Note that there can be multiple templates in a single generation (option "Add Entity Type" Figure 3). A template is basically one run of the EMBench as we have described it so far. However, in many cases it is desired to generate different groups of entities with each group having different characteristics. The user has the ability to introduce as many templates as desired (finite number though), and for each one to

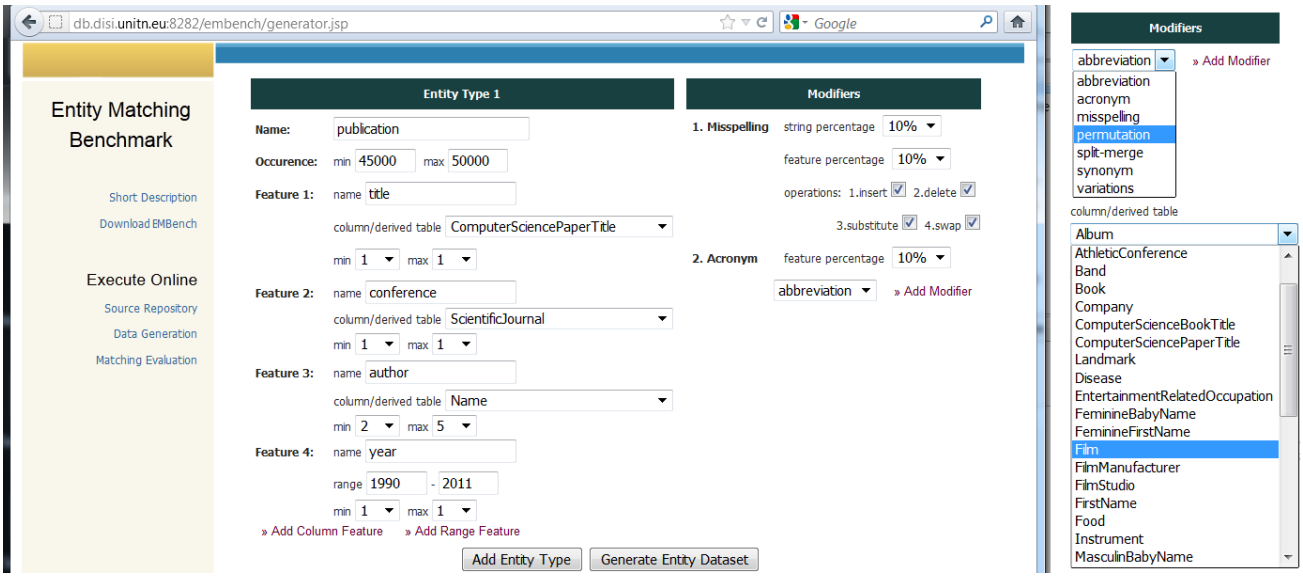


Fig. 3 Screenshot of EMBench GUI for generating an entity collection.

specify the parameters for the generation of the respective entity collections.

With the configuration parameters set, the user can finally ask EMBench to run. The output is a set of entity collections, with each entity collection accompanied with it's modifier version. Thus, every entity has two versions: the one that contains the original data and another that contains the modified data. In addition, the user also receives a list that provides the sequence of modifiers that were applied on each entity.

Evaluating Matching Algorithms. Of course, since we need to evaluate entity matching algorithms, we need to quantify the matching success. A straight forward approach is to compare the collection containing the original entities with the collection containing the modified entities. There are already various metrics of this kind in the literature. Some of the well-known, also included in the default EMBench implementation, are the following:

1. *Precision* is the percentage of the successfully detected matches divided by the number of all returned matches.
2. *Recall* is the percentage of the successfully detected matches divided by the number of matches that should have been found by the algorithm.
3. *F-measure* is the harmonic mean of precision and recall, defined as: (2×precision×recall)/(precision+recall).

As already discussed, the above metrics are not the only available options. An excellent collection and discussion of the different metrics can be found here [35] and here [6]. To allow that incorporation of additional metrics in EMBench, we did not restrict the implementation to the specific metrics but allow users to easily incorporate additional metrics.

One of the issues with the metrics included in EMBench is that they are too strict on the success or not of the matching process. In particular, many entity matching tools may return an ordered list of possible entities as an answer. If the expected entity is not in the first position, the matching tool is penalized the same independently of whether the expected entity is in the second, third, fourth position, or is not returned at all. To alleviate this problem, the metrics that have been already included in EMBench will have to be adjusted to take into consideration the position of the expected entity in the result set.

In any case, the metric that is to be used to evaluate the success of the entity matching tool for a scenario is highly related but orthogonal to our work. The goal of our system is to generate the right set of test cases, with which any kind of metric can be used.

To evaluate a matching algorithm, the user need to extends the EMBench's evaluation class and implement two methods. The first method is for constructing entities as needed by the specific matching algorithm given the data from the EMBench. The second method is for comparing two entities and deciding if there are a match or a non match. The user then executes EMBench and receives the evaluation results.

8 Applicability Experience

We now demonstrate the applicability of EMBench by using it to evaluate different entity matching scenarios. Section 8.1 discusses the use of EMBench for comparing various matching algorithms, and Section 8.1 illustrates the performance evaluation of a single algorithm. Finally, Section 8.3 provides a comparison of EMBench with an existing benchmarking for matching applications, and more specifically with the SWING system that we discussed in Section 2.2.

8.1 Comparison of Matching Algorithms

The first scenario we investigated was using EMBench for comparing various matching algorithms. To demonstrate the applicability of EMBench in this scenario, we needed a small collection of matching algorithms. For this we used SecondString[21], which is an open-source package with approximate string matching algorithms. Cohen et al. [23] performed a comparison between several string matching algorithms, with their results reported in a heavily cited publication (currently with more than 930 citations).

As the authors reported in [23], evaluation was performed over a small set of entity types, such as animals, birds, CORA publications[22], and games, with less than 6,000 entities in the collection with the largest size. We now extend the specific experimental evaluation by comparing some of the string matching algorithms over a larger set of entity types and over an increasing modification level. For this we used the misspelling, abbreviation, and permutation modifiers and configured their feature percentage to 5%, 10%, and 15%.

Table 1 reports the results of the additional comparisons between four string similarity algorithms: Jaro, JaroWinkler, Monge-Elkan, and TFIDF. Execution time was affected when using a different modification level, and thus we only report one execution time per collection and string similarity algorithm. The results clearly indicate that Monge-Elkan requires the most time for processing the entity comparisons, Jaro and TFIDF require almost the same time, and JaroWinkler requires the less time. In addition to time, the table also reports the F-measure of each algorithm for the various modification levels. The JaroWinkler algorithm has the highest F-measure followed by the Jaro algorithm.

In addition to the comparison between algorithms from SecondString, we have also examined the performance of more generic algorithms. We need here to re-emphasize that this paper is not a study of matchers, thus, our goal in this paper is not to dictate the best matcher. There are already many such works in the literature. Instead, we simply wanted to illustrate how sucessfully EMBench can highlight the advantages and the limitations of the matchers it evaluates. For this reason, we do not provide the names of these generic algorithms but we will refer to them as "matcher A" and "matcher B".

For this evaluation we used a dataset with 5,000 publication entities, out of which the 2,500 where modified using the misspelling, permutation, and abbreviation modifiers. Using EMBench we first retrieved F-measure when no modifier was applied, then when only the first modifier was applied, followed by two modifiers, and finally with all three modifiers. We repeated this evaluation using a different order in the execution of the modifiers. Figure 4 shows the results for these executions. Each plot shows F-measure and the modifier(s) that were applied on the dataset. As shown in the plot, every additional modifier reduces the performance of the matching algorithm. However, we can also notice that some combinations are better handled than others. For instance, matcher A is able to better cope with the combination misspelling and then abbreviation, than with abbreviation and then misspelling. This is of course an aspect of matcher A that should be further investigated by the people that created the specific algorithm.

[21] http://secondstring.sourceforge.net/

[22] This dataset was described in Section 2.

		Modification Level			Time
		5%	10%	15%	msec.
Publication	size=10,000				
(title, 3-4 authors, year, venue)					
	JaroWinkler	.941	.927	.921	.026
	Jaro	.919	.904	.898	.025
	Monge-Elkan	.773	.761	.753	.333
	TFIDF	.672	.662	.665	.015
Academic	size=10,000				
(person name, university)					
	JaroWinkler	.833	.837	.836	.033
	Jaro	.779	.781	.782	.030
	Monge-Elkan	.491	.498	.499	.429
	TFIDF	.424	.424	.424	.016
Item	size=5,000				
(name, company)					
	JaroWinkler	.853	.851	.843	.021
	Jaro	.806	.805	.805	.018
	Monge-Elkan	.559	.549	.561	.221
	TFIDF	.44	.44	.44	.014
Song	size=5,000				
(name, band, album, year)					
	JaroWinkler	.955	.954	.937	.014
	Jaro	.936	.935	.916	.012
	Monge-Elkan	.804	.796	.794	.132
	TFIDF	.73	.73	.73	.016
Person	size=5,000				
(name, occupation, 0-1 university)					
	JaroWinkler	.957	.951	.954	.029
	Jaro	.941	.940	.940	.011
	Monge-Elkan	.77	.77	.77	.334
	TFIDF	.650	.650	.650	.014

Table 1 Extending the experimental evaluation of Cohen et al. [23] to a larger set of entity types and an increasing level of modification. The table reports F-measure plus the average time of entity comparisons.

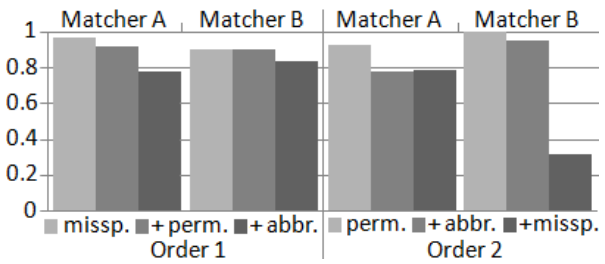


Fig. 4 F-measure for two different sequences of modifiers.

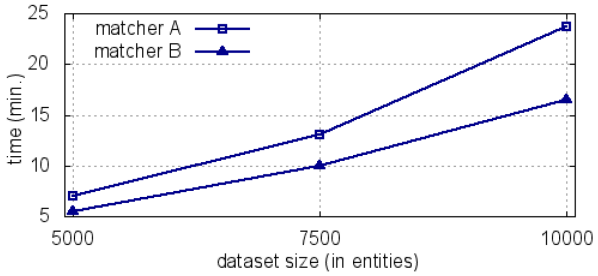


Fig. 5 Execution time for different entity collection sizes (i.e., dataset).

The final evaluation focused on testing the capabilities of the matching algorithms with respect to the size of the dataset, i.e., the number of entities contained in the collection. This is important in entity matching since entity matching takes place at the instance level and entity collections may scale up to thousands or even millions of entities, especially after the advent of big data in our lives. To create collections with different number of entities, we used a publication collection with 5,000 entities and then included additional entities. Following this process, we created two additional collections, one containing 7,000 entities and another containing 10,000. The modifier was executed on the 2,500 entities from each collection. On these three collections we applied the two matchers. Figure 5 plots the time that each matcher required for these three collections. As expected, when the collection size is increased the required time is also increased for both matchers. However, we can clearly see that matcher A can better handle a collection with larger size than matcher B, since the time increase for matcher A is less than for matcher B.

8.2 Measuring a Matching Algorithm's Performance

In this scenario we tested the performance of a single algorithm. More specifically, we use the RDFsim algorithm [48] introduced for detecting semantic-aware near duplicates among data integrated from various sources and applications. The detected duplicates are then grouped together, merged, or removed, in order to avoid repetition and redundancy, and in order to increase the diversity in the information provided to the user.

As described in [48], the original evaluation for the specific algorithm was based on a prototype that crawled news articles from various agencies (e.g., BBC, Reuters, and CNN) taken from the Google News Web site. The entities, such as people, locations, organizations, and events, from the new article were extracted using the OpenCalais Web service[23]. It is clear that this evaluation captured the use of multiple entity types as well as scalability.

[23] http://www.opencalais.com/

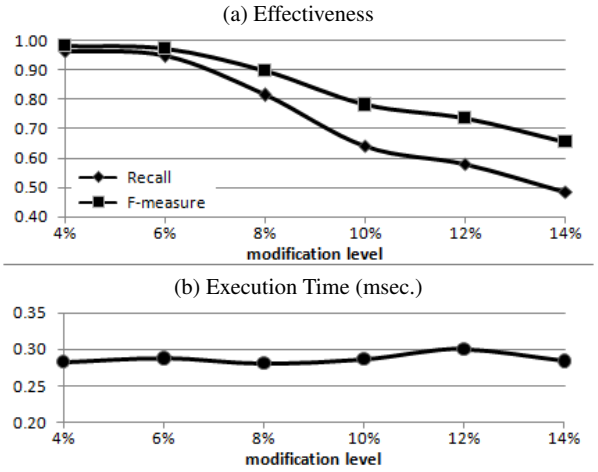


Fig. 6 Extending the experimental evaluation from [48] over entity collections that have an increasing modification level.

One aspect that was not investigated in the original evaluation was the effectiveness of the semantic-aware near duplicate detection when the entities contain different modification levels. News agencies typically provide articles with a sufficiently cleaned content. Also, even if modifications between the entities existed, it was not obvious how to "measure" them in order to investigate the algorithm's effectiveness with respect to modifications.

To investigate this aspect of the algorithm we evaluated the algorithm using EMBench. We generated a collection with 8,000 publications that contained a title, 3-4 authors, a year, and a venue. We then introduced modifications to the publications using misspelling, abbreviation, and permutation. We applied modifications on the collection several times, each time increasing the modification level, i.e., configuring the attribute percentage from 4% until 14%. We then used EMBench to evaluate the algorithm and receive Precision, Recall, and F-measure. Figure 6 (a) shows the results for the last two metrics (Precision was always 1), and (b) shows the execution time. From the results we notice that the execution time is not affected from the modification level. However, it is clear that the effectiveness is reduced when the modification level is increased. This indicates that prior using the specific algorithm on such data collections, one should first improve this aspect of the algorithm.

8.3 EMBench vs. SWING

The last part of the applicability experience section, presents a comparison between EMBench and an existing benchmarking for matching applications. For this comparison, we used the SWING system, which is a semantic Web instance generator that can be use to evaluate matching applications (discussed in Section 2.2).

	EMBench	SWING
Data Acquisition		
supported data sources	any other system	linked data repository
currently in API	DBLP, Amazon, IMdb, DBPedia, Lyrics, & any relational db	Freebase
additional acquisition options	combinations using derived tables	enrichment of the data
quality of resulted data	distinct cleaned values	quality as in Freebase
Data Generation		
entities types	arbitrary, i.e., users can defined by them	based on Freebase classes
entity data	attributes follow Normal or Zipf distribution	retrieved alphabetically from Freebase
entity alignments	generated	generated
Matching Scenarios		
textual variations	misspelling, word permutation, acronyms, initials, abbreviations	word/character modification or addition, gender format, modifications on dates, names, integers and floats
structure variations	attribute merge or split, underspecified and overspecified entities	property/class deletion or addition
semantic variations	synonyms, multilingualism support, support of entity evolution	synonyms
automatic testing	included	included

Table 2 Overview of the functionalities provided by EMBench and SWING.

In order to perform this comparison we used the description and source code of SWING[24]. Table 2 provides an overview of the functionalities included in the EMBench system with the functionalities included in the SWING system. These functionalities are grouped into three categories: data acquisition, data generation, and matching scenarios.

As shown in the table, the two systems have various differences with respect to data acquisition. SWING aims at retrieving data from existing linked data repositories, and, as reported in the description of the [37], only Freebase is used in the current implementation. In contrast, EMBench uses shredders that can connect to any other system for retrieving values. The current implementation of EMBench contains shredders for various systems as well as a generic shredder for relational database.

The second category contains functionalities related to data generation. According to the current implementation, SWING receives two movie titles and then retrieves all Freebase movies those titles are alphabetically in between the given titles. The resulted entities are then used as the basis for SWING's entity collection. This methodology has a few negative aspects. The first is that users can not specify the number of movies that should be included in the collection, but will need to handle all movies returned by the system. Another negative aspect is that SWING assumes that the retrieved entities contain "clean" data and without any duplicates. Given the repository with individual clean values, EMBench is able to generate various entity types that are not covered by the SWING approach. Also, EMBench is able to simulate real world data scenarios through the incorporated distributions.

The last category contains functionalities related to the supported matching scenarios. Here, we see that both systems support similar scenarios with respect to textual variations. However, EMBench supports additional scenarios for the other variations, such as underspecified and overspecified entities.

[24] http://code.google.com/p/swing-generator/

9 Conclusions

In this paper, we have presented EMBench, an implementation of a framework for evaluating entity matching systems through a systematic generation synthetic test cases. EMBench imports data from many different real databases, and based on them generates entity collections of different sizes and different heterogeneities. By tuning its configuration parameters, the users can control the degree of heterogeneity added to the data, and thus stress test the under evaluation entity matching tool. We explained how our tool can be used, and we performed a series of experiments over existing matching techniques for illustrating the kind of information that can be obtained using the EMBench system.

Entity matching has received considerable attention during the last decade, but the research area still lacks a widely acceptable methodology for evaluating and comparing entity matching algorithms. We are hoping that EMBench will allow researchers to better evaluate their matching algorithms, identify the capabilities of the algorithms, and also guide performance improvements on the existing entity matching systems. A wide adoption of EMBench may allow it to be adopted as a norm, paving the way for a standard.

Acknowledgements This work has been partially supported by the EU Project OKKAM ICT-215032 and the ERC Grand Lucretius 267856.

References

1. Aizawa, A., Oyama, K.: A fast linkage detection scheme for multi-source information integration. In: WIRI, pp. 30–39 (2005)
2. Alexe, B., Tan, W., Velegrakis, Y.: STBenchmark: towards a benchmark for mapping systems. PVLDB **1**(1), 230–244 (2008)
3. Alexe, B., Tan, W.C., Velegrakis, Y.: Comparing and evaluating mapping systems with STBenchmark. PVLDB **1**(2), 1468–1471 (2008)
4. Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: VLDB, pp. 586–597 (2002)
5. Andritsos, P., Fuxman, A., Miller, R.J.: Clean answers over dirty databases: A probabilistic approach. In: ICDE (2006)
6. Bellahsene, Z., Bonifati, A., Duchateau, F., Velegrakis, Y.: On Evaluating Schema Matching and Mapping. In: Z. Bellahsene, A. Bonifati, E. Rahm (eds.) Schema Matching and Mapping, chap. 9, pp. 253–291. Springer (2011)
7. Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Whang, S., Widom, J.: Swoosh: a generic approach to entity resolution. VLDB Journal **18**(1), 255–276 (2009)
8. Bergamaschi, S., Domnori, E., Guerra, F., Lado, R.T., Velegrakis, Y.: Keyword Search over Relational Databases: A Metadata Approach. In: SIGMOD, pp. 565–576 (2011)
9. Bergamaschi, S., Guerra, F., Rota, S., Velegrakis, Y.: A Hidden Markov Model Approach to Keyword-based Search over Relational Databases. In: ER (2011)
10. Bergamaschi, S., Guerra, F., Rota, S., Velegrakis, Y.: KEYRY: a Keyword-based Search Engine over Relational Databases based on a Hidden Markov Model. In: ER, pp. 328–331 (2011)
11. Bergamaschi, S., Guerra, F., Rota, S., Velegrakis, Y.: Understanding Linked Open Data through Keyword Searching: the KEYRY approach. In: LWDM, pp. 34–35 (2011)
12. Bernstein, P.A., Melnik, S., Churchill, J.E.: Incremental Schema Matching. In: VLDB, pp. 1167–1170 (2006)
13. Bhattacharya, I., Getoor, L.: Deduplication and group detection using links. In: LinkKDD (2004)
14. Bilenko, M., Mooney, R.: Adaptive duplicate detection using learnable string similarity measures. In: KDD, pp. 39–48 (2003)
15. Bilenko, M., Mooney, R., Cohen, W., Ravikumar, P., Fienberg, S.: Adaptive name matching in information integration. IEEE Intelligent Systems **18**(5), 16–23 (2003)
16. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. **5**(3), 1–22 (2009)
17. Bonifati, A., Mecca, G., Papotti, P., Velegrakis, Y.: Discovery and Correctness of Schema Mapping Transformations. In: Z. Bellahsene, A. Bonifati, E. Rahm (eds.) Schema Matching and Mapping, chap. 5, pp. 111–147. Springer (2011)
18. Bonifati, A., Velegrakis, Y.: Schema Matching and Mapping: From Usage to Evaluation. In: EDBT, pp. 527–529 (2011)
19. Bouquet, P., Stoermer, H., Bazzanella, B.: An entity name system (ENS) for the semantic web. In: EWSC, pp. 258–272 (2008)
20. Bykau, S., Mylopoulos, J., Rizzolo, F., Velegrakis, Y.: Supporting queries spanning across phases of evolving artifacts using steiner forests. In: CIKM, pp. 1649–1658 (2011)
21. Bykau, S., Mylopoulos, J., Rizzolo, F., Velegrakis, Y.: On Modeling and Querying Concept Evolution. Journal on Data Semantics **1**, 31–55 (2012)
22. Cohen, W.: Data integration using similarity joins and a word-based information representation language. ACM Transactions on Information Systems (TOIS) **18**(3), 288–321 (2000)
23. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: IIWeb co-located with IJCAI, pp. 73–78 (2003)
24. Cohen, W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: KDD, pp. 475–480 (2002)
25. Dalvi, N., Kumar, R., Pang, B., Ramakrishnan, R., Tomkins, A., Bohannon, P., Keerthi, S., Merugu, S.: A web of concepts. In: PODS, pp. 1–12 (2009)
26. Doan, A., Halevy, A.: Semantic integration research in the database community: A brief survey. AI Magazine **26**(1), 83–94 (2005)
27. Doan, A., Lu, Y., Lee, Y., Han, J.: Object matching for information integration: A profiler-based approach. In: IIWeb co-located with IJCAI, pp. 53–58 (2003)
28. Dong, X., Halevy, A.: Indexing dataspaces. In: SIGMOD Conference, pp. 43–54 (2007)
29. Dong, X., Halevy, A., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference, pp. 85–96 (2005)
30. Duchateau, F.: Towards a Generic Approach for Schema Matcher Selection: Leveraging User Pre- and Post-match Effort for Improving Quality and Time Performance. Ph.D. thesis, Universite Montpellier II - Sciences et Techniques du Languedoc (2009)
31. Duchateau, F., Bellahsene, Z., Hunt, E.: XBenchMatch: a Benchmark for XML Schema Matching Tools. In: VLDB, pp. 1318–1321 (2007)

32. Elmagarmid, A., Ipeirotis, P., Verykios, V.: Duplicate record detection: A survey. TKDE **19**(1), 1–16 (2007)
33. Euzenat, J., Ferrara, A., van Hage, W., Hollink, L., Meilicke, C., Nikolov, A., Ritze, D., Scharffe, F., Shvaiko, P., Stuckenschmidt, H., Sváb-Zamazal, O., Cássia, T.: Final results of the ontology alignment evaluation initiative 2011. In: OM co-located with ISWC (2011)
34. Euzenat, J., Meilicke, C., Stuckenschmidt, H., Shvaiko, P., Cássia, T.: Ontology alignment evaluation initiative: Six years of experience. Journal of Data Semantics **15**, 158–192 (2011)
35. Euzenat, J., Shvaiko, P.: Ontology matching. Springer-Verlag (2007)
36. Fagin, R., Haas, L., Hernandez, M., Miller, R., Popa, L., Velegrakis, Y.: Clio: Schema mapping creation and data exchange. In: Conceptual Modeling: Foundations and Applications, pp. 198–236. Springer (2009)
37. Ferrara, A., Montanelli, S., Noessner, J., Stuckenschmidt, H.: Benchmarking matching applications on the semantic web. In: ESWC (2), pp. 108–122 (2011)
38. Ferrara, A., Nikolov, A., Scharffe, F.: Data Linking for the Semantic Web. Journal of Data Semantics **7**(3) (2011)
39. Getoor, L., Diehl, C.: Link mining: a survey. SIGKDD Explorations **7**(2), 3–12 (2005)
40. Giunchiglia, F., Shvaiko, P., Yatskevich, M.: S-Match: an algorithm and an implementation of semantic matching. In: Semantic Interoperability and Integration (2005)
41. Halevy, A., Franklin, M., Maier, D.: Principles of dataspace systems. In: PODS, pp. 1–9 (2006)
42. Hassanzadeh, O., Kementsietsidis, A., Velegrakis, Y.: Data Management Issues on the Semantic Web. In: ICDE, pp. 1204–1206 (2012)
43. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2011)
44. Hernández, M., Stolfo, S.: Real-world data is dirty: Data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery **2**(1), 9–37 (1998)
45. Ioannou, E., Nejdl, W., Niederée, C., Velegrakis, Y.: On-the-fly entity-aware query processing in the presence of linkage. PVLDB **3**(1), 429–438 (2010)
46. Ioannou, E., Niederée, C., Nejdl, W.: Probabilistic entity linkage for heterogeneous information spaces. In: CAiSE, pp. 556–570 (2008)
47. Ioannou, E., Niederee, C., Velegrakis, Y.: Enabling Entity-Based Aggregators for Web 2.0 data. In: WWW, pp. 1119–1120 (2010)
48. Ioannou, E., Papapetrou, O., Skoutas, D., Nejdl, W.: Efficient semantic-aware detection of near duplicate resources. In: ESWC, pp. 136–150 (2010)
49. Kalashnikov, D., Mehrotra, S.: Domain-independent data cleaning via analysis of entity-relationship graph. TODS **31**(2), 716–767 (2006)
50. Kalashnikov, D., Mehrotra, S., Chen, Z.: Exploiting relationships for domain-independent data cleaning. In: SIAM SDM (2005)
51. Kopcke, H., Rahm, E.: Frameworks for entity matching: A comparison. DKE **69**(2), 197–210 (2010)
52. Koudas, N., Marathe, A., Srivastava, D.: Flexible string matching against large databases in practice. In: VLDB, pp. 1078–1086 (2004)
53. Legler, F., Naumann, F.: A Classification of Schema Mappings and Analysis of Mapping Tools. In: BTW, pp. 449–464 (2007)
54. Lenzerini, M.: Data integration: A theoretical perspective. In: PODS, pp. 233–246 (2002)
55. McCallum, A., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets with application to reference matching. In: KDD, pp. 169–178 (2000)
56. Miklós, Z., Bonvin, N., Bouquet, P., Catasta, M., Cordioli, D., Fankhauser, P., Gaugaz, J., Ioannou, E., Koshutanski, H., Maña, A., Niederée, C., Palpanas, T., Stoermer, H.: From web data to entities and back. In: CAiSE, pp. 302–316 (2010)
57. Minack, E., Paiu, R., Costache, S., Demartini, G., Gaugaz, J., Ioannou, E., Chirita, P., Nejdl, W.: Leveraging personal metadata for desktop search: The Beagle^{++} system. Journal of Web Semantics **8**(1), 37–54 (2010)
58. Morris, A., Velegrakis, Y., Bouquet, P.: Entity identification on the semantic web. In: SWAP (2008)
59. Mottin, D., Palpanas, T., Velegrakis, Y.: Entity Ranking Using Click-Log Information. Intelligent Data Analysis Journal **17**, 5 (2013)
60. Ontology alignment evaluation initiative (OAEI) co-located with ISWC. http://oaei.ontologymatching.org/
61. Papadakis, G., Giannakopoulos, G., Niederée, C., Palpanas, T., Nejdl, W.: Detecting and exploiting stability in evolving heterogeneous information spaces. In: JCDL, pp. 95–104 (2011)
62. Papadakis, G., Ioannou, E., Niederée, C., Fankhauser, P.: Efficient entity resolution for large heterogeneous information spaces. In: WSDM, pp. 535–544 (2011)
63. Papadakis, G., Ioannou, E., Niederée, C., Palpanas, T., Nejdl, W.: Eliminating the redundancy in blocking-based entity resolution methods. In: JCDL, pp. 85–94 (2011)
64. Papadakis, G., Ioannou, E., Niederée, C., Palpanas, T., Nejdl, W.: Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In: WSDM, pp. 53–62 (2012)
65. Parag, Domingos, P.: Multi-relational record linkage. In: MRDM Workshop co-located with KDD, pp. 31–48

(2004)
66. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB Journal **10**(4), 334–350 (2001)
67. Rastogi, V., Dalvi, N., Garofalakis, M.: Large-scale collective entity matching. PVLDB **4**(4), 208–218 (2011)
68. Rizzolo, F., Vaisman, A.: Temporal XML: modeling, indexing, and query processing. VLDBJ **17**(5), 1179–1212 (2008)
69. Rizzolo, F., Velegrakis, Y., Mylopoulos, J., Bykau, S.: Modeling Concept Evolution: A Historical Perspective. In: ER, pp. 331–345 (2009)
70. Roşoiu, M., Cássia, T., Euzenat, J.: Ontology matching benchmarks: generation and evaluation. In: OM co-located with ISWC (2011)
71. Rosoiu, M.E., dos Santos, C.T., Euzenat, J.: Ontology matching benchmarks: generation and evaluation. In: OM (2011)
72. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: KDD, pp. 269–278 (2002)
73. Shen, W., DeRose, P., Vu, L., Doan, A., Ramakrishnan, R.: Source-aware entity matching: A compositional approach. In: ICDE, pp. 196–205 (2007)
74. Shvaiko, P., Euzenat, J.: Ten challenges for ontology matching. In: OTM Conferences (2), pp. 1164–1182 (2008)
75. Tejada, S., Knoblock, C., Minton, S.: Learning domain-independent string transformation weights for high accuracy object identification. In: KDD, pp. 350–359 (2002)
76. Tsinaraki, C., Velegrakis, Y., Kiyavitskaya, N., Mylopoulos, J.: A Context-based Model for the Interpretation of Polysemous Terms. In: ODBASE, pp. 939–956 (2010)
77. Tummarello, G., Delbru, R., Oren, E.: Sindice.com: Weaving the open linked data. In: ISWC/ASWC, pp. 552–565 (2007)
78. Vaccari, L., Shvaiko, P., Pane, J., Besana, P., Marchese, M.: An evaluation of ontology matching in geo-service applications. GeoInformatica **16**(1), 31–66 (2012)
79. Weikum, G., Ntarmos, N., Spaniol, M., Triantafillou, P., Benczúr, A., Kirkpatrick, S., Rigaux, P., Williamson, M.: Longitudinal analytics on web archive data: It's about time! In: CIDR, pp. 199–202 (2011)
80. Whang, S., Menestrina, D., Koutrika, G., Theobald, M., Garcia-Molina, H.: Entity resolution with iterative blocking. In: SIGMOD Conference, pp. 219–232 (2009)
81. Yatskevich, M.: Preliminary evaluation of schema matching systems. Tech. Rep. DIT-03-028, University of Trento (2003)