Towards a Domain Independent Platform for Data Cleaning

Arvind Arasu Surajit Chaudhuri Zhimin Chen Kris Ganjam Raghav Kaushik Vivek Narasayya
Microsoft Research
{arvinda,surajitc,zmchen,krisgan,skaushi,viveknar}@microsoft.com

Abstract

We present a domain independent platform for data cleaning developed as part of the Data Cleaning project at Microsoft Research. Our platform consists of a set of core primitives and design tools that allow a programmer to develop sophisticated data cleaning solutions with minimal programming effort. Our primitives are designed to allow rich domain and application specific customizations and can efficiently handle large inputs. Our data cleaning technology has had significant impact on Microsoft products and services and has been successfully used in several real-world data cleaning applications.

1 Introduction

Modern enterprises rely on sophisticated data-driven *Business Intelligence (BI)* technologies [6] for their decision making. An integral component of the BI architecture is the *data warehouse*, where data from a variety of sources is aggregated prior to analysis. Similar data aggregation also occurs in offline data processing pipelines of search verticals such as Bing Maps and Bing Shopping [5]. The data provided by sources can have data quality issues; for example, the data can have missing values and typographical errors. Further, different sources can use different representations for the same real-world entity. For example, the state *Washington* can be represented using `Washington` in one source and `WA` in another. It is important to fix such errors and reconcile representational inconsistencies since they can adversely affect the quality of data analysis (in BI) and search (in verticals such as Bing Maps). The process of fixing errors and inconsistencies in data is often referred to as *data cleaning*. Since integration into a warehouse typically involves large data volumes, performance and scale are important considerations in data cleaning.

Data cleaning encompasses a variety of high-level *tasks* such as *record matching*, *deduplication*, *segmentation*, and *null-filling* [16, 11]. The goal of record matching and deduplication is to identify *matching* records, defined to be records that correspond to the same real-world entity. The output of record matching is pairs of matching records while the output of deduplication is clusters of matching records. The goal of segmentation is to extract structured records from unstructured text, e.g., extracting details such as street name, city, and postal code from an address string. The goal of null-filling is to identify missing values in records, e.g., identifying a missing postal code using information from city and state. As we describe later, these data cleaning tasks are related to each other; e.g., an implementation of record matching might use segmentation as a sub-module.

Data cleaning is applicable in a wide variety of domains such as addresses, products, health records, publications and citations, and music meta-data. Most current commercial solutions can be viewed as *verticals* and cater

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

to a single domain. For example, Trillium [18] provides data cleaning functionality only for address domain. The aspirational goal of our Data Cleaning project at Microsoft Research [17] (which has been ongoing since 2000) is to design and develop a *domain independent, horizontal platform* for data cleaning, such that vertical solutions such as the functionality provided by Trillium can be developed over the platform with little programming effort. The basis for this goal is the observation that some fundamental concepts such as textual similarity are part of most data cleaning solutions across different domains, and our platform is designed to capture and support these concepts.

Our current platform, shown in Figure 1, consists of a set of core *primitives* and a set of *design tools*. A programmer develops a data cleaning solution on our platform by composing and customizing one or more primitives, and she can leverage the design tools to identify a good way to customize and compose the primitives. Our primitives include a customizable notion of textual similarity along with an efficient lookup operation based on this similarity called *Fuzzy Lookup*, a clustering primitive called *Fuzzy Grouping*, and a parsing (segmentation) primitive. We have designed our primitives so that they can be customized to a specific domain—e.g., we can customize Fuzzy Lookup to recognize that `Bob` and `Robert` are synonymous while matching people names. Further, our primitives have been designed for high performance and can handle large inputs and complex customizations efficiently. While other research efforts have focused on horizontal approaches to data cleaning (see [11] for a survey), one important differentiator of our approach is the focus on customizability and performance.

Our design tools include a suite of *learning tools* to help a programmer configure and customize a data cleaning solution using the primitives available in our platform. The learning tools rely mostly on labeled examples to navigate the space of possible configurations. Our platform also includes a *Data Profiling* tool that assesses data quality of a source and identifies properties such as keys and functional dependencies satisfied by data. As noted earlier, functional dependencies can be useful for null-filling.

Our data cleaning technology has had significant product impact within and outside of Microsoft [17]: (1) Fuzzy Lookup and Fuzzy Grouping were shipped as part of SQL Server Integration Services (SSIS), Microsoft SQL Server ETL system, in 2005, and they were the first data cleaning technology to appear in SSIS. (2) In the aftermath of Hurricane Katrina, Fuzzy Lookup was used to develop a matching solution that allowed matching the Government's evacuee database with the database of missing people, thereby helping with the search and identification of missing people. Similar solutions have been developed in recent years, e.g., by Google [13]. (3) Data Profiling tool was shipped as part of SSIS 2008. (4) The Bing maps service currently uses Fuzzy Lookup to match a user query against a large database (containing tens of millions of records) of landmarks and locations. Bing maps service has stringent performance requirements and Fuzzy Lookup currently has an average latency of 3 ms per match query. (5) The group within Microsoft that performs *master data management* of the Microsoft's customers currently uses a solution based on Fuzzy Lookup for record matching over customer data. This solution replaced an earlier commercial third party solution after their evaluations showed that it had better performance and quality. (6) The Bing shopping team currently uses the segmentation primitive as part of its offline data cleaning pipeline before product data from various sources is loaded into its warehouse. (7) Fuzzy lookup is also available as an add-in for Microsoft Excel [12].

This paper provides an overview of our data cleaning primitives and design tools and highlights key design decisions. In particular, we focus on Fuzzy Lookup (Section 2) and provide a brief overview of segmentation (Section 3) and design tools (Section 4). We emphasize that this paper is not a general overview of data cleaning and we refer the reader to [11] for an extensive survey. An overview of our project as of 2006 appears in [8].

2 Fuzzy Lookup

The same real world entity could be represented in different ways, e.g., due to typographical errors, differences in conventions, and missing attributes. For example, the real world organization `Microsoft` could be represented both as ⟨`Microsoft Corp, 1 Microsoft Way, Redmond, WA, 98052`⟩ and ⟨`Microsoft`

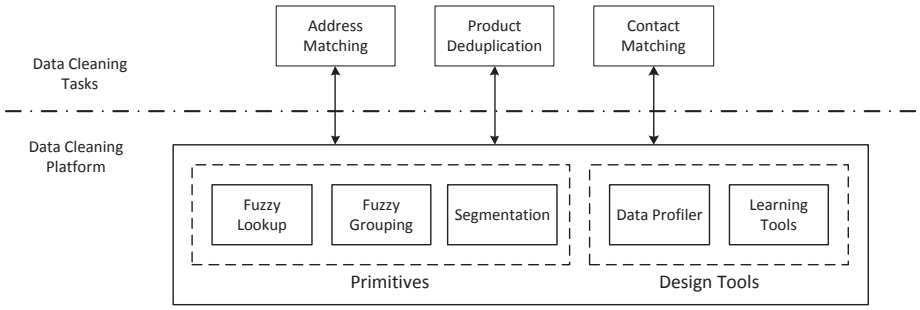

Figure 1: Data cleaning primitives and design tools.

`Corporation, One Microsoft Way, Redmnd, Wash, 98052`⟩. The goal of a record matching task is to identify pairs of matching records that represent the same real-world entity.

At the core of record matching is the notion of a *similarity* function that given two strings returns a numeric value (typically between 0 and 1) quantifying their similarity, a higher value indicating greater similarity. We describe our similarity function in Section 2.1. Fuzzy Lookup supports the following functionality—given a *reference table* R of strings, a query string s, and a threshold parameter θ, find all strings in R whose similarity with s is greater than or equal to θ. To support efficient lookups, Fuzzy Lookup builds an *index* over R. The index structure and lookup algorithm are discussed in Section 2.2.

2.1 Similarity Function

The standard way in which similarity is computed is to use the *textual similarity* between strings using functions such as edit distance, Hamming distance, and Jaccard similarity [11]. Textual similarity is a strong indicator of string similarity. Indeed, the first version of our Fuzzy Lookup primitive [7] relied exclusively on textual similarity. However, in course of our experience in using Fuzzy Lookup with various scenarios including the Hurricane Katrina scenario described above, we discovered the need for customizing the similarity function. For example, there are many cases where strings that are syntactically far apart can still represent the same real-world object. This happens in a variety of settings—street names (the street name `SW 154th Ave` is also known as `Lacosta Dr E` in the postal area corresponding to zipcode 33027), the first names of individuals (`Robert` can also be referred to as `Bob`), conversion from strings to numbers (`second` to `2nd`) and abbreviations (`Internal Revenue Service` being represented as `IRS`). Therefore we modified the design of our similarity function to start with a textual similarity function and extended it to allow a rich customization that enables it to match strings that are syntactically far apart. The current version of Fuzzy Lookup which is used, e.g., in Bing Maps, supports such customizations. We first discuss the core measure of textual similarity followed by the various customizations we expose.

2.1.1 Set Similarity

As our core measure of textual similarity, we use Jaccard similarity. We model a string as a *set of tokens*. The Jaccard similarity between two sets is the size of their intersection as a fraction of the size of the union. The Jaccard Similarity between the two strings `University Avenue, Seattle, WA` and `University Ave, Seattle, WA` tokenized at word boundaries is 3/5. Our rationale for choosing Jaccard similarity is that in addition to being an interesting similarity function in its own right, we can also reduce other similarity functions such as edit distance to it. The intuition is as follows: consider tokenizing a string into it's q-grams. All q-grams that are "far away" from the place where the edits take place must be identical. Hence, a small edit distance implies large Jaccard similarity between the sets of q-grams. For example, consider the strings

Microsoft and Mcrosoft; their edit distance is 1 and the Jaccard similarity between their 1-grams is 8/9. The above relationship between edit distance and Jaccard similarity has been formalized in prior work [14].

In general, we can customize Jaccard similarity by specifying how we tokenize a string. For example, we could tokenize a string into a set of words or a set of q-grams or design language specific tokenizations. We also draw upon the notion of *token weighting* present in previous work on Information Retrieval (IR). Jaccard similarity can be extended to handle token weights by computing a weighted intersection and union. We can use token weights to further customize Jaccard similarity. For example, by assigning higher weights to the tokens Microsoft and Oracle and lower weights to Corporation and Corp, we can ensure that the similarity between the pair (Microsoft Corporation, Microsoft Corp) is greater than that between the pair (Microsoft Corporation, Oracle Corporation). Intuitive weights could be assigned using the IR notion of inverse document frequency (idf).

2.1.2 Transformation Rules

As noted above, there are many cases (e.g., the first names of individuals where Robert can also be referred to as Bob) where strings that are syntactically far apart represent the same real-world object. We posit that it is impractical to expect a generic similarity function to be cognizant of variations such as those of the above examples since they are highly domain-dependent. Rather the matching framework has to be customizable to take the variations as an *explicit input*. Prior work addresses this issue primarily using *token standardization* [16, 11]. The idea is to pre-process the input data so that all variations of a string are converted into a uniform (standard) format. Thus, for example, the input could be pre-processed so that all occurrences of Bob are converted to Robert. However, this approach is inadequate since it does not capture variations that are not equivalences; for example a first initial (such as J) can be expanded in multiple ways (John, Jeff, etc.) that are clearly not equivalent [1].

We therefore introduce the notion of *transformation rules (transformations)*. A transformation rule consists of a triplet $(context, lhs \rightarrow rhs)$ where each of *context*, *lhs*, and *rhs* is a string. (Transformations with empty context are represented $lhs \rightarrow rhs$.) Examples of transformations are: (Bob $\rightarrow$ Robert) and (33027, SW 154th Ave $\rightarrow$ Lacosta Dr E). We now describe how a string s can be transformed given a set of transformation rules T. A rule $(context, lhs \rightarrow rhs)$ can be applied to s if both *context* and *lhs* are substrings of s; the result of applying the rule is the string s' obtained by replacing the substring matching *lhs* with *rhs*. We can apply any number of transformations one after another. However, a token that is generated as the result of applying a transformation cannot participate in any subsequent transformation.

Example 1: We can use the transformation (Drive $\rightarrow$ Dr) to generate the string Lacosta Dr E from the string Lacosta Drive E. However, we cannot further convert Lacosta Dr E to Lacosta Doctor E using the transformation (Dr $\rightarrow$ Doctor).

The set of strings generated by s is the set of all strings obtained by applying zero or more transformations to s. We combine set similarity with transformations to obtain an overall similarity function. Given a set of transformations T, the similarity between strings s_1 and s_2 *under* T is defined to be the *maximum* Jaccard similarity among all pairs s_1' and s_2' respectively generated by s_1 and s_2 using T. We illustrate the above definition through the following example.

Example 2: Consider the strings SW 154th Ave Florida 33027 and Lacosta Dr E FL 33027 and the transformations (FL $\rightarrow$ Florida) and (33027, SW 154th Ave $\rightarrow$ Lacosta Dr E). Each of the strings derives two strings as shown in Figure 2. The overall similarity is the maximum Jaccard similarity among the four pairs of derived strings which in this instance is 1 since both of the above strings generate the same string Lacosta Dr E Florida 33027.

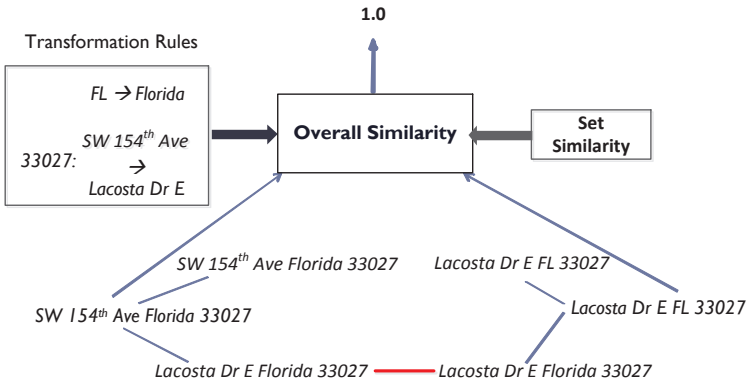

Figure 2: Combining Set Similarity With Transformations.

In general, we allow different sets of the transformations for the reference table and to the input query. Although we assume that the set of transformations is an input, they need not be provided in the form of a table. For example, we could handle transformations obtained by concatenating words (e.g., `Disney Land` $\rightarrow$ `DisneyLand`) programmatically by generating them dynamically at query time. Since the number of possible word concatenations is large, generating them dynamically may be preferable to materializing them ahead of time. Similarly, we could programmatically capture word level edits (e.g., `Mcrosoft`$\rightarrow$`Microsoft`) and abbreviations (e.g., `J`$\rightarrow$`Jeff`). Fuzzy lookup provides an interface where we can plug in user-defined transformations. We next discuss how we address the efficiency challenges posed by our notion of similarity.

2.2 Efficient Lookups under the similarity function

Recall that Fuzzy Lookup provides the following functionality—given a *reference table* of strings, R, a set of transformation rules, T, a query string s, and a threshold parameter θ, find all strings in R whose Jaccard similarity under T with s is $\geq \theta$. To support efficient lookups, Fuzzy Lookup builds an *index* over R. The indexing algorithm is based on the idea of *signature schemes* [3]. A signature scheme, given a string, generates a set of signatures with the property that: if the Jaccard similarity between r and s is greater than or equal to θ, then they share a common signature. A signature based algorithm operates as follows. It generates signatures for each record in the reference table R which are indexed using an inverted index. Given a query string s, the lookup algorithm finds candidate strings r such that the signatures of r and s overlap. In a post-filtering step, the similarity predicate is checked and the matching records are returned.

Several signature schemes have been proposed in the literature for Jaccard similarity (without transformations) [16] including signature schemes developed by us [3, 9]. We now briefly discuss a signature scheme developed by us [9] called *prefix filtering*. Consider a total ordering over all tokens. Given a weighted set of tokens r and $0 \leq \beta \leq 1$, define $prefix_\beta(r)$ as follows: sort the elements in r by the global token ordering and find the shortest prefix such that the sum of the weights of the tokens in the prefix exceeds β times the weight of r. We have the following result [9]: if the jaccard similarity between strings r and s is greater than or equal to θ, then $prefix_{1-\theta}(r) \cap prefix_{1-\theta}(s) \neq \phi$. Fuzzy Lookup supports the signature schemes developed by us as well as the well-known *locality sensitive hashing (LSH)* which is a probabilistically approximate signature scheme. Multiple signature schemes enable Fuzzy Lookup to operate under different precision performance trade-offs.

In the presence of transformations, a naive adaptation of the above algorithms is as follows. Consider the expanded relation $ExpandR$ obtained by applying the transformations to each string in R. Given a query string s, we can similarly generate an expanded set of strings, look up each against $ExpandR$ and return the distinct matching rows. One problem with this approach is that the size of the expanded sets can be prohibitively high when we have a large number of transformations.

Id	*ProductName*
1	ThinkPad T43 Centrino PM 750 1.8GHz/512MB/40GB/14.1"TFT
2	ThinkPad T43 Centrino PM 740 1.7GHz/512MB/40GB/14.1"TFT
3	Lenovo ThinkPad T43 Notebook (1.7GHz Pentium M Centrino 740, 512MB, 40 GB, 14.1TFT)

(a)

Id	*Brand*	*Line*	*Model*	*Processor*	*Processor Model*	...
1		ThinkPad	T43	Centrino PM	750	...
2		ThinkPad	T43	Centrino PM	740	...
3	Lenovo	ThinkPad	T43	Pentium M Centrino	740	...

(b)

Figure 3: Sample Product Records

Example 3: Consider the citation: "N Koudas, S Sarawagi, D Srivastava. Record linkage: Similarity Measures and Algorithms. Proceedings of the 2006 ACM SIGMOD International Conference. Chicago, IL, USA." Suppose that we consider the set of rules {N→Nick, S→Sunita, D→Divesh, SIGMOD→"Special Interest Group on Management of Data", ACM→"Association for Computing Machinery", IL→Illinois}. The number of strings generated by this citation under these rules is $2^6 = 64$.

Since each lookup of a derived string in turn generates signatures that are then looked up, we can improve the performance by only looking up the *distinct* signatures over all the derived strings.

Example 4: Consider Example 3. Suppose the set (of last names) {Koudas, Sarawagi, Srivastava} is a signature generated by all the 64 generated records. This signature need not be replicated 64 times; one copy suffices.

Similarly, in the presence of transformations, even computing the similarity function becomes challenging. The straightforward method is to enumerate all generated strings which is worst-case exponential. For the large class of single-token transformations where both the `lhs` and `rhs` are single tokens (e.g., `St`→`Street`), we show that it is possible to compute the Jaccard similarity between two strings (under the transformations) efficiently (in polynomial time) by reducing this problem to bipartite matching [1]. Using the above optimizations and additional optimizations proposed in our previous work [1], we are able to support highly efficient record matching. For example, over a database of 10 million US landmarks used by Bing Maps, in the presence of a total of 24 million transformations, the average response time per lookup is less than 3 ms.

3 Segmentation

In this section, we discuss another primitive called *segmentation (parsing)*. The segmentation primitive takes as input unstructured text and produces as output a structured record obtained by assigning labels to various segments of the text. Figure 3(a) shows a set of sample laptop product names and Figure 3(b) shows the desired output of segmenting the product names. Segmentation is a useful data cleaning (transformation) primitive since structured records enable richer search compared to unstructured text. Further, segmentation primitive can be useful for record matching. Consider record matching directly over the records in Figure 3(a). Here, textual similarity is very misleading: The records with id 1 and 2 are textually very similar but are not matches since they have different processor models. Conversely, records 1 and 3 are textually dissimilar but correspond to the same laptop configuration. The segmented records of Figure 3(b) let us overcome the limitations of textual similarity. We can design sophisticated record matching logic that, for example, checks if two processor models are equal before classifying the corresponding records as matches. More generally, segmentation can be used as a preprocessing step before using primitives such as Fuzzy Lookup in the design of record matching packages. Our current implementation of the segmentation operator involves *parsing* an input string using a regular expression and using the parse to identify segments and assign labels. The regular expression can reference *dictionaries*

which encode domain knowledge. As an illustrative (and highly simplified) example, we can segment address strings (e.g., "`123 Main St Seattle WA 12345`") using the regular expression:

$$\underbrace{[\text{digit}]+}_{Street\ Num}\ \underbrace{[\text{letter}]+\ \text{“St”}}_{Street\ Name}\ \underbrace{[\text{City}]}_{City}\ \underbrace{[\text{State}]}_{State}\ \underbrace{[\text{digit}]+}_{PostalCode}$$

where [digit] and [letter] represent digits and letters, and [City] and [State] refer to dictionaries of city and state names. We note that dictionaries are a form of customization input similar to transformations for Fuzzy Lookup.

4 Design Tools

Recall from Section 1 that designing a data cleaning solution in our platform involves composing various primitives into a *package (or program)* and customizing each primitive in the package for the domain of interest. For complex data cleaning tasks, manually identifying the best way to compose and customize primitives can be difficult, motivating the need for design tools that reduce the manual effort involved.

Customizing Fuzzy Lookup involves specifying a tokenization scheme, token weights, and transformation rules. For most record matching scenarios, whitespace-based tokenization and IDF-based token weights are reasonable defaults. Identifying a useful set of transformations is more challenging. We have recently developed a tool [2] that automatically identifies candidate transformations from examples of matching records. The basic idea is to identify a concise set of transformations that explain textual "differences" between the example records. For example, the two matching addresses `123 Main St` and `123 Main Street` suggest the transformation `St`→`Street`. Similarly, customizing the segmentation primitive involves identifying appropriate dictionaries. We recently developed techniques to automatically identify dictionaries (e.g., list of cities) given a few examples (e.g., `San Francisco`, `New York`) using Web data [15]. The technique uses the observation that many lists on the web contain members of a concept, e.g., names of cities. In the example above, other strings that occur in the same lists as `San Francisco` and `New York` are likely to be cities, and can therefore be identified. Developing a general tool to automatically identify the best way to combine primitives for a given data cleaning task is challenging since the space of possible compositions is quite large. We have taken some initial steps in this direction, and have a tool that identifies a good record matching package that uses only Fuzzy Lookup primitive [4]. Ideally, it is desirable to have a single design tool for both customizing primitives and combining them; developing such a tool would be an interesting avenue for future work.

When integrating data from various sources into a warehouse, it is useful to be able to assess the quality of data being integrated, and we have developed a *Data Profiling* tool for this purpose. The Data Profiling tool efficiently identifies various properties of a dataset such as value distributions, (approximate) keys, functional dependencies, foreign keys, and regular expression patterns in data [10]. These properties can help identify data quality issues and also serve as input for subsequent data cleaning steps. For example, functional dependencies can be exploited for filling missing values and regular expressions can be used for segmentation (see Section 3).

5 Conclusions

In this paper, we provided a brief overview of the data cleaning project at Microsoft Research where we have developed primitives and design tools that allow a programmer to develop data cleaning solutions with minimal programming effort. In particular, we presented two primitives for data cleaning, namely, Fuzzy Lookup and segmentation. While our primitives are generic, they allow rich customizations that can be domain or application specific and can efficiently handle large datasets. Our primitives have been successfully used in several large real-world data cleaning applications.

Acknowledgments

Venkatesh Ganti was a (founding) member of our Data Cleaning project until 2009. He was involved in several research initiatives and the shipping of the initial version of Fuzzy Lookup in SSIS.

References

[1] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, 2008.

[2] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1), 2009.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set similarity joins. In *VLDB*, 2006.

[4] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, 2010.

[5] Bing shopping. `http://www.bing.com/shopping`.

[6] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *CACM*, 54(8), 2011.

[7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD*, June 2003.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 29(2), 2006.

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[10] Z. Chen and V. R. Narasayya. Efficient computation of multiple group by queries. In *SIGMOD*, 2005.

[11] A. Elmagarmid, P. G. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowledge and Data Engg.*, 19(1), 2007.

[12] Fuzzy lookup add-in for excel. `http://www.microsoft.com/bi/en-us/Community/BILabs/Pages/FuzzyLookupAddInforExcel.aspx`.

[13] Person finder: 2011 japan earthquake. `http://japan.person-finder.appspot.com/`.

[14] L. Gravano, P. G. Ipeirotis, et al. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[15] Y. He and D. Xin. Seisa: set expansion by iterative similarity aggregation. In *WWW*, pages 427–436, 2011.

[16] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[17] Data cleaning project. http://research.microsoft.com/en-us/projects/datacleaning/default.aspx.

[18] Trillium Software. www.trilliumsoft.com/trilliumsoft.nsf.