Data Debugger: An Operator-Centric Approach for Data Quality Solutions

Surajit Chaudhuri Venkatesh Ganti Raghav Kaushik
Microsoft Research
{surajitc,vganti,skaushi}@microsoft.com

1 Introduction

Data cleaning is an essential step in populating and maintaining data warehouses. Owing to likely differences in conventions between the external sources and the target data warehouse as well as due to a variety of errors, data from external sources may not conform to the standards and requirements at the data warehouse. Therefore, data has to be transformed and cleaned before it is loaded into the warehouse so that downstream data analysis is reliable and accurate. This is usually accomplished through an Extract-Transform-Load (ETL) process.

Typical data cleaning tasks include record matching [10, 12, 6, 9, 14, 16], deduplication [16], and column segmentation [5, 1] which often go beyond traditional relational operators. This has led to development of utilities that support data transformation and cleaning. Such software falls into two broad categories. The first category consists of verticals such as Trillium [20] that provide data cleaning functionality for specific domains, e.g., addresses. By design, these are not generic and hence cannot be applied to other domains. The other category of software is that of ETL tools such as Microsoft SQL Server Integration Services (SSIS) [22] and IBM Websphere Information Integration [23] that can be characterized as "horizontal" *platforms* that are applicable across a variety of domains. These platforms provide a suite of operators including relational operators such as `select, project` and `equi-join`. A common feature across these frameworks is *extensibility*—applications can plug in their own custom operators. A data transformation and cleaning solution is built by composing these (default and custom) operators to obtain an operator tree or a graph. This extensible operator-centric approach is also adopted in research initiatives such as Ajax [13] and Morpheus [11].

While the second category of software can in principle support arbitrarily complex logic by virtue of being extensible, it has the obvious limitation that most of the data cleaning logic potentially needs to be incorporated as custom code since creating optimized custom code for data cleaning software is nontrivial. It would be desirable to extend its repertoire of "built-in" operators beyond traditional relational operators with a few core data cleaning operators such that with very less extra code, we can obtain a *rich variety* of data cleaning solutions.

In our Data Debugger project, we seek to achieve the above goal. Thus, we aspire to identify key primitive data cleaning operators and then ensure their efficient implementation on horizontal ETL engines such as SSIS. Thus, we adopt the approach of developing a domain-neutral framework of generic data cleaning operators. We believe that decomposing a data cleaning solution into simpler well-defined operators makes it easier to compose data cleaning operators with each other and with other (relational and non-relational) operators. In particular, it will be possible to easily customize and to analyze the overall data cleaning solution.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

While this is the long-term goal of our project, we illustrate our approach with the record matching operation in this paper. Section 2 goes over prior approaches to this problem and we examine our solution in Section 3. We conclude in Section 4 with a brief discussion of next steps.

2 Record Matching: An Overview

The *record matching* problem forms the basis for record linkage (e.g. [10, 12, 6, 9, 14]) and identifying approximately duplicate entities in databases (e.g., [16]). This is a very important problem in data cleaning. We illustrate our approach for developing data cleaning solutions using this problem.

Record matching is the operation of joining "similar" data. For example, consider a sales data warehouse. Owing to likely errors in the data such as typing mistakes, differences in conventions, product names and customer names in sales records may not match exactly with records in the master product catalog and reference customer registration tables, respectively. For example, two records "[Microsoft Corp.,, Redmond, WA, 98052]" and "[Microsoft Corporation, One Microsoft Way, Redmond, WA, 98052]" are not equal even though they identify the same organization.

The record matching problem has been considered in several domains such as web informatics [3, 15] and genome sequencing [17]. At the core of this problem, across all domains, is the concept of a measure that can be used to compare records, called a *similarity function*. The similarity function returns a value between 0 and 1, a higher value indicating that the two records are closer to being the same. A value of 1 indicates exact equality. A similarity join between two relations R and S returns all pairs of records, one each from R and S, such that the similarity function when applied to them is greater than a threshold [12, 16, 10]. By setting this threshold to a "high" value, we find all pairs that we can consider to be matches. Examples of similarity functions include edit distance, cosine similarity, Jaccard similarity and the recently proposed generalized edit distance [6]. Recent work on record matching [2] has also proposed the use of information beyond the text of records in order to determine whether they are duplicates. An example of this use of contextual information is to determine whether two state names (say "Washington" and "WA") are the same by checking similarity of the co-occurring sets of cities present in these states. Note that these notions of similarity apply independent of the textual content in the records. We informally use the term "co-occurrence similarity function" to apply to these notions of similarity.

Each of the above similarity functions has a unique characteristic and it is well-known that no single similarity function is universally applicable across all domains and scenarios [18]. For example, the characteristics of an effective similarity function for matching products based on their part names, where the errors are usually spelling errors, would be different from those matching street addresses because even small differences in the street numbers such as "148th Ave" and "147th Ave" are crucial, which would be different from similarity functions for matching person names based on their sounds. In fact, techniques based on machine learning for combining several similarity functions have been shown to be effective and is an active area of research [18, 21, 4, 19].

A general purpose data cleaning platform has to efficiently support this whole variety of notions of similarity. A naive implementation is to apply the similarity predicate after the cross product between the input relations. Obviously, this approach is prohibitively expensive when the input relations are large.

Several prior techniques provide efficient implementations for specific similarity functions [14, 9]. Naturally this option is tedious and it does not enable us to get to a more general solution. Alternatively, *blocking* heuristics have been proposed as a replacement for cross products. The idea is to first "group" potential candidate duplicates together and then apply similarity joins among small groups of candidates. Candidate groups are identified by grouping all records which match exactly on "approximate key" attributes. These approximate key attributes may either be user-specified [16] or be artificially generated via heuristic signature generation schemes [20]. These techniques are heuristic in nature. They do not necessarily guarantee that all pairs of records which satisfy the similarity predicate will be returned. Thus, the independence between the operator specification and the implementation is lost.

We are left with the challenging option of supporting a foundational primitive which can be used as a building block to implement a broad variety of notions of similarity. Our approach to record matching is driven by the desire to support a wide variety of standard similarity functions like Jaccard similarity with (IDF) weights on tokens, edit similarity. We also consider the generalized edit similarity function [6], which builds upon edit distance and jaccard containment measures with (IDF) weights on tokens, and has been shown to provide better matches in several scenarios. More importantly, we want to support all of these functions using a uniform abstraction so that we may then focus on efficient support for this abstraction. We achieve this using the SSJoin operator described in the next section. Our second goal that distinguishes us from the above approaches based on blocking is to separate, as in the case of relational operators, the operator specification from its implementation. Our implementation of similarity join returns all results required to be returned by its specification.

3 Record Matching Using the SSJoin Operator

As noted in Section 2, our goal is to efficiently support a foundational primitive which can be used as a building block to implement a broad variety of notions of similarity. The *SSJoin* primitive proposed in [7] and discussed below enables such a general implementation. The idea is to model strings as sets and perform a set-similarity join over these sets (hence the name S(et)S(imilarity)Join). In this way, most of the the effort in a similarity join can be done by the SSJoin operator. The benefit of an efficient implementation of the SSJoin operator simultaneously translates to a variety of similarity functions.

OrgName	3-gram	Norm
Microsoft Corp	mic	12
Microsoft Corp	icr	12
Microsoft Corp	cro	12
...	...	...
Microsoft Corp	cor	12
Microsoft Corp	orp	12

R

OrgName	3-gram	Norm
Mcrosoft Corp	mcr	11
Mcrosoft Corp	cro	11
Mcrosoft Corp	ros	11
...	...	...
Mcrosoft Corp	cor	11
Mcrosoft Corp	orp	11

S

Figure 1: Example sets from strings

We now describe the SSJoin operator. The operator works with sets and intuitively finds all pairs of sets that have a high overlap. We refer to the size of the intersection between two sets s_1, s_2 to be their overlap similarity, denoted $Overlap(s_1, s_2)$. While our discussion below focuses on unweighted sets, the discussion extends to weighted sets as in [7].

We assume that sets are represented in First Normal Form by storing every element in a separate record. Consider relations $R(A, B)$ and $S(A, B)$ where A and B are subsets of columns. Each distinct value $a_r \in R.A$ defines a group, which is the subset of tuples in R where $R.A = a_r$. Call this set of tuples $Set(a_r)$. Similarly, each distinct value $a_s \in S.A$ defines a set $Set(a_s)$. The simplest form of the SSJoin operator joins a pair of distinct values $\langle a_r, a_s \rangle$, $a_r \in R.A$ and $a_s \in S.A$, if the projections on column B of the sets $Set(a_r)$ and $Set(a_s)$ have a high overlap. The formal predicate is $Overlap(\pi_B(Set(a_r), \pi_B(Set(a_s))) \geq \alpha$ for some threshold α. We denote $Overlap(\pi_B(Set(a_r), \pi_B(Set(a_s)))$ as $Overlap_B(a_r, a_s)$. Hence, the formal predicate is $Overlap_B(a_r, a_s) \geq \alpha$. We illustrate this through an example.

Example 1: *Let relation $R(OrgName, 3\text{-}gram)$ and $S(OrgName, 3\text{-}gram)$ shown in Figure 1 associate the strings "Microsoft Corp" and "Mcrosoft Corp" with their 3-grams. Denoting* OrgName *by A and* 3-gram *by B, the SSJoin operator with the predicate $Overlap_B(a_r, a_s) \geq 10$ returns the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since the overlap between the corresponding sets of 3-grams is 10.*

In general, we may wish to express conditions such as: the overlap similarity between the two sets must be 80% of the set size. Thus, in the above example, we may wish to assert that the overlap similarity must be higher than 80% of the number of 3-grams in the string "Microsoft Corp". We may also wish to be able to assert that the overlap similarity be higher than say 80% of the sizes of *both* sets. We now formally define the SSJoin operator as follows, which addresses these requirements.

Definition 1: Consider relations $R(A, B)$ and $S(A, B)$. Let $pred$ be the predicate $\bigwedge_i Overlap_B(a_r, a_s) \geq \ e_i$, where each e_i is an expression involving only constants and columns from either $R.A$ or $S.A$ (but not both). We write R SSJoin_A^{pred} S to denote the following result: $\{\langle a_r, a_s\rangle \in R.A \times S.A | pred(a_r, a_s)$ is true $\}$. We also denote $pred$ as $\{Overlap_B(a_r, a_s) \geq e_i\}$.

We illustrate this through the following examples based on Figure 1, where the third column *Norm* denotes the length of the string. In general, the *norm* could denote either the length of the string, or the cardinality of the set, or the sum of the weights of all elements in the set. Several similarity functions use the norm to normalize the similarity.

Example 2: *As shown in Figure 1, let relations $R(OrgName, 3\text{-}gram, Norm)$ and $S(OrgName, 3, Norm)$ associate the organization names with (1) all 3-grams in each organization name, and (2) the number of 3-grams for each name. The predicate in the SSJoin operator can be used to capture different notions of similarity by varying the predicate specification.*

- *Absolute overlap: $Overlap_B(a_r, a_s) \geq 10$ joins the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since the overlap between the corresponding sets of 3-grams is 10.*
- *1-sided normalized overlap: $Overlap_B(\langle a, norm\rangle_r, \langle a, norm\rangle_s) \geq 0.8{\cdot}R.norm$ joins the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since the overlap between the corresponding sets of 3-grams is 10, which is more than 80% of 12.*
- *2-sided normalized overlap: $Overlap_B(\langle a, norm\rangle_r, \langle a, norm\rangle_s) \geq \{0.8 \cdot R.norm, 0.8 \cdot S.norm\}$ also returns the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since 10 is more than 80% of 12 and 80% of 11.*

3.1 Exploiting SSJoin for Similarity Joins

First, note that the above definition of SSJoin can be directly used to capture notions of similarity based on co-occurrence [2] where it has been shown to be very effective for identifying approximate duplicates. We illustrate with an example below.

Example 3: *Suppose we have two tables, say from different sources that are being integrated, of author names joined with the titles of the papers, say with the schema* <PTITLE, ANAME>. *Since we want a unified view of all authors, we are interested in identifying author names that are likely to represent the same author. Now, if the naming conventions in the two sources are entirely different, it is quite likely that the textual similarity between the author names is only a partial indicator of their similarity. We are forced to rely on alternative sources of information for identifying duplicate author entities. In this instance, we can use the set of paper titles associated with each author to identify authors. The idea is that if two authors are the same, then the set of paper titles co-occurring with them must have a large overlap.*

The notion of set overlap can also be used to capture various string similarity functions—edit distance, jaccard similarity and generalized edit similarity as shown in [7]. The way the SSJoin operator is used for string similarities is outlined in Figure 2. Let Rbase(A) and Sbase(A) be relations where A is a string-valued attribute. The goal is to find pairs $\langle$Rbase$.A$, Sbase$.A\rangle$ where the string similarity, according to a specific similarity function, is above a threshold α. We first convert the strings Rbase(A) and Sbase(A) to sets, construct normalized representations $R(A, B, norm(A))$ and $S(A, B, norm(A))$, and then perform an SSJoin between the normalized representations. The SSJoin predicate, especially the overlap threshold α', is chosen so that all pairs whose

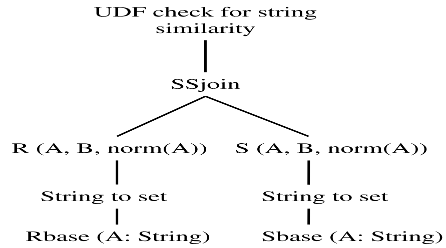

Figure 2: String Similarity Join using SSJoin

string similarity is greater than α are *guaranteed* to be in the result of the SSJoin. Hence, the SSJoin operator provides a way to efficiently produce a small superset of the correct answer. We then compare the pairs of strings using the actual similarity function, declared as a UDF within a database system, to ensure that we only return pairs of strings whose similarity is above α.

Note that a direct implementation of the UDF within a database system without SSJoin is most likely to lead to a cross-product where the UDF is evaluated for all pairs of tuples. Hence, this is an impractical approach. On the other hand, an implementation using SSJoin exploits the support within database systems for equi-joins to result in a significant reduction in the total number of string comparisons. This results in orders of magnitude improvement in performance [7]. Ajax [13] proposed a matching primitive based on a user-specified similarity function. First, this focuses purely on record level similarity and hence cannot capture co-occurrence similarity functions. Further, they do not identify any common features across all similarity functions they support. Hence, they run the risk that their implementation would reduce to a cross-product followed by a UDF check and hence would also be impractical.

We now illustrate how similarity join with respect to edit distance can be performed using SSJoin. The edit distance between strings is the least number of edit operations (insertion and deletion of characters, and substitution of a character with another) required to transform one string to the other. For example, the edit distance between strings ‘microsoft’ and ‘mcrosoft’ is 1, the number of edits (deleting ‘i’) required to match the second string with the first.

We illustrate the connection between edit distance and overlap through the following example.

Example 4: *Consider the strings “Microsoft Corp” and “Mcrosoft Corp”. The edit distance between the two is 1 (deleting ’i’). The overlap similarity between their 3-grams is 10, more than 80% of the number of 3-grams in either string.*

The intuition is all q-grams that are “far away” from the place where the edits take place must be identical. Hence, if the edit distance is small, then the overlap on q-grams must be high. This intuitive relationship between edit distance and the set of q-grams can be formalized as follows [14].

Property 2: [14] Consider strings σ_1 and σ_2, of lengths $|\sigma_1|$ and $|\sigma_2|$, respectively. Let $QGSet_q(\sigma)$ denote the set of all contiguous q-grams of the string σ. If σ_1 and σ_2 are within an edit distance of ϵ, then $Overlap(QGSet_q(\sigma_1), QGSet_q(\sigma_2)) \geq \max(|\sigma_1|, |\sigma_2|) - q + 1 - \epsilon \cdot q$

Thus, in the above example, the edit distance is 1, and Property 2 asserts that at least 9 of all the 3-grams from the two strings have to be common. Based on this property, we can implement similarity join based on edit distance using SSJoin, as detailed in [7].

3.2 Top-K Similarity Join

Consider the example where we want to match a product name in a sales transaction reported by a reseller against the standardized set of products in a product reference table. In this context, we know that the incoming products have to match with those in the reference table. If there is no record in the reference table which does not match exactly with an input product name, we want to match it with the "best matching" record in the reference table. This operation has been shown to be very effective in dealing with input errors [6, 20]. The intuitive notion of match quality can again be quantified using a similarity function and thus this operation is like the similarity join between the input relation and a reference table. However, unlike the similarity join where a user specifies the threshold, we want the most similar record or in general the K most similar records to each record in the input relation.

Observe that we stipulate one of the two relations involved as a *reference table*, and for each record in the other relation we want the K most similar records from the reference table. Thus, this operation is inherently asymmetric unlike the similarity join, which can be symmetric if the similarity function is. As with the similarity join operator, we ideally want to be able to support the top-K similarity join operator over many similarity functions discussed earlier. Currently, we support an index-based implementation (with probabilistic guarantees) of this operation for the generalized edit similarity in SQL Server Integration Services 2005. We call this specific implementation the *fuzzy lookup* transform. Other details of this implementation are discussed in [6].

4 Conclusions

In this paper, we discussed the Data Debugger framework for data cleaning. The main goal is to identify generic and robust abstractions for data cleaning operators, and to support efficient implementations of these abstractions within the platform. We are now able to compose these operators with other relational and non-relational operators to derive operator trees.

We illustrated the Data Debugger framework for the specific problem of record matching, which is only one important operation in data cleaning. Deduplication (also called entity resolution) [16, 8], which partitions a relation into groups of records based on their pairwise similarities, is also an important data cleaning operation. Another operation is one which takes an attribute value and segments it into constituent attribute values [5, 1]. Commercial address cleansing tools (e.g., Trillium) rely heavily on this operation, which they call "parsing". We want to extend the Data Debugger framework to these other important data cleaning operations.

References

[1] E. Agichtein and V. Ganti. Mining reference tables for automatic text segmentation. In *Proceedings of ACM SIGKDD*, 2004.

[2] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of VLDB*, 2002.

[3] K. Bharat and A. Z. Broder. Mirror, mirror on the web. *Computer Networks*, 1999.

[4] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of ACM SIGKDD*, 2003.

[5] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *Proceedings of ACM SIGMOD*, 2001.

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

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

[8] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proceedings of ICDE*, 2005.

[9] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD*, 1998.

[10] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on information systems*, 2000.

[11] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, and M. Stonebraker. Data integration through transform reuse in the morpheus project. In *Proceedings of ACM SIGMOD*, 2006.

[12] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 1969.

[13] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of VLDB*, 2001.

[14] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of VLDB*, 2001.

[15] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the 3rd Intl. Workshop on Web and Databases*, 2000.

[16] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of ACM SIGMOD*, 1995.

[17] M. Narayanan and R. M. Karp. Gapped local similarity search with provable guarantees. In *Proc. of the 4th Intl. Workshop on Algorithms in Bioinformatics*, 2004.

[18] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of ACM SIGKDD*, 2002.

[19] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of ACM SIGMOD*, 2004.

[20] Trillium Software. www.trilliumsoft.com.

[21] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems*, 2001.

[22] Microsoft SQL Server 2005 integration services. http://msdn.microsoft.com/sql.

[23] IBM Websphere Information Integration. http://ibm.ascential.com.