

SPIDER: Flexible Matching in Databases

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Amit Marathe
AT&T Labs–Research
marathe@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

We present a prototype system, SPIDER, developed at AT&T Labs–Research, which supports flexible string attribute value matching in large databases. We discuss the design principles on which SPIDER is based, describe the basic techniques encompassed by the tool and provide a description of the demo.

1. INTRODUCTION

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases (see, e.g., [4]). Poor data quality is the result of a variety of reasons, including data entry errors (e.g., typing mistakes), poor integrity constraints and lack of standards for recording database fields (e.g., addresses). Data quality issues could instigate a variety of business problems, such as inefficient customer relationship management (e.g., inability to retrieve a customer record during a service call), billing errors and distribution delays.

Every large organization maintaining a multitude of databases is likely to face data quality problems. A very typical problem encountered in such settings is that customer information (names of individuals, names of corporations, addresses) is represented differently across related databases. Similar problems exist with product names, product descriptions, etc. Such information consists primarily of strings. As a result, there is a pressing need for technologies that enable flexible (fuzzy) matching of string information in a database. At AT&T Labs–Research, we have been developing a prototype system called SPIDER to address various aspects of data quality. In this demonstration of our tools we will be exhibiting the support provided for flexible attribute value matching in large databases. We will briefly outline the design principles on which SPIDER is based, describe the basic techniques encompassed by the tool and provide a description of the demo.

2. DESIGN PRINCIPLES

Trends in data quality research and development have evolved in two main directions. The first one is based on the use of proprietary technology that requires exclusive access to the data. A variety of companies (including, e.g., [1, 2]) provide tools that operate on

data to identify quality problems using a multitude of technologies. Such tools commonly extract data out of relational databases and apply proprietary algorithms.

The second direction makes use of declarative specifications of data quality tasks. Data quality algorithms and techniques operate on data directly in the relational database, and are often expressed as SQL statements (see, e.g., [3, 5, 6, 7]). Such an approach has the advantage of not requiring extraction of the data outside the database which can be a serious concern in large enterprises. Moreover it can readily leverage query optimization and execution provided by the database. Our SPIDER tool falls in this category. At a very high level, SPIDER interfaces with any relational database, issues declarative statements to preprocess any information it requires and is capable of expressing operations necessary to enable flexible matching in a declarative way.

3. DESCRIPTION

SPIDER, due to its design principles, is very versatile in terms of correlating and manipulating databases. Its main features are described in the following subsections.

3.1 Core Technology

In order to enable flexible matching on strings (customer names, product names, addresses, etc.), there is a need for principled techniques to quantify the closeness of corresponding relational attribute values. A variety of predicates exist for such a purpose, including *edit distance* and *tf-idf cosine similarity*, for which efficient declarative specifications and realizations on relational databases have been proposed [3, 5, 6, 7]. In applying these predicates to our application data, we have found that tf-idf cosine similarity works much better than edit distance.

Let *Base* denote the table with a string-valued attribute *sva* against which flexible matching needs to be performed, and let *Search* denote the table containing the search strings (this may consist of just a single record with a single attribute value, or may be more complex). Flexible string matching is done in two stages, using the techniques described in [7].

- At *pre-processing time*, the *Base* table is indexed, and *q-gram* tokens are extracted from each string in *Base.sva*. A variety of auxiliary tables get created to compute the *idf* values of each token, and ultimately to associate each database string with a (normalized) weight vector (incorporating both *tf* and *idf*) corresponding to the tokens extracted from it.
- At *query time*, a similar process is first done with respect to the *Search* table. Then, an SQL query that operates on the auxiliary tables created from *Base* and *Search* is executed, which identifies the matching records, along with their

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 \$5.00.

Approximate Search Page (2 column)

Search String 1:

Search String 2:

Search Database:

Similarity Threshold:

Matching LBDW location records

	sim	str1	str2
Explain sim	0.568322017929254	WORLDCO	110 wall st new york ny
Explain sim	0.568322017929254	WORLDCO	110 wall st new york ny
Explain sim	0.481778518568938	WORLDCOM	111 8th ave new york ny
Explain sim	0.462023103220271	WORLDCOM	60 hudson st new york ny
Explain sim	0.461736895695425	WORLDCOMM	140 west st new york ny
Explain sim	0.461223519217079	WORLDCOM POP	750 e main st chattanooga tn

Figure 1: Double column flexible matching

similarity score. Essentially, this query computes the cosine similarity of the weight vectors of the search string with the weight vectors of the database strings in `Base.sva`, taking the weights of the common tokens into account.

3.2 Flexible String Matching

After a table has been indexed, SPIDER can perform flexible string matching on any indexed attribute. It is frequently the case that tables have many string valued attributes and what is desired is a multi-column search, which, given a sequence of strings (e.g., company name and address), returns all database tuples “close” to the search strings. A multi-column search can be thought of as a series of single-column searches along with a combining function.

SPIDER asks the user to input one or more search strings and a similarity threshold. It returns all tuples whose tf-idf cosine similarity score is greater than the given threshold. For multi-column matching, the user can also specify a combining function which is used to merge the results of the single-column sub-searches. A variety of techniques are available for this purpose, including static weighting and various variants of dynamic weighting. In static weighting, each search attribute is assigned a fixed weight, while in dynamic weighting the relative weights of the search attributes are adjusted on a per-tuple or per-search basis [7].

Figure 1 presents a sample results page of SPIDER for the case of a query using the cosine similarity function against two attributes of a table. The search strings (name and address), a similarity threshold and the attribute specifications are provided, and the results are displayed. Note that, using our tf-idf cosine similarity, we can also take care of small edit errors (“worldcom” in the search string vs “worldco” in the database string).

3.3 Result Explanation

When trying to understand any fuzzy matching algorithm, there is a need for an *explain* feature that can be used to validate the results returned. This is important not just for debugging but also from a user experience perspective.

When local metrics like edit-distance are used, an explain feature is of limited utility because the validity of a result is just a matter of checking whether a *locally testable predicate* is satisfied by a pair of strings. With global metrics like tf-idf cosine similarity, the

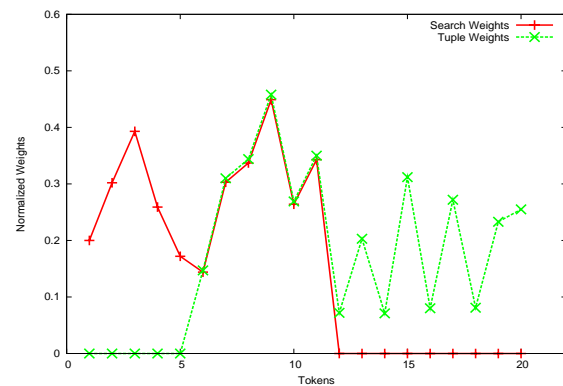


Figure 2: Result explanation

score of a tuple depends not only on the search string but also on all the other tuples in the indexed table. It is for global metrics like these that an easily available explanation of the results can go a long way in helping users develop a feel for the behavior of the metric and thereby assist them in evaluating the appropriateness of the metric to the problem at hand.

In SPIDER, each result is presented along with a hyperlink. Clicking on the hyperlink takes the user to a new page which displays the normalized weights for all tokens in the search and database strings along with their raw idf weights. This allows the user to see at a glance the tokens common to the two strings, and how they contribute to the final score. The user can also quickly determine whether a low score is due to a lack of common tokens or due to the absence of some high weight token from the search string.

Figure 2 presents a sample explanation page of the similarity between the search string “Microsoft” and the database string “Microsoft Inc”. It can be seen that there are quite a few high-weight tokens that are not common to the two strings. That accounts for the relatively low similarity score of 0.62 for the database string.

3.4 Query Refinement

At times, it may be necessary to increase/decrease the importance of particular tokens in the search string. For example, when searching by customer names we may want to emphasize the last name (as opposed to the first name). In these instances, we want the ability to change the weights of some search tokens from their default tf-idf weight. Such a feature will enable users to run a more focused search while still retaining the basic properties of the tf-idf cosine similarity metric.

SPIDER allows the user to indicate which tokens of the search string should be considered for query refinement. The weights of all these tokens are modified (based on a user-specified factor) and the search weight vector is re-normalized. The before and after results of this change can be seen side-by-side in the result pane.

Figure 3 shows the results for a query on the string “429 ridge rd dayton nj”. The tokens derived from the first word of the query have their weights increased by a factor of 10. It can be seen that this adjustment has the effect of dropping some old matches while picking up a few new matches. One issue worth noting is that, after the query refinement, the similarity score of the exact match drops. The reason is that the weight vectors for the query and the data strings are normalized using the l_2 -norm (length in the Euclidean space) of the corresponding un-normalized weight vectors. So the similarity score is 1 for some database string if and only if its weight vector is identical to the search weight vector. Since query refinement changes the search weight vector but not the database

Refine Query

Search String:

Refine: by factor

Search Database:

Similarity Threshold:

Matching LBDW records

address	sim_before	sim_after	tid1	t
429 ridge rd dayton nj	1	0.63	44	5
305 ridge rd dayton nj	0.69		44	1
431 ridge rd dayton nj	0.68		44	5
437 ridge rd dayton nj	0.66		44	1
473 ridge rd dayton nj	0.65		44	1
429 2nd ave new york ny		0.53	44	4
429 w 15th st new york ny		0.53	44	4

Figure 3: Query refinement

string weight vectors, query refinement downgrades some matches while raising the scores of others. This can also be understood using the result explanation feature, described earlier.

3.5 Data Refinement

While the similarity metric used in SPIDER takes into account the frequencies of all tokens (rare tokens get a high weight and common tokens get a low weight), it does not consider the joint distribution of sets of tokens. For example, in a table of customer names, “john” and “kumar” might be common first and last names but “john kumar” may be an uncommon name. In such a scenario, we would like the ability to adjust the weight vector of names close to “john kumar” to incorporate the rarity of that combination. The tuples which contain “john” (or “kumar”) but are not close to “john kumar” would not be affected. This makes for a fine-grained mechanism to dynamically adjust the tuple weights so as to emphasize particular rare combinations of common tokens.

SPIDER allows the user to dynamically enter such infrequent co-occurrences of common substrings. For each such string it searches the database for appropriate tuples. The user can select any combination of these tuples to participate in the weight adjustment. The weights of the “frequent” tokens in each selected tuple, and only in these tuples, are increased by a user specified parameter. Finally, all the selected tuples have their weight vectors re-normalized. Subsequent searches run against this modified database.

3.6 Synonym Table

It is often the case that the same entity is represented in multiple ways inside the database. For example, “1 ATT Way Bedminster NJ” and “900 Route 202/206 Bedminster NJ” are both valid addresses for ATT’s headquarters. We would like our fuzzy matching algorithms to be aware of such semantics, so that an entity can be searched for using any of its representations. Furthermore, we also want this semantic matching to be robust in dealing with errors and multiple conventions (see, e.g., [7]).

Semantic equivalences are represented in SPIDER through a synonym table. The user can add pairs of strings to this table. Every search string is first checked against this table to pick up any equivalent strings, which are then appended to the search string. This

augmented string is then used for a regular tf-idf cosine similarity search on the database,

4. USE CASES

This section outlines some possible scenarios for how users might interact with SPIDER. We will name the users Alice and Bob.

Consider a database table where the address components (street, city, state) are in separate columns. For simplicity of matching, the address components are concatenated into a single string. Alice starts by searching for the address “180 Avenue of the Americas New York NY”. After looking at the results and drilling down to the explanation of a few scores, she observes that the city-state substring “New York NY” is not contributing much to the score. This is probably because the database has many addresses from New York City. Alice is not interested in out-of-state addresses, so she decides to use query refinement to increase the importance of the city-state substring in the search. This has the effect of discounting addresses which previously matched on the street part but did not match on the city-state part.

The results after query refinement are more to her liking. However, Alice notices that an expected address “180 6th Avenue New York NY” is not among the search results. To correct this false negative, she enters “6th Avenue New York NY” and “Avenue of the Americas New York NY” as a pair into the synonym table. After this addition, addresses on “6th Avenue” in New York City can be searched using the street name synonym “Avenue of the Americas” (and vice versa).

During another search on “451 New Jersey Ave Madison Wisconsin”, Bob comes across the high-scoring address “451 Wisconsin Ave Madison New Jersey”. Obviously, the search algorithm is not distinguishing between different parts of the address. To evaluate an alternative approach, Bob decides to search the address components of street, city and state separately, based on per-column SPIDER indexes. He then tries out multi-column matching with static and dynamic weights. The results are tighter than before but the downside is that the search string is no longer free-form: it must be entered as three distinct fields.

5. CONCLUSION

SPIDER is a prototype system that supports flexible string attribute value matching. Given the declarative (SQL based) nature of its flexible matching, SPIDER can easily support many variations of the basic single-column matching, including multiple column matching, and the use of synonym tables. Given the global nature of the tf-idf cosine similarity function used effectively in SPIDER, our tools can effectively support query and data refinement capabilities. The result explanation utility provides considerable help to the user in understanding the flexible matching methodology.

6. REFERENCES

- [1] Dataflux Inc. www.dataflux.com.
- [2] Netrics Inc. www.Netrics.com.
- [3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. *Proceedings of SIGMOD*, 2003.
- [4] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.
- [5] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. *Proceedings of VLDB*, 2001.
- [6] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. *Proceedings of WWW*, 2003.
- [7] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. *Proceedings of VLDB*, 2004.