Text Joins for Data Cleansing and Integration in an RDBMS

Luis Gravano* Panagiotis G. Ipeirotis* Nick Koudas Divesh Srivastava
Columbia University AT&T Labs–Research
{gravano,pirot}@cs.columbia.edu {koudas,divesh}@research.att.com

Abstract

An organization's data records are often noisy because of transcription errors, incomplete information, lack of standard formats for textual data or combinations thereof. A fundamental task in a data cleaning system is matching textual attributes that refer to the same entity (e.g., organization name or address). This matching can be effectively performed via the cosine similarity metric from the information retrieval field. For robustness and scalability, these "text joins" *are best done inside an RDBMS, which is where the data is likely to reside. Unfortunately, computing an exact answer to a text join can be expensive. In this paper, we propose an approximate, sampling-based text join execution strategy that can be robustly executed in a standard, unmodified RDBMS.*

1. Introduction

A fundamental task in data cleaning is the detection of records in a database that refer to the same entity but have different representations across relations or across databases. Many approaches to data cleaning use a text matching step, where similar textual entries are matched together as potential duplicates. Although text matching is an important *component* of data cleaning systems [1, 9], little emphasis is put on the efficiency of this operation. For scalability and robustness reasons, it is important to perform this matching within an RDBMS, which is the place where the data is likely to reside. We propose a technique for implementing the text matching step completely *within an unmodified RDBMS*, using plain SQL statements.

We use the cosine similarity metric [10] of textual similarity to identify potential text matches across relations. Consider, without loss of generality, two relations R_1 and R_2 with one textual attribute each. Each textual attribute can be decomposed into a collection of atomic "entities" that we generally refer to as *tokens*, and which can be defined in a variety of ways[1]. Our discussion treats the term token as generic, as the choice of token is orthogonal to the design of our algorithms.

*Luis Gravano and Panagiotis G. Ipeirotis were funded in part by NSF under Grants No. IIS-97-33880 and IIS-98-17434.

[1]The tokens might be the *words* that appear in the textual attribute. Alternatively, we could divide a textual attribute into q-grams, which are substrings of q consecutive characters. For example, "$A," "AT," "T&," "&T," "T ," " L," "La," "ab," "bs," "s#," are the 2-grams for "AT&T Labs" after we add dummy characters "$" and "#" at the beginning and end of the text [5].

According to the vector-space retrieval model, we conceptually map each tuple $t \in R_i$ to a vector v_t. The value of the j-th component $v_t(j)$ of v_t is a real number that corresponds to the weight of the token j. We exploit an instance of the well-established *tf.idf* weighting scheme [10] to assign weights to tokens. Under this scheme, the weight for a token w in a tuple is high if w appears a large number of times in the tuple (tf factor) and w is a sufficiently "rare" token in the relation (idf factor). Using this scheme, say for a relation with company names, relatively infrequent tokens such as "AT&T" or "IBM" will have higher weights than more frequent tokens such as "Inc." A variant of this general weighting scheme has been successfully used for our task by Cohen's WHIRL system [3]. Our technique can be viewed as bringing WHIRL's functionality inside an RDBMS. To simplify the computation of vector similarities, we normalize vector v_t to unit length.

Definition 1 (Cosine Similarity) *Given tuples $t_1 \in R_1$ and $t_2 \in R_2$, let v_{t_1} and v_{t_2} be their corresponding normalized weight vectors and let D be the set of all tokens in R_1 and R_2. The* cosine similarity *(or just* similarity*, for brevity) of v_{t_1} and v_{t_2} is defined as $sim(v_{t_1}, v_{t_2}) = \sum_{j=1}^{|D|} v_{t_1}(j) v_{t_2}(j)$.*

This similarity metric has values between 0 and 1. Intuitively, two vectors are similar if they share many important tokens. For example, "IBM" will be highly similar to "IBM Corp," since they differ only on the token "Corp," which is likely to appear in many different tuples and hence have low weight. On the other hand, "IBM Research" and "AT&T Research" will have lower similarity as they share only one relatively common token (i.e., "Research").

We use this similarity metric to define a *text join* between relations on textual attributes:

Definition 2 (Text Join) *Given two relations R_1 and R_2, together with a user-specified similarity threshold $0 \le \phi \le 1$, the* text join *$R_1 \tilde{\bowtie}_\phi R_2$ returns all pairs of tuples (t_1, t_2) such that $t_1 \in R_1$ and $t_2 \in R_2$, and $sim(v_{t_1}, v_{t_2}) \ge \phi$.*

2. Sample-based Text Joins in SQL

The text join of two relations can be computed in a number of ways. Cohen's WHIRL system [3] does so using an A^*-based stand-alone procedure. In contrast, in this section we focus on computing text joins inside an RDBMS for robustness

```
SELECT    r1w.tid AS tid1, r2w.tid AS tid2
FROM      R1Weights r1w, R2Weights r2w
WHERE     r1w.token = r2w.token
GROUP BY  r1w.tid, r2w.tid
HAVING    SUM(r1w.weight*r2w.weight) ≥ φ
```

Figure 1. Baseline approach for computing the exact value of $R_1 \tilde{\bowtie}_\phi R_2$.

and scalability, relying only on standard, unmodified SQL. Section 2.1 defines the auxiliary relations that we use. Then, Section 2.2 discusses an efficient, sampling based implementation of text joins in SQL.

2.1. Creating Weight Vectors for Tuples

To compute the text join $R_1 \tilde{\bowtie}_\phi R_2$, we need the weight vector associated with the tokens of each tuple in R_1 and R_2, for a specific choice of tokens (e.g., q-grams or words). Initially, we create the relations $RiTokens(tid, token)$, which contain an entry for each $token$ present in the R_i tuple with id tid. This relation can be implemented in SQL (the implementation varies with the choice of tokens). Using the *RiTokens* relations we can again create in SQL the relations $RiWeights(tid, token, weight)$, where a tuple $\langle tid, token, w\rangle$ indicates that $token$ has normalized weight w in the R_i tuple identified by tid. Finally, we create the relations $RiSum(token, total)$ to store for each $token$ the total added weight $total$ in relation R_i, as indicated in relation $RiWeights$. The SQL statements to create these relations are available at `http://www.cs.columbia.edu/~pirot/DataCleaning/`.

2.2. Implementing Text Joins in SQL

A *baseline approach*, adapted from [6], to compute $R_1 \tilde{\bowtie}_\phi R_2$ is shown in Figure 1. This SQL statement computes the similarity of each pair of tuples and filters out any pair with similarity less than the threshold ϕ. This approach produces an exact answer to $R_1 \tilde{\bowtie}_\phi R_2$ when $\phi > 0$. The result of $R_1 \tilde{\bowtie}_\phi R_2$ only contains pairs of tuples from R_1 and R_2 with similarity ϕ or higher. Usually, we are interested in high values for threshold ϕ, which should typically result in only a few tuples from R_2 matching each tuple from R_1. The baseline approach in Figure 1, however, calculates the similarity of all pairs of tuples from R_1 and R_2 that share at least one token. As a result, this baseline approach is inefficient: most of the candidate tuple pairs that it considers do not make it to the final result of the text join.

We present a sampling-based technique to execute text joins efficiently, drastically reducing the number of candidate tuple pairs that are considered during query processing. The intuition behind the sampling-based approach, which can be viewed as a specialization of the technique presented in [2], is the following: The cosine similarity between two tuples t_1 and t_2 is equal to $\sum_{j=1}^{|D|} v_{t_1}(j) v_{t_2}(j)$. To compute tuple pairs with high similarity, we consider *only* partial products $v_{t_1}(j) v_{t_2}(j)$ with *high* values. Since a product $v_{t_1}(j) v_{t_2}(j)$ cannot be high when either $v_{t_1}(j)$ or $v_{t_2}(j)$ is too small, we can effectively ignore tokens that have low weight $v_{t_1}(j)$ or $v_{t_2}(j)$ and still get a good approximation of the correct similarities. Hence, instead of using the relation $RiWeights$, we can use a smaller relation $RiSample$ that contains a subset of the tokens present in $RiWeights$.

```
SELECT rw.tid, rw.token, rw.weight/rs.total AS P
FROM   RiWeights rw, RiSum rs
WHERE  rw.token = rs.token
```

Figure 2. Creating an auxiliary relation that we sample to create $RiSample(tid, token, c)$.

```
INSERT INTO RiSample(tid,token,c)
SELECT  rw.tid, rw.token,
        ROUND(S * rw.weight/rs.total, 0) AS c
FROM    RiWeights rw, RiSum rs
WHERE   rw.token = rs.token AND
        ROUND(S * rw.weight/rs.total, 0) > 0
```

Figure 3. A deterministic version of the sampling step, which results in a compact representation of *RiSample*.

To create $RiSample$, we use *weighted* sampling and *not* uniform sampling, and we sample each token j from a vector v_{t_q} with probability $\frac{v_{t_q}(j)}{Sum(j)}$, where $Sum(j) = \sum_{k=1}^{|R_i|} v_{t_k}(j)$. We perform S trials for each $RiWeights$ row, yielding approximately S samples for each token j (S is a sampling parameter; larger values result in higher accuracy at the expense of query processing time). We can implement this sampling step in SQL. Conceptually, we join the relations $RiWeights$ and $RiSum$ on the *token* attribute as in Figure 2. The P attribute in the result is the probability with which we should pick a particular tuple. For each tuple in the output of the query of Figure 2 we need to perform S trials, picking each time the tuple with probability P. Then, we insert into $RiSample$ tuples of the form $\langle tid, t, c\rangle$ where c is the number of successful trials. The S trials can be implemented in various ways. We can open a cursor on the result of the query in Figure 2, read one tuple at a time, perform S trials on each tuple, and then write back the result. Alternatively, a pure-SQL "simulation" of the sampling step deterministically defines that each tuple will result in *Round*($S \cdot \frac{RiWeights.weight}{RiSum.total}$) "successes" after S trials, on average. This deterministic version of the query is shown in Figure 3. We have implemented and run experiments using the deterministic version, and obtained virtually the same performance as with the cursor-based implementation of sampling over the Figure 2 query.

After creating the weighted sample of a relation R_2, $R2Sample$, we join it with the other relation R_1 to approximate $R_1 \tilde{\bowtie}_\phi R_2$. The sampling step used only the token weights from R_2 for the sampling, ignoring the weights of the tokens in the other relation, R_1. The cosine similarity, however, uses the products of the weights from *both* relations. During the join step we use the token weights in the non-sampled

```
SELECT   r1w.tid AS tid1, r2s.tid AS tid2
FROM     R1Weights r1w, R2Sample r2s,
         R2sum r2sum
WHERE    r1w.token = r2s.token AND
         r1w.token = r2sum.token
GROUP BY r1w.tid, r2s.tid
HAVING   SUM(r1w.weight*r2sum.total*r2s.c) ≥ S * φ
```

Figure 4. Implementing the sampling-based text join in SQL, by sampling R_2 and weighting the sample using R_1.

relation to get estimates of the cosine similarity, as follows. We focus on each $t_q \in R_1$ and each token j with non-zero weight in v_{t_q}. For each $R2Sample$ tuple $\langle i, j, c\rangle$, we compute the value $v_{t_q}(j) \cdot Sum(j) \cdot c$, which is an approximation of $v_{t_q}(j) \cdot v_{t_i}(j) \cdot S$. The sum of these partial products across all tokens is then an estimate of the similarity of t_q and t_i, multiplied by S. Hence, we can output as the answer those tuple pairs whose associated sum is greater than $S \cdot \phi$, where ϕ is the user-specified threshold.

We implement the join step as an SQL statement (Figure 4). We weight each tuple from the sample according to $R1Weights.weight \cdot R2Sum.total \cdot R2Sample.c$, which corresponds to $v_{t_q}(j) \cdot Sum(j) \cdot c$. Then, we sum the partial products for each tuple pair (see GROUP BY clause). For each group, the result of the SUM is the estimated similarity of the tuple pair, multiplied by S. Finally, we apply the filter as a simple comparison in the HAVING clause: we check whether the similarity of a tuple pair exceeds the threshold. The final output of this SQL operation is a set of tuple id pairs with estimated similarity exceeding threshold ϕ.

3. Related Work

The problem of approximate string matching has attracted interest in the algorithms and combinatorial pattern matching communities [8] and results from this area have been used for data integration and cleansing applications [4, 5]. The string *edit distance* [7] (with its numerous variants) has been frequently used for approximate string matching. Gravano et al. [5] presented a method to integrate approximate string matching via edit distance into a database and realize it as SQL statements.

The information retrieval field has produced approaches to speed up query execution that involve computation of the cosine similarity metric using inverted indexes [13]. Techniques that are based on the pruning of the inverted index [11, 12] are close in spirit to our work, especially if we implement the sampling step using the ROUND function (Figure 3), which effectively prunes all the tokens with small weights.

Cohen's WHIRL system [3] is also highly relevant to our work. WHIRL reasons explicitly about text similarity to compute text joins using the cosine similarity metric. A key difference in our proposed techniques is our goal to perform text joins *within an unmodified RDBMS* for robustness and scalability. Grossman et al. [6] share this goal and present techniques for representing text documents and their associated term frequencies in relational tables, as well as for mapping boolean and vector-space queries into standard SQL queries. They also use a query-pruning technique, based on word frequencies, to speed up query execution. Finally, the sampling-based algorithm presented in Section 2.2 can be viewed as an instance of the approximate multiplication algorithm presented in [2]. The main difference is that our technique is adapted for efficient join execution. A direct application of the algorithm in [2] for approximate text joins would require a nested-loop evaluation of the join.

4. Discussion

We performed a thorough evaluation of our technique using different token choices (i.e., words and q-grams for different values of q) and comparing against alternative strategies. We do not report this evaluation here for space constraints. As a summary of our results, our proposed technique is orders of magnitude faster than the baseline technique of Figure 1. Furthermore, the answers that we produce are a good approximation of the exact text joins. We also performed a comparison with WHIRL, which showed the importance of leveraging an RDBMS for scalability. Using WHIRL, in contrast, was problematic for large data sets when we used q-grams as tokens. In conclusion, our experiments provide evidence that our proposed technique is robust and scalable for approximate text join computation.

References

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

[2] E. Cohen and D. D. Lewis. Approximating matrix multiplication for pattern recognition tasks. In *SODA 1997*.

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

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

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

[6] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *JASIS*,48(2), 1997.

[7] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4), 1965.

[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1), 2001.

[9] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD 2002*.

[10] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[11] A. Soffer, D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, and Y. S. Maarek. Static index pruning for information retrieval systems. In *SIGIR 2001*.

[12] A. N. Vo, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR 2001*.

[13] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, second edition*. Morgan Kaufmann Publishing, 1999.