Multi-column Substring Matching for Database Schema Translation

Robert H. Warren
rhwarren@uwaterloo.ca

Frank Wm. Tompa
fwtompa@uwaterloo.ca

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

We describe a method for discovering complex schema translations involving substrings from multiple database columns. The method does not require a training set of instances linked across databases and it is capable of dealing with both fixed- and variable-length field columns. We propose an iterative algorithm that deduces the correct sequence of concatenations of column substrings in order to translate from one database to another. We introduce the algorithm along with examples on common database data values and examine its performance on real-world and synthetic datasets.

1. MOTIVATION

As the number, size and complexity of databases increases, the problem of moving information where it is needed and sharing it is becoming an important one.

In the past, much work on database integration has been done to develop standards and interfaces to facilitate the transfer of the data. Application programming interfaces, such as JDBC and ODBC make it now possible to easily retrieve information from any table or column within most databases. With proper documentation of the database design and operation, a logical process can be written to integrate multiple databases together.

The integration process is driven by a database expert, and a great part of the problem is essentially a clerical process that has little value-add, except for the information extracted about the very high level semantics of the database. It is this clerical process that we aim to automate in our research. Whereas several projects have begun to tackle the problem from a top-down perspective, we use a bottom-up approach that is data-driven and that focuses on the matching and the translation of the data from one database to another.

Seligman et al. [19] have published a survey that ranked the acquisition of knowledge about the data sources as the data integration step that required the most effort. Large and complex industrial database schemas with over 10,000 tables and over 1,600 attributes per table are not unheard of, and even with good documentation, the search for the right matches is time consuming. Similarly, multiple standards exist to represent the same information in a concise format, and understanding which representation is in use takes time. For example, the Open Group lists 22 locales, each with its own typeset standard for date and time information [1].

This is why we are investigating tools that can automate the search for matching information within a database schema and infer a mechanism for the translation of the data from one representation to another. We have in mind situations where databases are numerous, large and complex and where partial automation of the process, even when computationally expensive, is desirable.

In particular, we wish to find a general purpose method capable of resolving complex schema matches made from concatenating substrings from columns within a database. While heuristics can be attempted for simple translation operations such as "concat (firstname, lastname) → fullname," no general purpose solution has yet been devised capable of searching for and generating translation procedures.

We wish to find a method capable of discovering a solution for problems as diverse as unknown date formats, unlinked login names, field normalisations, and complex column concatenations. Thus, we wish to find a generalisable method capable of identifying complex schema translations of the sort "4 leftmost characters of column lastname + 4 rightmost characters of column birthdate → column userid" or translating dates from one undocumented standard to another, e.g.:"2005/05/29 in database D → 05/29/2005 in database D'."

This paper describes a generalisable method that can be used to identify complex, multi-column translations from one database to another in the form of a series of concatenations of column substrings. The algorithm will discover translations as long as there exists overlap between the translated instances of the source and target schemas. To our knowledge, this form of matching is previously untried, and our solution is novel.

2. PREVIOUS WORK

Rahm and Bernstein present a general discussion and taxonomy of column matching and schema translation [17, 16]. They classify column matchers as having "high cardinality" when able to deal with translations involving more than one column. These types of matchers have been implemented on a limited basis in the CUPID system [12] for specific, pre-coded problems of the form "concatenate A and B."

As a means of abstracting away from the specific data being processed, Doan et al. proposed "format learners" [4]. These infer the

[1] `http://www.opengroup.org/bookstore/catalog/l.htm`

VLDB '06, September 12-15, 2006, Seoul, Korea.

formatting and matching of different datatypes, but the idea has not been carried forward to multiple columns. Recently, Carreira and Galhardas [1] looked at conversion algebras required to translate from one schema to another, and Fletcher [6] used a search method to derive the matching algebra. Embley et al. [5] explored methods of handling multi-column mappings through full string concatenations using an ontology-driven method.

The IMAP system [3] takes a more domain-oriented approach by utilising matchers that are designed to detect and deal with specific types of data, such as phone numbers. It also has an approach to searching for schema translations for numerical data using equation discovery.

We also use a search approach to find translations, but apply it to string operations. However, unlike IMAP, we do not assume that the record instances are pre-matched from one database to another. This makes the problem more difficult in that a primitive form of record linkage must be performed as the translation formula is discovered.

We attack the problem of schema matching and translation from an instance-based approach, where the actual values from individual columns are translated and matched across databases. This is done within the context of database integration, and our work is intended to be incorporated as part of a larger database integration system, such as IMAP, CUPID or Clio [21].

For example, in our model we assume that a specific 'aggregate' column and a number of potential 'source' columns have been tentatively identified by the database integration system. We accept that not all of the suggested source columns may actually be related to the target column and that a data-driven translation formula may discover a translation which is not intended. Our objective is to provide the integration system with possible translations formulas, with the understanding that some of these may be discarded by a higher-level component of the integration system in favour of another solution.

We have developed our solution to be as generic as possible, assuming only that the relational databases provide an SQL facility that can be accessed through an interface. As with the work of Koudas et al. [10], we have restricted ourselves to implementing our algorithms with basic SQL commands in an attempt to manipulate the data within the database systems. This is necessary to prevent the integration system from using excessive amounts of memory when dealing with large, complex databases and from over-burdening database communication systems.

3. PROPOSED APPROACH

Let us assume that we have a table T_1, which we term the source table for convenience, with columns B_1, B_2, ..., B_n. These columns may or may not be relevant to the translation. Similarly, we have a second table T_2, named the target table, with a single aggregate column A. The instances of T_1 and T_2 are available for retrieval, but no example translations are provided, nor are individual records of T_1 linked to their T_2 equivalents.

The operating algebra is simple, consisting of two operators: concatenate and substring. We wish to find a mapping such that many values in the target column A can be defined as a series of concatenation operations of the form $A = \omega_1 + \omega_2 + \cdots + \omega_\nu$, where each ω_i represents a substring function to be applied to some source columns B_j, and a single value for A is obtained when all functions are applied to a single row in T_1.

3.1 Principles of the approach

Let target tuple t' result from concatenating substrings from various fields within source tuple t. We will write $t' = t\ [\beta_1^{x_1 \cdots y_1} + \beta_2^{x_2 \cdots y_2} + \cdots + \beta_\nu^{x_\nu \cdots y_\nu}]$ if the i^{th} subfield of t' is taken from characters x_i through y_i from the β_i attribute value in tuple t.

We note that a source attribute B_j can contribute characters to several subfields in the target tuple (i.e., the β_i are not necessarily distinct), and in fact a particular source *character* may be copied to more than one target subfield; however, each target character, by definition, comes from only one source subfield.

This is a *search* problem (find a tuple in the source table that contains substrings from which the target tuple can be constructed) and an *optimization* problem (find a formula that can be reused to create as many target tuples as possible, each from its own source tuple while ensuring that the translation is concise). If the source table contains many columns and many tuples, and especially if some of the source columns are very wide, the search problem to match a single target tuple will have many potential solutions, and many of the potential source tuples will have many potential formulas that could be applied to form the target value; it is the optimization problem that dictates which of these solutions is most appropriate.

We have chosen a greedy algorithm to attack the optimization problem. Although not guaranteed to find an optimal solution, in practice this approach works well to find a conversion formula that produces many target tuples from the source table. Whether or not a sub-optimal solution is obtained, by removing all matched source and target tuples from the input and then repeating the process, more and more matches can be discovered.

If we could find a subfield in some source column for which many source tuples could contributes many characters to some target tuples, we would reduce our problem substantially. Thus our method comprises three steps: selecting an initial source column B_k, creating an initial translation recipe that isolates a substring ω_x from it, and then iterating for additional columns. The overall algorithm is shown in Algorithm 1.

Data: For a set $\mathcal{B}$ of columns B_1, B_2, ..., B_n and target A
Find column B_{start} most likely part of A;
Generate a translation τ partially translating B_{start} to A;
while τ has unknowns **do**
  foreach Column $B_k \in \{B_1, B_2, ..., B_n\}$ **do**
    Sample rows from T_1 and select values of A matching partial translation τ;
    Generate a new τ' partially translating B_k to A;
    Score each τ', B_k;
  end
  Insert highest ranked τ' to be part of translation τ;
end

Algorithm 1: Overall algorithm

In the first step, all source columns are scored to identify those most likely to be part of the target column. This step serves as a filter to eliminate all but the most productive column from the more expensive computation in Step 2. We use the identified column to create an initial translation formula, which partially maps the source column to the target column. Using this coarse translation formula, we iterate through additional substring selections from any column until either a complete translation formula has been found, or the addition of more substrings no longer provides additional information.

Instead of iteratively determining additional contributing subfields, one alternative approach would be to identify all possible solutions to the search problem, and then determine which of these are applicable to many tuples. This is clearly infeasible because of the large number of potential solutions for a single target tuple (which *a priori* could have been produced from any source tuple).

Alternatively, we could try to identify several possible starting points that apply to many tuples, possibly using subfields from several source targets, and then determine which of these fit together to form the beginning of a solution to the search problem. In practice, however, a target column is often produced from one wide subfield and several very narrow ones (see, for example, Table 1). To find such solutions with this alternative approach, we would need to include many narrow source fields among our potential starting points. This would result in an inordinate number of false potential mappings before any pruning could be applied, especially if the source table includes one or more wide columns.

In the following sections, we review each of the steps of our approach in detail, using examples based on the data contained in Table 1.

Source				Target
first	middle	last	...	login
robert	h	kerry	...	nawisema
kyle	s	norman	...	jlmalton
norma	a	wiseman	...	rhkerry
...	...	...	...	...
amy	l	case	...	alcase
josh	a	alderman	...	ksokmoan
john	l	malton	...	ksnorman

Table 1: The first sample problem, where login names must be matched to the columns of an unlinked table.

3.2 Beginning the search

In order to choose candidate columns and generate possible translation formulas for very large tables, we need a method to sample values from the source columns. The objective is not to get an optimal column selection as much as to identify a feasible one. In effect, we are trying to "bootstrap" the translation with a single useful column from which we can begin to look for additional contributing subfields.

With the example in Table 1, we would prefer to pick the column last, out of all possible candidate columns, because it has the most overlapping data with the target column login. On the other hand, we can tolerate picking instead any of the other related columns. This must be done in a manner that is simple and that will take into account that the match between source and target column instances is imperfect. In this section, we describe how we select the first source column, as detailed in Algorithm 2.

For each candidate source column, we first sample a pre determined fraction of the *distinct* values within the column, yielding t values. We use distinct values to prevent the value distribution in the source column from influencing the number of matches. The sampling of the source column is done in an interleaved manner, where values are taken at equally distanced rows. Gravano et al. [7] found this sampling to be as good as random sampling, but much less expensive since a database cursor can be used to retrieve each value in a single step. (It is even more efficient if the column has a sorted, e.g. B-tree, index.)

We next use each of those t values from the source column to produce a larger set of q-grams [20], that is, q-length subsequences of consecutive characters from each string. As an example, the string *possible*, contains five 4-grams, namely *poss*, *ossi*, *ssib*, *sibl* and *ible*, and in general, a string of length n contains $n - q + 1$ q-grams. We use the set of q-grams obtained from the t sample values as search keys for the target column. We then count the number of matches in the target column and normalise the count to yield a score that reflects the length of the common substrings and

set B_{best} to null;
set $score_{best}$ to 0;
foreach column B_k of T_1 **do**
  count distinct values of B_k as *dcount*;
  set $t = dcount * fraction$;
  HitCount =0;
  for $j = 1$ **to** t **do**
    get value key from column B_k in tuple $\frac{j}{fraction}$;
    $localc$ = count T_2 where A includes q-grams of *key*;
    $HitCount += \frac{localc}{length(key)}$;
  end
  ScoreCol(B_k) = $\left(\frac{HitCount}{dcount/10}\right)^q$;
  if score(B_k)> $score_{best}$ **then**
    $score_{best}$ = ScoreCol(B_k);
    B_{best} = B_k;
  end
end

Algorithm 2: Initial column selection using a fixed q-gram and sample size.

the average record overlap between the source and target columns (see below). We choose the starting column for our translation to be the one that generates the highest score.

$$\text{ScoreCol} = \left(\sum_{j=1}^{t} \frac{HitCount(j)}{t * length(key_j)}\right)^q \quad (1)$$

More specifically, Equation (1) re-expresses the column scoring function from Algorithm 2 in a single expression. It serves as a cheap filter to eliminate all but the most productive column from the more expensive computations in Step 2. The number of distinct hits for each key ($HitCount(j)$) is divided by the length of the key ($length(key_j)$) and by the total count of distinct values (t) sampled within the source column. This yields the average overlap. By raising this value to the power q, we account for the decreased probability of this substring occurring randomly in the target. Note that by definition q must be equal to or smaller than the narrowest column being searched.

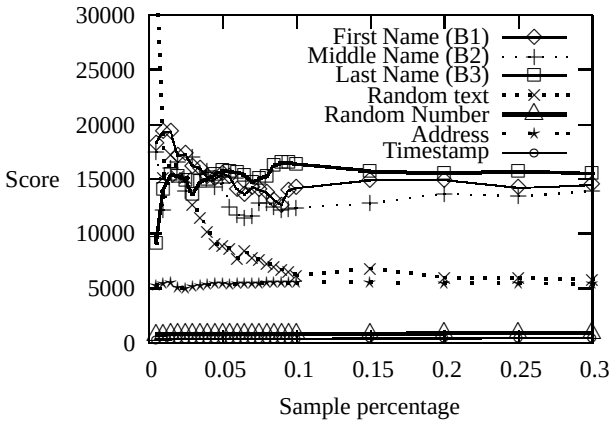

Figure 1: Effect of sample size on scores.

Figure 1 and Table 2 represent empirical evidence of the algorithm's performance when it is used on a sample dataset similar in nature to the data in Table 1 and sized at 6,000 rows. To verify the robustness of the method, we included several noise columns in the source table, including columns containing random character strings, time-and-date values, random numbers, and random street addresses. Figure 1 shows that the column scoring function works extremely well using 10% of the distinct values in each source column.

first	middle	last	text	time	numb.	addr
14194	**12391**	**16374**	6151	354	792	5505

Table 2: Score results generated with a 10% sample.

An additional experiment shows that the column scoring function works surprisingly well even with a very small sample when the dataset is very large. Figure 2 plots the results of the column selection formula on a dataset containing over 700,000 concatenated first and last names to be matched against a table with first name, last name, random text, and random addresses. Even with a very small sample of several hundred rows, the column selection order reflected by the scores is accurate.

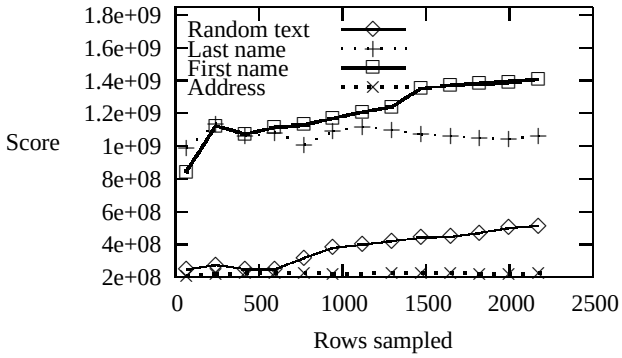

Figure 2: Effect of sample size on scores for a large dataset.

3.3 Creating an initial translation formula

With a specific column B_k selected as a starting point, we next need to create a partial translation formula to transform values from B_k to values found in A. To do this, we need to retrieve instances of A that are similar to the sampled values from the current source column B_k. We can then use each sampled value of B_k and the similar A entities to discover a partial translation formulas $A = \omega_1 + \omega_2 + ... + \omega_i$ that applies to many of the source values.

3.3.1 Identifying candidate pairs

Recall that, for the sake of efficiency, we are dealing with only a sample of the chosen source column's values. Thus we first require a method that will retrieve similar entities from the A column for each of the values sampled from the source column B_k. In identifying the best column B_k, we found tuples from column A based on the occurrence of any q-gram element from the sampled value. While this method was satisfactory for ranking columns, it is inadequate for finding suitable matches for specific source values. In particular, it suffers from low precision due to the serendipitous occurrences of q-gram elements.

B_k (last_name)	A
warner	rhwarner
	klwarder
	ghkarer
amy	laramy
	amyrose
	camyro
wang	mkwang
wayne	opwayne

Table 3: Instances of A sufficiently similar to B_3.

When trying to identify values from the target column that match a specific source value, another possibility is to rank target values according to the number of q-grams of the sampled column B_k that are matched. Hence, with bi-grams "ab," "bc," and "de" the instance "abcd" would score lower than the instance "abcde," in the manner of Equation 2.

$$score(a,b) = \sum_{n=1}^{j} \begin{cases} 1, & \text{if instance } a \text{ of } A \text{ has } q\text{-gram}_j \\ & \text{of instance } b \text{ of } B_k. \\ 0, & \text{if not.} \end{cases} \quad (2)$$

This improves our precision in that the entities that have the most elements in common with the sampled value will be ranked highest. However, this still does not take into account the relative frequencies of q-grams and can improperly rank some entities that contain many commonly occurring q-grams over extremely rare and relevant q-grams.

We can correct this by borrowing methods from the information retrieval community. Koudas at al. [10], Chaudhuri et al. [2] and Gravano et al. [7] all use variations of this approach to match similar records using td-idf and cosine similarity [18]. This is done by assigning a weight to each q-gram that represents its relative significance within the database.

Equation 3 represents the tf-idf formula for calculating a weight for each q-gram: w_{ij} is the weight assigned to q-gram j for instance i of column A, where tf_{ij} is the frequency of q-gram j in instance i in column A, N is the number of instances in column A, and n is the number of instances in column A where q-gram j occurs at least once. Equation 4 then represents the scoring function for a pair of values from A and B_k.

$$w_{ij} = tf_{ij} * log_2(N/n) \quad (3)$$

$$\text{ScorePair}(a,b) = \sum_{n=1}^{j} w_{aj} * w_{bj} \quad (4)$$

Thus, to find pairs of similar values from the two columns, first a sample of values are chosen from column B_k. Then for each source value, the target table is queried for values having have scores from Equation 4 that exceed a given threshold. Such generated pairs, as in Table 3, are then passed on to the next phase of processing. (Note that as an alternative to keeping all pairs with scores above a given threshold, the top r ranked pairs could be retained instead.)

3.3.2 Creating edit recipes for pairs

With a set of pairs of similar instances from column A to column B_k (Table 3), we next find a partial translation formula that will match the common information between the two sets of column instances. We achieve this by looking for longest common substrings

between the pairs of column instances. By keeping track of the locations of the common substrings over several samples of B_k, we can both infer the correct area within the target column A that is related to B_k and what area of B_k is matched.

We characterise a translation formula for a single subfield as taking characters from certain consecutive positions in some value from B_k and inserting them into templates for A by assigning them to a specific location within the target value. For our purposes, we use the term *recipe* to characterise such insert operations, and henceforth the term *region* refers to any consecutive series of characters taken from B_k. For example, one (partial) translation formula relating the instance "warner" to "rhwarner" would be "%B_3[123456]" which states that characters 1 through 6 from column B_3 are to be mapped to something (as yet unknown) followed by that region.[2]

To discover appropriate recipes for a single pair of source and target values, we must be able to describe the shortest editing sequence required to transform one string into another. Although Levenshtein distance [11] provides us with the minimum number of operations to transform the first string into another, it does not produce the actual operations used. However, Paterson [15] provides a good survey of several algorithms available to solve the problem. For example, Hirschberg [8] describes a method which is optimised for the maximal common subsequences in O($|s_1| * |s_2|$) time. Hunt and Szymanski [9] provide an interesting solution of complexity O($(n + R)$ log n) where n is the length of the longest string and R is the number of substring matches between the two strings. Most of these methods rely on a matrix of operations similar to Table 4 which illustrates the different matches possible for strings "rhwarner" and "warner."

	r	h	w	a	r	n	e	r
w	R	R	=	I	I	I	I	I
a	R	R	R	=	I	I	I	I
r	=	R	R	D	=	I	I	=
n	D	R	R	R	D	=	I	I
e	D	R	R	R	D	D	=	I
r	=	R	R	R	=	D	D	=

Table 4: The longest common string (underlined). "R" stands for a replaced character, "I" for an inserted character and "D" for a deleted one.

The highlighted path contains the longest common substring between the two strings. We select this partial path and complete the recipe using the edit distance metrics to find the lowest-cost path before and after the longest common substring. In case we find two equal-length common substring, we arbitrarily select the leftmost string. Selecting potential matches and creating initial recipes is summarised in Algorithm 3.

3.3.3 Creating a partial translation formula

From these recipes derived from pairs of tuples, we must now create a partial translation formula (ω_n) that is inferred from all of the collected recipes and that can be applied to the source and target tables as a whole. This is done by creating a candidate ω_n from each individual region within a recipe. Then, we collate the candidate translations and select the one that occurs most often. Algorithm 4 explains this process in pseudo-code, and we discuss it here in detail.

[2]We use the convention that % signifies any match.

Data: A candidate column B_k
Result: Edit recipes $\mathcal{R}$
count distinct values of B_k as *dcount*;
set $t =$ *dcount* $*$ *fraction*;
$\mathcal{R} =$ null;
for $j = 1$ **to** t **do**
  get value key from B_k in tuple $\frac{j}{fraction}$;
  retrieve set $\mathcal{A}$ from T_2 where ScorePair(a, *key*) exceeds threshold;
  foreach a **in** $\mathcal{A}$ **do**
    Recipe $R =$ edit-distance(*key*, a) ;
    if $R \in \mathcal{R}$ **then**
      increase count for R entry by 1 ;
    else
      create new entry in $\mathcal{R}$ for R with score 1;
    end
  end
end

Algorithm 3: Creating an initial set of recipes from a candidate.

As each recipe is processed, its known and unknown character sequences are translated into a series of regions. Each region ω_x represents a string element either from an unknown source or copied from specific character positions within a designated source column. The sequence of these regions $\omega_1+\omega_2+...+\omega_i$ describes a translation formula which provides a partial method to translate the information from the set $\mathcal{B}$ of source columns to the target column A.

As ω_n represents a fragment of one of the source columns B_k being copied, we need a model for the copying operation. A possibility is to create a regular expression using the recipes as examples. Instead of such an expensive general approach, we use the absolute character positions within the source columns, and build the translation as a sequence of these column references. This method has the advantage in that it provides some support for columns of both fixed and variable lengths.

For fixed-field data, it is straightforward to identify the commonly repeating recipes, because the absolute locations of the overlapping substrings will always align across recipes. Any superfluous matches (that is, other characters matching the overlapping field) will occur infrequently enough that the outlier recipes can be recognised and discarded.

For variable-length fields, however, the problem is slightly more difficult as the absolute locations of the matching values are not aligned. Thus we need to add some provision to the edit program to handle these situation. When generating the absolute character positions of the source column, we check if the region stops at the end of the string. If it does, we generate an additional copy of the translation where the current region is explicitly marked as copying the remainder of the string.

Furthermore, by having the translation behave as a sequence, the relative ordering in which the substrings occur is preserved. This allows us to deal with problems such as the dataset in Table 1, where the column widths are variable. Neither of these properties hinder fixed-width columns and thus our solution remains generalisable. Our editing algebra and edit distance methods cannot accommodate all specification of substrings (e.g.: the second-to-last character); however our simple algebra is sufficient for most practical purposes.

Table 5 represents the partial translations that were derived from the recipes generated in Section 3.3.2. As explained earlier, the

```
Data: Edit recipes R
Result: Partial translation formulas T
foreach R in R do
    create empty T ;
    begin region ;
    foreach char in R do
        if key chars still in sequence then
            region continues ;
        else if 1st char is from key then
            region continues ;
        else if region still unknown then
            region continues ;
        else if 1st char unknown then
            region continues ;
        else if known region ends on key boundary then
            clone region ;
            mark cloned region as end-of-string;
            link both regions to end of T chain ;
            begin region ;
        else
            (un)known region or recipe ends
        end
        link regions to end of T chain ;
    end
    if T ∈ T then
        increase count of T entry by 1 ;
    else
        create new entry in T for T with score 1;
    end
end
```

Algorithm 4: Generation of translation formulas from recipes.

typesetting convention used is % for any unmatched region and column[n] for matched characters, where n refers to the nth character of the source column named Column. Note that in several cases, two different translations are produced for a single recipe.

Not all recipes will represent correct matches. For instance, "warner" is similar to both instances "rhwarner" and "klwarder" with only "rhwarner" being an actual match. However, serendipitous matches are probabilistically unlikely to occur at the same positions and sequence number.

We select the translation that occurs most frequently and discard the others. For the example in Table 5, we would pick %B_3[1–n] since it occurs most often. The partial translation formula then becomes the starting point for searching the rest of the database.

3.4 Selecting additional columns

We now begin an iterative process to reduce the sizes and number of unknown regions within the translation formula by finding additional fragments of source data that match the target values. The partial translation we have already found induces a mapping from values in the start column, and hence rows in the source table, to values in the column table. Thus the only data fragments that are available for providing additional information to the target value are the ones contained within any of the fields of a corresponding row from the source table.

For example, in the first relation in Table 1, if we have found that instance "kerry" from column last is mapped to instance "rhkerry" from column login, then for column first to also be involved in the translation, instance "robert" from that same source row must contribute some data to that same target instance "rhkerry." This restriction on the instances that is provided by the relation allows us to restrict our search to values and columns likely to form part of the target column translation. This is captured in Algorithm 5, which is described in the remainder of this section.

Column		$\omega_1 + \cdots + \omega_n$
B_3	A	
warner	rhwarner	%B_3[123456] or %B_3[1-n]
	klwarder	%B_3[123]%B_3[56] or %B_3[123]% B_3[5-n]
	ghkarer	%B_3[23]B_3[56] or %B_3[23]B_3[5-n]
amy	laramy	%B_3[1]%B_3[123] or %B_3[1]%B_3[1-n]
	amyrose	B_3[123]% or B_3[1-n]%
	camyro	%B_3[123]% or %B_3[1-n]%
wang	mkwang	%B_3[1234] or %B_3[1-n]
wayne	opwayne	%B_3[12345] or %B_3[1-n]

Table 5: Sample edit recipes for the login data, where B_3 is used in place of last_name.

Whereas initially we first selected a column and then created a translation from that column, we now create translations for all candidate columns and then select the best translation regardless of column. The algorithm depends on two functions, CreateRecipes() and ScoreTrans(), for which details are given in the following subsections.

The search for improved translation formulas is done by considering each potential column for new recipes, generating alternative translation formulas based on the obtained recipes, and selecting the highest ranked translation formula based on a scoring formula. This process follows the same basic steps as those described in Section 3.3, namely, find pairs of matching rows, derive edit formulas, and create the best translation formula. However, each step is modified to account for the partial translation formula already chosen.

3.4.1 Identifying refined candidate pairs

As before, for each candidate column, we begin by equidistantly sampling instances from that column. However, we retrieve not only the values for the candidate column, but also the corresponding values for the source columns that are already part of the translation. That is, instances from all source columns are preserved together through T_1, as in Table 7.

Then, as in Section 3.3.1 we retrieve similar instances from the target column A. However, instead of merely searching for matching q-grams, we now refine the search for instances that respect the partial translation that we have developed so far. Hence, should our partial transformation be %last[1-n], the instance of last be "kerry" and the candidate instance for middle be "henry," candidate target values must end with the five characters "kerry" and have some substring of "henry" within the preceding region.

This has the effect of reducing the number of incorrectly retrieved instances from the target column, because we are actively enforcing the elements of the translations that we have decided upon and only producing candidate pairs that refine the partial translation. The resulting record linkage constraint also prevents sampled rows with no equivalent target instances from generating serendipitous recipes.

```
Data: A set of candidate columns B, a partial translation T
Result: A new translation T
foreach column B_i in B do
    R = CreateRecipes(B_i, T);
    foreach R in R do
        create empty T_new ;
        begin region ;
        foreach char in R do
            if key chars still in sequence then
                region continues ;
            else if 1st char from part of T then
                region continues ;
            else if region still unknown then
                region continues ;
            else if 1st char unknown then
                region continues ;
            else if known region ends on key boundary then
                clone region ;
                mark cloned region as end-of-string;
                link both regions to end of T_new chain ;
                begin region ;
            else
                (un)known region or recipe ends
            end
            link regions to end of T_new chain ;
        end
        if T_new ∈ T then
            increase count of T_new entry by 1 ;
        else
            create new entry in T for T_new with score 1;
        end
    end
end
Init T_best to have score 0;
foreach T in T do
    if ScoreTrans(T) > ScoreTrans(T_best) then
        T_best = T;
    end
end
return T_best;
```

Algorithm 5: Selecting additional columns.

3.4.2 Creating edit recipes for refined pairs

In Section 3.3.2 we used a combination of an edit-distance and longest common substring method to identify common information between the instances. We do so again here, but add a constraint that only characters from the target column that are not known to be part of the partial translation formula can be used for matching. This both prevents the algorithm from assigning the same target region to two source columns and also diminishes the run-time for the task.

Table 6 graphically represents the matrix of operations for comparing instance "henry" to "rhwarner" from Table 1, where the target has been masked to remove regions already covered by the partial translation formula. In this case, two possible recipes are present and both substrings have the same length; thus we select the left-most, or earliest occurring, recipe as indicated, leading to the refined translation formula %first[1-1]last[1-n].

Similarly, recipes are generated for all retrieved instances that are matched to the values sampled from the target table. From these recipes, we next create new translation formulas that combine both the information from the old formula and the information within the recipes.

	r	h	~~w~~	~~a~~	~~r~~	~~n~~	~~e~~	~~r~~
h	R	=	X	X	X	X	X	X
e	R	R	X	X	X	X	X	X
n	R	R	X	X	X	X	X	X
r	=	R	X	X	X	X	X	X
y	D	R	X	X	X	X	X	X

Table 6: Restricting the search for the first longest common substring (underlined).

3.4.3 Improving the partial translation formula

As in Section 3.3.3, we use each recipe to create a new translation formula, containing both previously selected columns and the current candidate column. Algorithm 6 encodes the function CreateRecipes() that is repeated called from Algorithm 5.

```
Data: A candidate column B_k, a candidate translation T
Result: Edit recipes R
for B_k and all columns in T do
    count distinct relations as dcount;
    set t = dcount * fraction;
end
for j=1 to t do
    Initialize SearchPattern;
    foreach region in T do
        if region is known then
            Get value of region column;
            Extract substring from column;
            Add substring to SearchPattern;
        else
            SearchPattern =+ '%';
        end
    end
    get value key from B_k ;
    Create set A from T_2 where A matches SearchPattern and
    contains q-grams of key;
    foreach candidate in A do
        Set c = candidate masked by SearchPattern;
        Recipe R = edit-distance(key, c) ;
        if R ∈ R then
            increase count of R entry by 1 ;
        else
            create new entry in R for R with score 1;
        end
    end
end
```

Algorithm 6: Creating edit recipes for a new candidate column.

Table 7 represents the new candidate translation formulas created from combining the previous partial formula and the new recipe. All of the candidate translation formulas are collated according to a complete match between the source columns, the sequence of their individual regions and the character positions within the source columns.

3.4.4 Scoring and selecting an improved translation formula

Because we are ranking multiple translation formulas from multiple candidate columns concurrently, we need to be able to score

source		target	Translation	
B_3	B_1	A	Previous	Candidate
kerry	robert	rhkerry	$\%B_3$[1-n]	B_1[1]$\%B_3$[1-n]
	robert	klkerry	$\%B_3$[1-n]	$\%B_3$[1-n]
	robert	gkerry	$\%B_3$[1-n]	$\%B_3$[1-n]
kyle	otto	opkyle	$\%B_3$[1-n]	B_1[1]$\%B_3$[1-n]

Table 7: Improved translation formulas based on partial recipes.

translations in a normalised manner. To do this, we use the function ScoreTrans(τ_j) to score the individual translations based on both the number of their occurrence and the source column (B_i) in use.

We found experimentally that with large ($>$ 500,000 rows) and wide columns ($>$ 80 characters) of random characters, the resulting serendipitous matches would increase noise to unacceptable levels. It is doubtful that a noise column of this type would arise in a realistic database integration problem, however we provide it as a worst-case scenario for study.

$$\text{ScoreTrans}(T\tau_j) = \frac{\textit{Frequency}(\tau_j)}{max(1, \textit{AvgLength}(B_i) - \sigma)} \quad (5)$$

Formula (5) scores candidate translations based on a per-column normalised occurrence score, but also penalises the score for using wide columns. The intuition behind the solution is to skew the selection of columns towards those that provide a concise answer and thus avoid serendipitous matches on large text fields. The term *Frequency* is the occurrence count of the candidate translation τ_n normalised to the total number of translations created by its parent column B_i. The denominator $max(1, \textit{AvgLength}(B_i) - \sigma)$ is a penalty term that was added to deal with especially noisy columns and that provides a gradual back-off for long strings. More specifically, the σ parameter prevents columns with less than a certain average width from begin penalised, while the max term prevents the denominator from being negative and ensures a mathematically well-behaved function. Experimentally, we determined that columns with an average length of over 4 characters ($\sigma = 2$) should be moderated by this penalty term. We also make an explicit decision not to implement backtracking in our method: this would only be worthwhile if the overall database integration system was capable of providing feedback on translation formulas, and we make no such assumption.

4. EXPERIMENTAL RESULTS

We implemented this method using the PostgreSQL [13] DBMS and a Java application front-end. We used bi-grams (i.e., $q = 2$) for scoring purposes and simple bi-gram matching for the retrieval of similar instances. This choice for q is easy to implement although precision is adversely affected (i..e., many spurious matches are found initially). As will be seen from the results, the effectiveness for finding matches is very good, in spite of the potential loss in precision.

Recipe generation was implemented using a modified Hirschberg [8] algorithm and an edit distance method as described by Monge et al [14]. Sensitivity experiments showed that the specific cost values for copy vs. deletion vs. replacement were not critical and that a value of 1 was reasonable for all edit costs.

We experimented with several different datasets. Unless noted otherwise, 10% samples were used for all experiments, and a series of noise columns were always added to the source table T_1 so that finding which source columns contribute to the target was not trivialised. More specifically, the extraneous columns included columns filled with random numerical data, random alphanumeric data, street addresses, and a full length RFC-2822 timestamp. The objective was to add enough data to ensure that the column selection made by the method was not serendipitous, and that the algorithm would work well in the presence of noise.

In the following experiments, small examples were resolved in less than 5 minutes, and runtimes for the larger problems were about 15 minutes, including instrumentation overhead.

4.1 UserID dataset

The first experiment was to match a listing of users' first, middle, and last names (with additional noise columns) against Unix login names extracted from our university's undergraduate computing systems (Table 1). The tables have about 6,000 rows in random order, and several different translation formulas are known to exist to create login names from the actual names. Our search algorithm returned the translation formula login = first[1-1] + last[1-n], which is, in fact, the most commonly used translation formula, accounting for about half of the tables' rows.

As part of our implementation, we added a facility to create SQL statements that would perform the translation. In the above experiment, the corresponding SQL query was:

```
select substring(first from 1 for 1) || last as login from table where first is
not null and char_length(substring(first_name from 1 for 1))=1 and last_name is
not null and char_length(last_name)>= 1
```

If we remove from both tables the records translated by this formula, and reapply the algorithm on the remaining rows, the method returns the next dominant translation login = first[1-1] + middle[1-1] + last[1-n], which covers about 1,200 rows. Inspection of the tables revealed that the remainder of the userids followed no other dominant pattern.

The results are not surprising in that the tables in this dataset are balanced, e.g., for each row in the source table T_1 there exists a row in the target table T_2. We attempted a second experiment with this dataset that added several rows of instances to each of the source columns. We selected these instances from another unordered set of first, middle and last names and inserted them incrementally along with new noise column values into the source table.

We found that with this dataset, the method would tolerate an additional 3,000 rows of source data (i.e., approximately one-third of the records were unmatched) before it made a wrong column selection. As it turned out, the algorithm correctly selected the last name as being a part of the userid, but then incorrectly selected a noise column for improving the translation.

4.2 Time dataset

Data similar to that in Table 8 was created using 10,000 randomly generated time-stamps, which were then merged into a single string. For this experiment, the correct translation from source to target column involved no substrings, only simple concatenations.

The same noise columns were used as for the first experiment. The returned SQL translation query was:

```
select substring(hour from 1 for 2) || substring(minutes from 1 for 2) ||
substring (seconds from 1 for 2) as fulltime from table where hour is not null
and char_length (substring(hour from 1 for 2)) = 2 and minutes is not null and
char_length(substring(minutes from 1 for 2)) = 2 and seconds is not null and
char_length(substring(seconds from 1 for 2)) = 2
```

which corresponds to the correct translation formula time = hour[1-2] + minutes[1-2] + seconds[1-2]. This experiment shows that even when sources columns are short, and the values in those columns come from highly overlapping domains, correct table matches can

Source				Target
secs.	mins.	hrs.	...	time
55	59	02	...	345407
43	23	05	...	330011
12	55	07	...	135741
...	...	...	...	...
33	00	11	...	004107
34	54	07	...	192609

Table 8: Time-stamps in single and multiple columns.

be found because of the properties of record linkage incorporated into the algorithm.

4.3 Name concatenations dataset

For the next experiment, we used a list of names to create data such as that shown in Table 9, where the first and last names are merged into a single column. For this experiment, the table contains about 700,000 rows with about 70,000 unique values in both source columns. The same noise columns were again used.

Source			Target
first	last	...	full
robert	kerry	...	robertkerry
kyle	norman	...	kylenorman
norma	wiseman	...	normawiseman
...	...	...	...
amy	case	...	amycase
josh	alder	...	joshalder
john	galt	...	johngalt

Table 9: Merged names dataset.

The target column full was generated using the translation full = first[1-n] + last[1-n], and as expected, the SQL translation query returned by the algorithm was:

```
select first||last as full from table where first is not null and char_length
(first)>=1 and last_name is not null and char_length(last_name)>=1
```

4.4 Citeseer dataset

We next used the Citeseer [3] citation indexes to provide an additional real-world translation problem. We pre-processed 526,000 records into a table containing columns for the year of publication, the title, and a series of 15 columns, each of which contains the name of a single author (up to 15). We then created a new table citation from the concatenation of the year of publication, title, and first author for all 526,000 records (and stored in a randomly shuffled order). This provides a test to study how our method performs on a dataset that has many tuples and many similar columns (each representing one author).

To further examine the robustness of our algorithm, we chose a sampling size of only 1% of the distinct values from each column. Even with such a small sample size, we were able to extract the correct transformation formula: citation = year[1-n] + title[1-n] + author1[1-n]. The prior examples were all resolved in less than 5 minutes elapsed time on a Sunfire v880 750MHz machine. In spite of the size of the problem (526,000 rows in each table and 17 columns in the source table, 15 of which have values from a single domain), the run time for this example was under 20 minutes in that same environment. More detailed analysis is provided after examining the results of our final experiment.

[3] http://citeseer.ist.psu.edu/oai.html

4.5 Cross dataset translation

A question that remained was how well the method would work when very little overlap exists between the source and target tables. To answer this question we designed an experiment where we attempted to link the citation column of the Citeseer data to the DBLP citation index [4].

This is a very hard problem, because although we expect that there should be overlapping citations, the citations often have misspellings, incomplete author lists, and incompatible abbreviations. We pre-processed the DBLP data in a manner similar to the Citeseer data and obtained a 17-column table with 233,000 rows.

While the maximum number of matches between both tables can be no more than 233,000, closer examination showed that there exist only 714 records that match based on an exact match of the year, title, and author1 data columns. Hence, when attempting to find a translation formula for the citation column from the Citeseer dataset to the DBLP dataset, not only must we sort through 17 columns to find the correct ones, but we must also deal with a very low number of overlapping records.

Surprisingly, our program did not return the expected translation formula, but instead returned the formula year [1-n] + title[1-n] + author2[1-n]. Subsequent examination of the tables revealed that there exist 378 records within the Citeseer dataset that are also present within the DBLP dataset, but with the first and second authors reversed! Removing the matched records and re-running the program then produced the expected formula.

While the first translation found actually occurs less often than the expected translation, both have a very low frequency of occurrence within the datasets: much less than 0.5% of the source records are involved. Which of the two correct solutions is returned first is determined by which tuples happen to be sampled from the database.

What is interesting in this experiment is that the first translation formula found by our method matches a block of articles within the Citeseer dataset with inverted first and second authors. Although unintended when we designed this experiment, we have shown that our method does in fact identify previously unknown relationships between datasets! This result supports our motivation that tools for data conversion must operate in environments where the schemas are only partially understood.

5. ALGORITHMIC ANALYSIS

The computational complexity of the algorithm described in this section is dominated by the number of select operations that must be performed to match source tuples in table T_1 to target tuples in table T_2. Let s_1 be the number of tuples in T_1 and s_2 be the number of tuples in T_2. Let n be the number of potential source columns from T_1, and let w be the maximum number of characters in any value in the target column in T_2. The worst case time is therefore $O(w * n * s_1 * s_2)$. The proof of this claim follows from the observation that the algorithm is dominated by the step described in Section 3.4, where on each iteration, for each source column, samples are selected, and for each sample, the target column is searched for matches. Since each iteration determines an additional region of the target that is included in the formula, at most w iterations are needed. In practice, however, regions are larger than one character each, only a small fraction of s_1 is required, and a smaller fraction of the s_2 target values are matched with each new iteration.

This can be clearly observed in Figure 3, which plots the cumulative time spent up to the end of each step of the method for various

[4] http://dblp.uni-trier.de/xml/

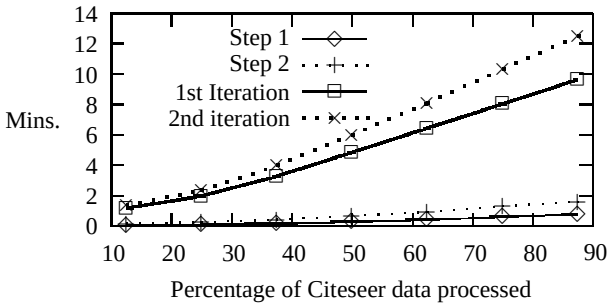

Figure 3: Wall clock time versus Citeseer dataset size.

subsets of the Citeseer citation example.[5] What is evident from inspecting the plot is the dis-proportionately high cost of searching for the second column during the first iteration of our search: for that step, the constraints on retrieving instances are few and we must search all of the columns.

This also shows the performance bottleneck of the method: the computational balance between retrieving similar instances (database I/O) and the quadratic time for the longest common substring for each string pair (client in-memory). The trade-off should favour efficient instance retrieval with good SQL engines when the client has limited capacity. This motivates the algorithms behind Sections 3.2 and 3.3 where the column is selected before recipes are generated. Notice that in Figure 3, both these operations are less costly than the first iteration.

The overall method has shown itself to be relatively insensitive to the size of the sample, much in the manner of Figures 1 and 2. Hence, it is acceptable to lower the sample size to very low values to deal with very large datasets. As demonstrated by the final experiment, in practice, only a few dozen 'good' samples are required for the method to function. Datasets with several million rows eventually require and justify the computational overhead for high precision instance retrieval methods, described in Section 3.3.1. The overall method in itself remain unchanged for very large datasets. Choosing sample sizes is problematic only when the overlap between datasets is unknown. We must ensure that some of the rows that are sampled have a reasonable expectation of being present within the other table. In future work, we wish to look at possible solutions to estimate the overlap and automate the selection of the sampling parameter.

6. SEARCHING FOR SEPARATORS AND MANY-TO-MANY TRANSLATIONS

In this section we review two additions to this method that allow it to deal with non-alphanumeric data separators (e.g., the hyphens in a date string "2-15-2005") and with many-to-many translations.

6.1 Non-alphanumeric separators in columns

The method as described so far deals well with translations that are composed exclusively from the data contained within the source columns. However for many reasons, including esthetic, historical, and error-checking concerns, separators are often present within the data. Examples include dates "2/15/2005", times "11:45:34", manufacturing part numbers "FRU-13423-2005", field delimiters "field a, field b, field c" and phone numbers "+1-321-555-1212".

[5]Recall that the experiment was run on a Sunfire v880 750MHz machine with 1% sampling.

A simple solution to this problem could be to assume that the separator will be found in the other database. However, such an assumption is inappropriate for serious database integration work. To the best of our knowledge, no previous work exists on the problem of finding separators within database elements.

We make the assumption that a separator character is not alphanumeric, that it occurs in all target column instances without exception, and that it is not to be copied over from any of the source columns. We attack this by querying the target column for consistent patterns of separator uses and then forcing the use of a separator template on the identification of similar pairs and on recipe generation.

Data: A target column A
Result: SearchKey: A representation of the separator pattern
SearchKey = null;
for $j = 1$ **to** *length(A)* **do**
  if charAt(j) is a separator character && all charAt(j) are the same **then**
    SearchKey = SearchKey + charAt(j);
  end
  else
    SearchKey = SearchKey + '%';
  end
end

Algorithm 7: A simple algorithm for finding separators.

Algorithm 7 represents a simple algorithm for creating a separator template representing the placement and values of the separators in a database column. For example, given a column of instances of timestamps of the form "11:45:34", the algorithm would return a separator search pattern of the form "%:%:%." We then use this pattern in two ways. First, whenever we search for similar instances within the target column, we make sure that search terms (individual q-grams) do not contain separators. Thus, we would not use a search key such as ":4" to search a timestamp column, as this would retrieve too many instances. Secondly, when building recipes, we use the characters deemed to be separators to align editing and translation generation, such as shown in Table 10. This essentially forces the method to generate aligned recipes whose translations will automatically match the column pattern.

$$\begin{pmatrix}
 & 0 & 4 & : & 1 & 2 & : & 5 & 3 \\
0 & = & I & I & I & I & I & I & I \\
4 & D & = & I & I & I & I & I & I \\
: & D & D & = & I & I & = & I & I \\
1 & D & D & D & = & I & I & I & I \\
2 & D & D & D & D & = & I & I & I \\
: & D & D & = & D & D & = & I & I \\
7 & D & D & D & D & D & D & R & R \\
3 & D & D & D & D & D & D & R & =
\end{pmatrix}$$

Table 10: The separator ":" aligns the strings.

This approach, however, is too simplistic: it cannot deal with both fixed and variable length target columns. An example of the need for a more general method is illustrated by the data in Table 11. In one database, the names are inserted into two columns while in the second database the names are in a single column, but with a comma and space separating them.

Our solution uses a histogram of all non-alphanumeric characters within the target column against all potential character posi-

Source			Target
first	last	...	full
robert	kerry	...	kerry, robert
kyle	norman	...	norman, kyle
norma	wiseman	...	wiseman, norma
...	...	...	...
amy	case	...	case, amy
josh	alder	...	alder, josh
john	galt	...	galt, john

Table 11: Requiring separators for variable-length regions.

tions. However, in order to be able to handle strings of variable length, we use relative positions allowing for as many positions as there are characters in the average length of the instances within the target column. For example, if the average instance length were 5, we would compute 5 relative positions, and if the current instance length were 10, we would retrieve the 4^{th} character when generating a histogram for relative position 2. (Note that this simplifies to absolute positions when a column is of fixed length.) For example, the histogram in Figure 4 plots the occurrence frequencies of potential separators in the full column for 700,000 instances similar to those shown in Table 11. Since the rounded average length for the column is 15 characters, we plot the histogram for relative positions 1 through 15.

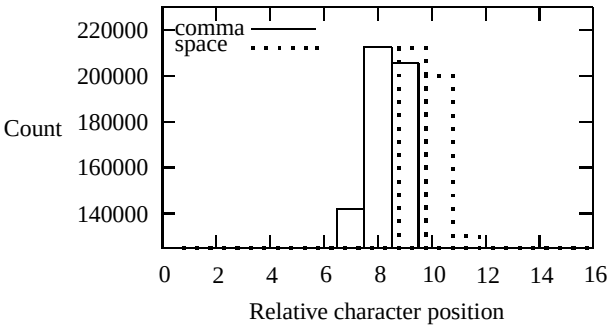

Figure 4: Histogram of possible separators and their locations within column full of Table 11.

From the histogram, we can see that there are many comma and space characters in the middle of the instances. We now need an algorithmic way to select which of these candidate separators and locations are actually valid for all column instances.

A candidate separator at some location is invalid if there is at least one instance that does not include it in that position. For a fixed column width, it would be sufficient to set a threshold to the number of instances within the column and simply select the characters and positions that score above it. However, for variable width columns, we must verify the separator template, as it is possible for artifacts of the data to generate an incorrect separator format. We therefore start by examining the most common separator/position pairs, and testing whether a template specifying those separators in those positions matches all the instances. If so, we augment the template to include the next most common separator-location pairs and continue until a candidate template no longer matches all instances.

Algorithm 8 encodes the building of the histograms followed by the search for the appropriate separator template by repeatedly lowering a threshold controlling which separator-location pairs to include. Using this algorithm, we are able to recover the separator recipe "%, %" for the data within the full column of Table 11. With the knowledge of this separator recipe and using the multi-column substring matching method described above, we recovered the translation formula used to create the column: last[1-n] + ", " + first[1-n].

Data: A target column A
Result: SearchKey: A representation of the separator pattern
SearchKey = '%';
AvgLength = Avg(Length(A));
Total = CountInstances(A);
for $j = 1$ **to** AvgLength **do**
  foreach Separator character s **do**
    foreach Instance a of A **do**
      if charAt(j/AvgLength*Length(a))==s **then**
        C_{sj}++;
      end
    end
  end
end
Threshold=Max(C_{sj});
TestSearchKey=SearchKey;
repeat
  SearchKey=TestSearchKey;
  for $j = 1$ **to** AvgLength **do**
    foreach Separator character s **do**
      if C_{sj} >Threshold **then**
        TestSearchKey = TestSearchKey + s;
      else
        TestSearchKey = TestSearchKey + '%';
      end
    end
  end
  Threshold−−;
until (CountInstances(A) like TestSearchKey) < Total;

Algorithm 8: Seperator indentification algorithm.

6.2 Dealing with many-to-many translations

Consider Table 12, where multiple target columns exist. It would be desirable for us to be able to identify both of the translations in use in this table while leveraging the fact that there are multiple concurrent translations in effect.

Source				Target	
birth day	first	middle	last	login	DOB
12-21-1923	robert	h	kerry	nawisema	5/6/73
11-13-1956	kyle	s	norman	jlmalton	8/11/48
5-6-1973	norma	a	wisema	rhkerry	12/21/23
...	...	...	...	...	...
1-3-1981	amy	l	case	alcase	1/3/81
5-29-1989	josh	a	alderman	ksokmoan	2/20/73
8-11-1948	john	l	malton	ksnorman	11/13/56

Table 12: A version of Table 1 with multiple targets.

The mechanism for choosing which target column to process first is beyond the scope of this work; we expect it to be chosen by another part of the database integration system. Our contribution to this problem assumes that one of the translations has already been identified and resolved, and we wish to use this knowledge in finding a subsequent translation.

In Section 3.3.1 we selected target instances based on their similarity to the sampled value, and in Section 3.4.1 we restricted the

retrieval further to instances which also fit the partial translation formula. In the many-to-many case, we already have a translation that relates rows of the source table to the target table. Therefore we can use that translation to restrict the selection of similar instances within rows to be those that are related by the known translation.

For example, let us assume that we have a translation for the column login that reads as first[1-1] + middle[1-1] + last[1-n] in Table 12. Let us also assume that we are trying to find a translation for the target column DOB and that we are retrieving similar values to the birth column instance '5-6-1973." If we trace the source column relation to the known translation for columns first, middle and last, we constrain possible target instances. For the example, starting with birth day = '5-6-1973," we find corresponding fields first = 'norma," middle = 'a," and last = 'wisema"; using the know translation formula, we obtain a value of 'nawisema" for target column login; from which we are constrained to using '5/6/73" for DOB. This is the direct algorithmic equivalent of having information about which tuples of T_1 match which tuples of T_2. By using this prior knowledge about the translations that link the tables, we are able here to dramatically reduce the number of instances to be evaluated and thus speed up the processing.

7. CONCLUSION

Whereas previous approaches required specialized domain specific matchers to form the matches and translations, we present here a generalized algorithm for most string-based matches. This method attempts to find a translation formula that composes a target column from the concatenation of an arbitrary number of column substrings. We do this without user training or explicit linkage between table rows, and experimental results validate the approach for realistic data.

Because the method matches complex column translations and because it is computationally expensive, it must function within a framework of a schema integration system. We make an explicit assumption that a certain overlap exists between both datasets and that the framework is able to provide us with both a potential target column and a set of candidate columns.

Although we found that in our examples, bi-grams and 10% sample sizes work well in practice, we are currently working on automating the selection of q and of sampling parameters that are used by the method. We also wish to develop a method to combine several applicable translation formulas into a single translation formula whenever this is appropriate. For example, it would be desirable to make use of optional values within translation rules to achieve greater coverage (e.g.: login = first[1-1] + middle[1-1] + last[1-n] would also encompass the rule login = first[1-1] + last[1-n]). We have not done so yet because of the algorithmic difficulty in searching for a negative result, but we plan to pursue rule-merging strategies [22] in our future work to achieve this. We showed how to identify separator data that is not present in the source columns, but we would like to expand this to the identification of other forms of missing information within the source table.

Acknowledgements

We gratefully acknowledge funding support from the Ontario Ministry of Training, Colleges, and Universities; the Natural Sciences and Engineering Research Council of Canada; and the University of Waterloo.

8. REFERENCES

[1] P. Carreira and H. Galhardas. Execution of data mappers. In *Intl. Workshop on Information Quality in Info. Sys.*, pages 2–9, 2004.

[2] S. Chaudhuri, K. Ganjam, V. Ganti, and R. M. ani. Robust and efficient fuzzy match for online data cleaning. In *Intl. Conf. ACM SIGMOD*, pages 313–324, 2003.

[3] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: discovering complex semantic matches between database schemas. In *Intl. Conf. ACM SIGMOD*, pages 383–394, 2004.

[4] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Intl. Conf. ACM SIGMOD*, page 509, 2001.

[5] D. W. Embley, L. Xu, and Y. Ding. Automatic direct and indirect schema mapping: experiences and lessons learned. *SIGMOD Rec.*, 33(4):14–19, 2004.

[6] G. H. L. Fletcher. The data mapping problem: Algorithmic and logical characterizations. In *Workshop on Databases For Next Generation Researchers at ICDE*, 2005.

[7] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *Intl. WWW Conference*, pages 90–101, 2003.

[8] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 18(6):341–343, 1975.

[9] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.

[10] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, pages 1078–1086, 2004.

[11] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady*, 10(8):707–710, Feb. 1966.

[12] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Intl. Conf. VLDB*, page 49, 2001.

[13] B. Momjian. *PostgreSQL: introduction and concepts*. Addison Wesley, 2001.

[14] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DMKD*, pages 0–, 1997.

[15] M. S. Paterson and V. Dancik. Longest common subsequences. In *Math. Foundations of Comp. Sci.*, pages 127–142, 1994.

[16] E. Rahm and P. Bernstein. On matching schemas automatically. Technical Report MSR-TR-2001-17, Microsoft Research, Feb. 2001.

[17] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[18] G. Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. *Comm. ACM*, 18(11):613, 1975.

[19] L. Seligman, A. Rosenthal, P. Lehner, and A. Smith. Data integration: Where does the time go?, Nov. 2005.

[20] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theor. Comp. Sci.*, 92(1):191–211, 1992.

[21] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Intl. Conf. ACM SIGMOD*, pages 485–496, 2001.

[22] M. D. Young-Lai and F. Tompa. Stochastic grammatical inference of text database structure. *Machine Learning*, 40:111–137, 2000.