Flexible String Matching Against Large Databases in Practice

Nick Koudas Amit Marathe Divesh Srivastava
AT&T Labs–Research
{koudas, marathe, divesh}@research.att.com

Abstract

Data Cleaning is an important process that has been at the center of research interest in recent years. Poor data quality is the result of a variety of reasons, including data entry errors and multiple conventions for recording database fields, and has a significant impact on a variety of business issues. Hence, there is a pressing need for technologies that enable flexible (fuzzy) matching of string information in a database. Cosine similarity with tf-idf is a well-established metric for comparing text, and recent proposals have adapted this similarity measure for flexibly matching a query string with values in a single attribute of a relation.

In deploying tf-idf based flexible string matching against real AT&T databases, we observed that this technique needed to be enhanced in many ways. First, along the *functionality* dimension, where there was a need to flexibly match along multiple string-valued attributes, and also take advantage of known semantic equivalences. Second, we identified various *performance enhancements* to speed up the matching process, potentially trading off a small degree of accuracy for substantial performance gains. In this paper, we report on our techniques and experience in dealing with flexible string matching against real AT&T databases.

1 Introduction

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases. Poor data quality is the result of a variety of reasons, including data entry errors (e.g., typing mistakes), poor integrity constraints and multiple conventions for recording database fields (e.g., company names, addresses). This has a significant impact on a variety of business issues, such as customer relationship management (e.g., inability to retrieve a customer record during a service call), billing errors and distribution delays. As a result, data cleaning has been at the center of research interest in recent years (see, e.g., [3]).

A key technology in data cleaning is flexible (fuzzy) matching of string information in a database. Such information is prevalent in corporate databases (e.g., customer names, company names, product names, addresses), and effectively matching such attribute values, taking into account the many sources of poor data quality, is a challenge. Consider, for example, the address of AT&T's headquarters in the US: "900 Route 202/206, Bedminster, NJ". Due to multiple conventions in representing such addresses, this address also occurs in various databases as "900 USHwy 202/206, Bedminster, NJ", "900 Rt 202, Bedminster, NJ". Similarly, when considering company names, it is common to see "Microsoft", "Microsoft Inc." and "Microsoft Corporation" being used in different records to represent the same entity. A simple equality or (even) substring comparison on names or addresses will not properly identify them as being the same entity, leading to a variety of potential business problems.

To effectively deal with flexible matching of string values in a database, while accounting for data quality issues, recent techniques [2, 4] have proposed the use of the well-established tf-idf (term frequency, inverse document frequency) metric, commonly used in Information Retrieval for comparing text. Intuitively, tokens (words, q-grams, etc.) are extracted from database strings, and each token is associated with a weight (idf) reflecting its commonality in the database (common tokens are assigned a low weight, uncommon tokens are assigned a high weight). Each database string is then associated with a (normalized) weight vector (incorporating both tf and idf) corresponding to the tokens extracted from it. Similarity between database strings, or between a database string and a query string, is then computed using the cosine similarity (inner product) of the corresponding weight vectors, essentially taking the weights of the common tokens into account.

In deploying such a technique against real AT&T databases, we observed that applications do not want to merely match string values in a single attribute.

- Often, there is a need to flexibly match along multiple string-valued attributes, for example, both company name and (partial) address. As can be expected, this helps to focus the search considerably. While

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

there might be many high-similarity flexible matches for both the company name (e.g., "Microsoft") and the partial address ("New York, NY"), individually, the combined query has much fewer high-similarity matches.

- Again, there are semantic relationships that are often known, which are unlikely to be matched using basic flexible string matching techniques. For example, AT&T's headquarters also has the (personalized) address "1 ATT Way, Bedminster, NJ", which is hard to match with (the standard address of) "900 Route 202/206, Bedminster, NJ". Similarly, "Worldcom Corp." and "MCI Inc." refer to the same company, but would not be matched using basic string matching techniques.

Such needs require that the basic string matching technique be enhanced along the *functionality* dimension. In addition, when such flexible string matching is done against large databases (with tens of millions of records), performance becomes a bottleneck, even when the technique is implemented, using SQL, inside the database. This requires the identification of novel *performance enhancements* to speed up the matching process. In talking to users of the tools that we built, we identified that it was acceptable to trade off a small degree of accuracy for substantial performance gains.

In this paper, we address these functionality and performance issues, and report on our experience in using flexible string matching techniques against real AT&T databases. The rest of this paper is structured as follows. In Section 2, we present a detailed description of tf-idf and cosine similarity, along with the SQL that serves as our baseline in this paper. We describe our various functionality enhancements in Section 3, and the performance enhancements in Section 4. In each section, we provide both the conceptual contributions and an experimental evaluation of the impact of these contributions. We identify additional challenges that we faced in practice, both along the functionality and performance dimensions, in Section 5, before concluding in Section 6.

2 Single Attribute TF-IDF Matching

In this section, we present a detailed description of tf-idf (term frequency, inverse document frequency) and cosine similarity for matching against the values in a single relational attribute, along with the SQL that serves as our baseline in this paper. Our description is based on the approach mentioned by Gravano et al. [4]. Our techniques can be adapted to use alternate approaches, such as the one proposed by Chaudhuri et al. [2], as well.

Let `Base` denote a base table with a string-valued attribute `sva` against which the 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 performed in two stages:

- At *pre-processing time*, the `Base` table is pre-processed, and tokens (words, q-grams, etc.) are extracted from each database string in `Base.sva`. A variety of auxiliary tables get created, to compute the idf's 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 similarity score. Essentially, this query computes the cosine similarity (inner product) 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.

2.1 Pre-processing Time: Steps

We now describe the SQL of the pre-processing in a step-by-step fashion. Assume that we have extracted the tokens from the string values in `Base.sva` and stored the result in the term frequency table `BaseTF(tid, token, tf)`, where `tid` refers to the record identifier in the `Base` table (and hence uniquely identifies the string in the `sva` attribute of that table), and `tf` is the number of occurrences of `token` in that string. Also, for simplicity, assume that the table `BaseSize(size)` contains a single one-attribute record containing a count of the number of records in `Base`. The next sequence of steps is as follows.

First, each token needs to be associated with a weight (idf) that reflects its commonality in the database; common tokens are assigned a low weight, uncommon tokens are assigned a high weight. This is computed into the `BaseIDF(token, idf)` table below.

```
insert into BaseIDF(token, idf)
  select T.token, LOG(S.size) -
    LOG(COUNT(T.tid))
  from BaseTF T, BaseSize S
  group by T.token
```

Once the `idf`'s have been computed, and the `tf`'s are known from the `BaseTF` table, the weight vector corresponding to a string can be easily computed by associating the product `tf*idf` with each token extracted from the string. However, this is an un-normalized weight vector. Before computing this vector, the second step computes this normalization term, for each `tid`, as the l_2-norm (length in the Euclidean space) of the un-normalized weight vector. This is computed into the `BaseLength(tid, len)` table below.

```
insert into BaseLength(tid, len)
  select T.tid,
    SQRT(SUM(I.idf*I.idf*T.tf*T.tf))
  from BaseTF T, BaseIDF I
```

```
where T.token = I.token
group by T.tid
```

In the third, and final, pre-processing step, the normalized weight vector, associated with each string, is computed into the `BaseWeights(tid, token, weight)` table below.

```
insert into BaseWeights(tid, token, weight)
  select T.tid, T.token, T.tf*I.idf/L.len
  from BaseTF T, BaseIDF I, BaseLength L
  where T.token = I.token
    and T.tid = L.tid
```

2.2 Query Time: Steps

At query time, given a query string in the `Search(sva)` table, the above sequence of steps are performed to compute the `SearchWeights(tid, token, weight)` table. Note that the `BaseIDF` table is used to obtain the idf's of the tokens extracted from the search string, to ensure that the data in the database table drives the weight vector associated with the search string.

Finally, our baseline query, for computing all matches (along with the scores) whose scores exceed a pre-specified similarity threshold `T`, is given below.

```
select S.tid, B.tid, SUM(S.weight*B.weight)
from SearchWeights S, BaseWeights B
where S.token = B.token
group by S.tid, B.tid
having SUM(S.weight*B.weight) > T
```

If, instead of being given a single search string to match against a database table, we would like to compute the join of two database tables based on a flexible string match of their columns, the above SQL code works (more or less) unchanged.[1]

2.3 Contributions of the Paper

In the rest of this paper, we describe how the above technique for effectively identifying flexible string matches was extended by us to satisfy the needs of applications against AT&T databases.

- In Section 3, we discuss functionality enhancements. In particular, the ability to flexibly match multiple string-valued attributes (eg., company name and address), and the ability to take advantage of known semantic relationships (e.g., multiple names for the same company, or multiple addresses for the same location).

- In Section 4, we discuss performance enhancements that are necessary when dealing with large databases (tens of millions of records) with string-valued attributes. Most of these result in a small loss of recall (i.e., some answers are not returned), for substantial performance gains. However, for the answers that are returned, their scores are computed accurately.

In each section, we provide both the conceptual contributions and an experimental evaluation of the impact of these contributions.

[1] The only change would be to use the strings in both tables to compute the idf's.

3 Functionality Enhancements

3.1 Multiple attributes

Consider a `Contacts` table containing the name and address for all companies. We can perform flexible string matching on each field individually. But what may be desired is a "combined" search which, given a name-address pair (N, A), returns all tuples from the table that are "close" to the search pair. The problem is to define metrics for the distance between a search pair (N, A) and a tuple pair (N_i, A_i).

These metrics should be efficient to implement and have the same robustness properties as the cosine similarity metric. We also want these metrics to be "data-driven" to the extent possible. In other words, the number of parameters that require user intervention to adjust should be kept to a minimum. The cosine similarity metric can be categorized as "data-driven" because it has a single parameter, the similarity threshold, that has to be varied to change the behavior of the match.

For the sake of illustration, the rest of the discussion is in terms of the `Contacts` table with name and address attributes. But is should be noted that our enhancements work with any table that has multiple string-valued attributes.

3.1.1 Attribute Concatenation

A straightforward approach is to concatenate the name and address attributes into a single string and perform flexible string matching on this concatenation. The disadvantage with this simple metric is that it ignores a lot of statistical information. For example, if "Corporation" is common within the name attribute but rare within the address attribute then all the tokens derived from "Corporation" are assigned a low weight in the combined name-address string. Hence, a search for an address containing "Corporate Drive" won't assign a particularly high score to the relevant tuples, even if the tokens derived from "Corporate" are uncommon among addresses. By concatenating the name and address strings we have lost useful data about the tokens which are common among names but not among addresses (or vice versa).

3.1.2 Using Static Weights

Another metric that comes to mind is combining the similarity scores from individual flexible matches on name and address. That is, if, after running two separate flexible searches, the name attribute value in a tuple has score p and the address attribute value in that tuple has score q then we

sim	name	address
0.75	WORLDCOM	600 s federal st chicago il
0.75	WORLDCOM	400 international pkwy dallas tx
0.75	WORLDCOM	300 renaissance ctr detroit mi
0.75	WORLDCOM	111 8th ave new york ny
0.75	WORLDCOM	165 boulevard se atlanta ga
0.75	WORLDCOM	165 boulevard ne atlanta ga
0.75	WORLDCOM	910 15th st denver co
0.75	WORLDCOM	401 fieldcrest dr greenburgh ny
0.75	WORLDCOM	1102 grand blvd kansas city mo
0.75	WORLDCOM	1102 grand blvd kansas city mo

Figure 1: Static weights: name = 0.75, address = 0.25

sim	name	address
0.5	WORLDCOM	600 s federal st chicago il
0.5	WORLDCOM	400 international pkwy dallas tx
0.5	WORLDCOM	300 renaissance ctr detroit mi
0.5	WORLDCOM	111 8th ave new york ny
0.5	WORLDCOM	165 boulevard se atlanta ga
0.5	WORLDCOM	165 boulevard ne atlanta ga
0.5	WORLDCOM	910 15th st denver co
0.5	WORLDCOM	401 fieldcrest dr greenburgh ny
0.5	WORLDCOM	1102 grand blvd kansas city mo
0.5	WORLDCOM	1102 grand blvd kansas city mo

Figure 2: Static weights: name = 0.50, address = 0.50

sim	name	address
0.568945894389856	TC PAYPHONES	w houston st mar
0.563179091332391	TC PAYPHONES	w 30th st manhat
0.559800387826103	PSI COLOCATE AMTRAK	w 33rd st manhat
0.559800387826103	LIRR	w 33rd st manhat
0.543052687383282	AMERICAN MUSEUM OF N	w 79th st manhat
0.522269095184392	AUDREY ZUCKNER	manhattan ny
0.522269095184392	TUMBLE INTERACTIVE M	manhattan ny
0.522269095184392	AMS TECHNOLOGIES	manhattan ny
0.522269095184392	EASTERN ELECTRONICS CORP	manhattan ny
0.493439926229466	SPRINT SPECTRUM	64th st manhatta

Figure 3: Static weights: name = 0.25, address = 0.75

sim	name	address	tid1	tid2
0.540166767574879	WORLDCO	110 wall st new york ny	1	90213
0.540166767574879	WORLDCO	110 wall st new york ny	1	90740
0.482300726524807	MANHATTAN CENTER FOR	e 116th manhattan ny	1	97312
0.447666521564224	WORLDCOM POP	750 e main st chattanooga tn	1	28302
0.445944986954701	MANHATTAN COMMUNICATION	963 manhattan ave brooklyn ny	1	96276
0.438408539297653	ALS CAGE AT MANHATTA	193 manhattan ave new york ny	1	87652

Figure 4: Attribute concatenation

define the combined score of that tuple to be $rp + (1 - r)q$ where r is a real number between 0 and 1. Such metrics have been well studied. It has the advantage of being easy to implement and by varying the value of r we can adjust the relative importance of the name and address attributes in the search. And while it preserves the different distributions of the name and address tokens it has the drawback that we a-priori have to fix the value of r and cannot change the weights assigned to the name and address scores in a dynamic manner. It is also not obvious how to infer a good value for r from the data.

3.1.3 Using Dynamic Weights

The metric we propose avoids these shortcomings by generalizing the normalization step performed during flexible matching. Recall that, in the 1-column flexible matching algorithm the raw tf-idf weights of all tokens in a tuple are divided by the l_2-norm of the weight vector to obtain normalized weights in the range [0, 1]. This normalization step also ensures that the similarity score of any tuple will be between 0 and 1.

In our metric, we run two flexible matches on the name and address attributes. But rather than normalize each weight vector separately, we normalize the disjoint union of the two vectors. Thus, the raw weight vectors from the name and address strings might be $X = (x_1, x_2, ..., x_k)$ and $Y = (y_1, y_2, ..., y_l)$. Let $L(X), L(Y)$ be the l_2-norms of these two vectors. Then, rather than dividing each weight in X by $L(X)$ and each weight in Y by $L(Y)$, we define $L(X, Y) = \sqrt{L(X)^2 + L(Y)^2}$ and divide all weights in X and Y by $L(X, Y)$.

Such normalization across attributes results in a dynamic adjustment in the relative importance of the attributes. For example, a search containing a common address like “100 Main St” will tend to give more importance to the name component. Conversely, a search on a common name will tend to place more emphasis on the address component.

3.1.4 Experiments

We now present an experiment comparing our dynamic weighting technique with static weighting. A table containing 100,000 rows of company names and addresses was used for this purpose. Both the name and address columns were indexed for flexible matching. We then ran a series of searches for the name-address pair (“Worldcom”, “Wall St Manhattan NY”) using static weights. The weights on the name and address columns took on the values (0.25, 0.75), (0.5, 0.5) and (0.75, 0.25). Finally, the same search was performed using attribute concatenation and with our technique.

Figures 1, 2 and 3 show the top results from static weight

sim	name	address
0.568322017929254	WORLDCO	110 wall st new york ny
0.568322017929254	WORLDCO	110 wall st new york ny
0.481778518568938	WORLDCOM	111 8th ave new york ny
0.462023103220271	WORLDCOM	60 hudson st new york ny
0.461736895695425	WORLDCOMM	140 west st new york ny
0.461223519217079	WORLDCOM POP	750 e main st chattanooga t
0.455999984346651	LIRR	w 33rd st manhattan ny
0.43316429737569	WORLDCOM	600 s federal st chicago il
0.425581077229952	TC PAYPHONES	w houston st manhattan ny
0.424860726905206	TC PAYPHONES	w 30th st manhattan ny

Figure 5: Dynamic weights

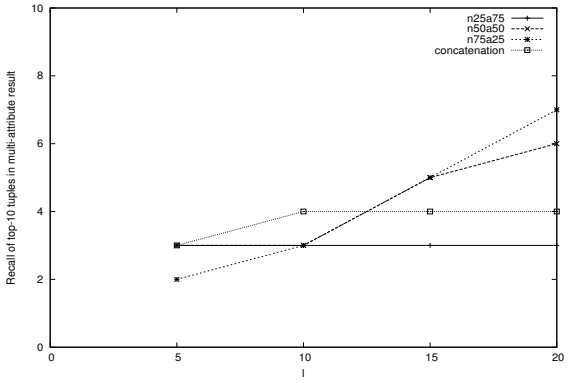

Figure 6: Ordered recall

searches. It is interesting to observe that the results from all the static weight searches are lopsided: figures 1 and 2 have exact matches on the name string but poor matches on the address string, while figure 3 has poor matches on the name component, but better matches on the address component. This is exactly the problem with static weights that we alluded to previously: it is difficult to choose a good distribution of weights among the attributes.

In contrast, the top result in the dynamic weights search ("Worldco", "110 Wall St New York NY"), shown in figure 5, is a very good overall match. This tuple is completely absent from the static weights searches because neither its match on name nor its match on address is high enough to place it at the top of any of those searches.

The attribute concatenation technique, shown in figure 4, does have the same top-2 matches as our dynamic weights technique. However, the latter matches are not as good: there are many tuples for which "Manhattan" appears in the name attribute. This illustrates the drawback inherent in the loss of information resulting from concatenating the two attributes.

The next experiment looks at top-k recall. Typically, we are interested in only the few top matches from a flexible match. Rather than considering the entire result set, we can restrict attention to the top-k matches from dynamic weighting and ask what fraction of those are found in the top-ℓ matches from competing techniques for various values of ℓ. Figure 6 presents the results of this experiment for k=10 and ℓ=5,10,15,20.

The results demonstrate that dynamic weighting is qualitatively different from static weighting or attribute concatenation and cannot be approximated by those techniques. As ℓ increases, some of the top-k matches from dynamic weighting are obtained using competing techniques. But even with $\ell = 20$, the recall numbers of competing techniques are quite low: only 40% for the attribute concatenation approach, and between 30% and 70% for the static weights approach. It is evident that the recall of static weighting is quite sensitive to the actual weights.

We have presented above a comparison of our multi-attribute dynamic weights matching with the attribute concatenation and the static weights techniques, on a specific query. Other queries exhibit a similar behavior in that static weighting tends to miss tuples that don't have a high match on at least one of the attributes. Thus the results of this section are illustrative of the ability of our multi-attribute matching technique to dynamically adjust the attribute weights and thereby return the most relevant tuples.

There may very well be domains in which dynamic weighting performs poorly as compared to other techniques. But the "data-driven" aspects of our multi-attribute matching lead to very desirable results, in all the domains we have encountered. An investigation into the quality of the metrics considered here is left for the future.

3.2 Semantic knowledge

The next enhancement involves incorporating domain specific semantic knowledge into the flexible string matching algorithm. It often happens that the same entity is represented in multiple ways inside the database. For example, the addresses "1 ATT Way Bedminster NJ" and "900 Route 202/206 Bedminster NJ" refer to the same location, AT&T's headquarters. Similarly, the same corporation may appear as "MCI" and as "Worldcom". Observe that the presence of more than one representation is not in itself an error: as these examples show, all the representations may be valid.

If the representations of an entity are sufficiently close in a textual sense then they can be captured using the cosine similarity metric. Thus, a flexible search for "IBM Corp" on a company names database will pick up not only exact matches but also alternate names like "IBM Corporation" and "IBM Inc". But some representations, like the address pair above, can be so far apart as to have few tokens in common. We would like our flexible matching to retrieve not only strings that are close to the search string but also their synonyms. Thus, a search for "900 Route 202 Bedminister NJ" should return "900 Route 202/**206** Bedminster NJ" and also "1 ATT Way Bedminster NJ".

Our proposed solution to this problem assumes that semantic equivalences are explicitly specified in a new relation. Conceptually, this is a symmetric two attribute relation $S(A, B)$. That is, for each equivalence $x = x'$, the tuples (x, x') and (x', x) would be in relation S. Let $T(P)$

	sim	address	tid1	tid2
Explain sim	0.752483612033064	4001 queens blvd queens ny	6	76542
Explain sim	0.701585125233138	4011 queens blvd queens ny	6	76364
Explain sim	0.692897932355297	queens ny	6	76627
Explain sim	0.692897932355297	queens ny	6	76630
Explain sim	0.692897932355297	queens ny	6	25573
Explain sim	0.692897932355297	queens ny	6	46546

Figure 7: Semantics: search on "4001 Rte 25 Forrest Hills NY"

be the one attribute relation on which we would like to enable this semantic-aware search.

3.2.1 Pre-processing Relation T

The first step involves pre-processing attribute values in relation T. This is done by computing the flexible string join of T and S, using attributes P and A respectively. For every result (p_i, a_i, b_i) in the join with a "high" similarity score, we augment the tokens associated with attribute value p_i in relation T with tokens derived from b_i. This has the effect of associating with each attribute value all the tokens corresponding to its synonyms as per relation S. In our company names example, the strings "MCI" and "Worldcom" will both be associated with the same set of tokens: those derived from the strings "MCI" and "Worldcom".

3.2.2 Processing at Query Time

In the next step, we carry out an analogous procedure on the search string q. The search string q is used in a flexible match operation on relation S. For all high scoring tuples (q, a_j, b_j) in the result, the set of tokens associated with the search string q is extended by the tokens derived from attribute value b_j.

The final step involves running the flexible match algorithm on the pre-processed relation T and the modified search string. Because we augment the set of tokens associated with both the search string and the attribute values in relation T with synonym information, this method is very robust in dealing with errors and multiple conventions in the string attributes of relation T and of synonym relation S.

3.2.3 Experiments

We now present the results from an experiment on using the above algorithm. We used a table of addresses containing 100,000 rows. A synonym table was populated by hand with a few sample equivalences. One of these tuples identified "Route 25 Forest Hills NY" as a synonym for "Queens Blvd Queen NY". The modified index was built on the addresses as described above with a threshold of 0.5. A relatively low threshold is required because the equivalences in the synonym table specified street aliases rather than complete address synonyms. In other words, none of the equivalences included a street number and hence a low threshold on the similarity score was needed when joining to the address table. The search string was "4001 Rte 25 Forrest Hills NY". This string was joined to the synonym table and a modified index was built using all tuples with a score of 0.6 or higher. The top results from the semantic match are shown in Figure 7.

We note that even in the presence of deliberate errors in both the synonym table (i.e., "Queen NY" instead of "Queens NY") and the search string (i.e., "Forrest Hills" instead of "Forest Hills", and "Rte 25" instead of "Route 25"), our algorithm was able to pick out the exact match and place it at the top of the results. This is a good illustration of the robustness of our technique.

4 Performance Enhancements

Recall that the basic query we run to find approximate matches above a certain similarity threshold T is

```
select S.tid, B.tid,
  sum(S.weight*B.weight)
from SearchWeights S, BaseWeights B
where S.token = B.token
and S.tid = N
group by S.tid, B.tid
having sum(S.weight * B.weight) > T
```

where N is the tuple id of the string we want to search on.

4.1 Indexing the Weights Table

The primary key on the BaseWeights table is (tid, token). In the absence of any other indices the above query has to scan through the BaseWeights table for each token in the search string. The obvious optimization that can be applied at this point is to build an index I1(token) on BaseWeights. Adding this index results in a "nested loops with indexing" execution plan for the above query. The performance improvement is shown below for base tables of different sizes.

Table size	Running time (sec)	
	NonIndexed	Indexed
100000	2	1
7000000	48	22
13000000	105	42

Searches run much faster with the index but, as the figure shows, they can still take a significant amount of time. The reason is that the SQL fragment above computes the dot product of the search vector with every tuple vector with which it shares a common token. For a base table with millions of rows that can be an expensive operation.

4.2 Pre-selecting High Weight Tuples

The next class of optimizations we consider all involve pre-selecting tuples from BaseWeights which are likely to be in the final result. This is done by adding another conjunctive condition to the where clause. This condition takes the following form

```
B.tid in (SubQuery)
```

where SubQuery selects a subset of tids from BaseWeights. Note that any optimization in this class has perfect precision: it may miss some tuples but it won't overestimate or underestimate the similarity score for any tuple. This is in contrast with [4] where the scores themselves are approximated by the performance enhancements. We now consider 4 optimizations in this class.

4.2.1 O1: High Weight Token

Each score in the final result is a sum of terms with each term being the product of the weight of a token in the search string and the weight of that token in the base table. We can conjecture that if this sum of terms exceeds the threshold T then at least one of the base weights exceeds a fixed fraction F of T. This is the basis for our first optimization which is defined by the following SubQuery.

```
select B.tid
from SearchWeights S, BaseWeights B
where S.tid = N
and B.token = S.token
and B.weight > T * F
```

Perusing the query above we observe that another index I2(token, weight) on the BaseWeights table is called for.

4.2.2 O2: High Weight Term

It may be the case that a token has low weight in the base table but high weight in the search string. The above optimization will miss such tuples. To compensate for this deficiency, we can change the last condition in O1's where clause to get the SubQuery below.

```
select B.tid
from SearchWeights S, BaseWeights B
where S.tid = N
and B.token = S.token
and B.weight > T * G / S.Weight
```

where G is a suitable fraction. Here, we cast a wider net in the SubQuery by also considering tokens which may have a low base weight, provided that the product of the search weight and the tuple weight is at least a fixed fraction of threshold T. Note that the index I2 we defined previously on BaseWeights also improves the execution plan for this SubQuery.

4.2.3 O3: Many High Weight Tokens

Both O1 and O2 pre-select tuples that have at least one "promising" token in common with the search string. To further narrow down this set of tuples we can pre-select only those tuples which have at least K (> 1) high weight tokens in common with the search string.

Optimization O3 is obtained by applying this heuristic to O1. The SubQuery in this case is as follows.

```
select B.tid
from SearchWeights S, BaseWeights B
where S.tid = N
and B.token = S.token
and B.weight > T * F
group by B.tid
having count(*) >= K
```

4.2.4 O4: Many High Weight Terms

The SubQuery obtained by applying the above heuristic to optimization O2 is given below.

```
select B.tid
from SearchWeights S, BaseWeights B
where S.tid = N
and B.token = S.token
and B.weight > T * G / S.Weight
group by B.tid
having count(*) >= K
```

4.3 Experiments

We now present some experiments comparing these optimizations. We used a company names table containing 13 million rows for the flexible matching. The similarity threshold was set to 0.4. Parameter F was varied from 0.2 to 0.8 (for optimizations O1 and O3), parameter G was varied from 0.05 to 0.20 (for optimizations O2 and O4) while parameter K was varied from 2 to 4 (for optimizations O3 and O4). These ranges were chosen to illustrate interesting tradeoffs in the various enhancements. In each experiment, we measured the running time and recall of each optimization relative to the naive query presented at the beginning of this section (with the I1 index on BaseWeights).

Figures 8, 9, 10 and 11 show the effect of parameters F, G and K on the recall and running time of these optimizations. We note that recall is inversely proportional to parameters F and G. Low values of these parameters lead to perfect (or near-perfect) recall. As we increase F and G, the number of tuples pre-selected by the SubQuery decreases because fewer tuples are likely to share a high weight token (optimizations O1 and O3) or a high weight term (optimizations O2 and O4) in common with the search string. Also, increasing K means that we insist on more and more high weight tokens or terms in common. Therefore, recall declines as K increases.

Execution time is positively correlated with recall for the same reasons. As we increase F, G and K the subset of tuples pre-selected by the SubQuery decreases in size.

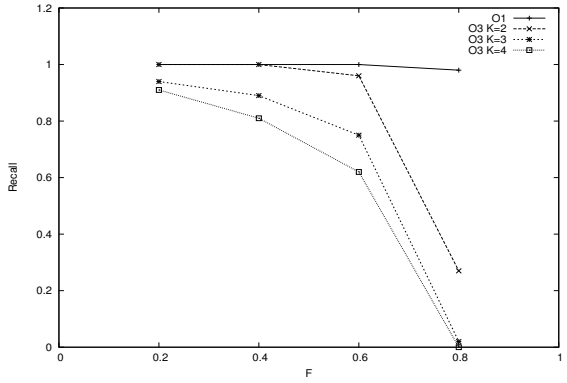

Figure 8: Recall for optimizations O1, O3

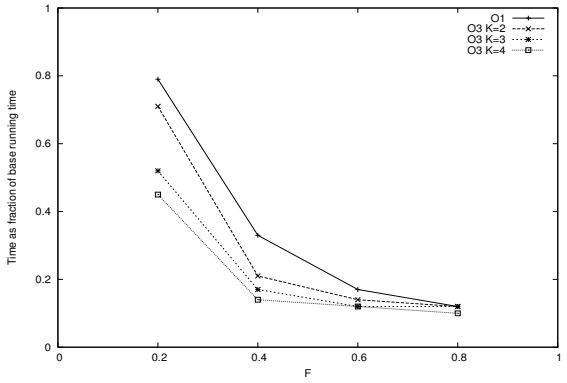

Figure 10: Execution times for optimizations O1, O3

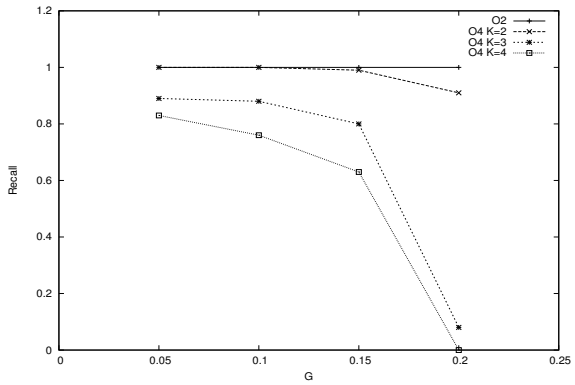

Figure 9: Recall for optimizations O2, O4

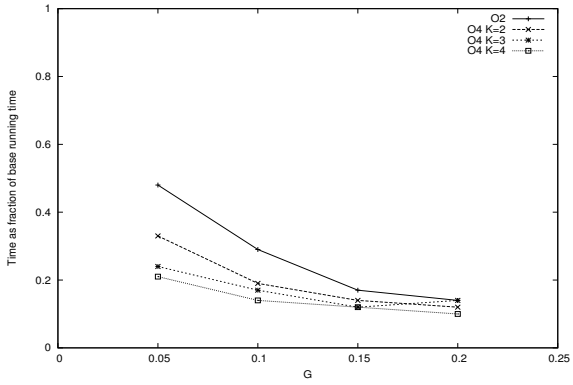

Figure 11: Execution times for optimizations O2, O4

Consequently, there are fewer tuples for which we have to compute the exact cosine similarity score and hence the overall query runs faster.

Since optimizations O3 and O4 have very good running times but low recall, the next two experiments try to understand which tuples from the base result set are absent from the results of these optimizations. We fixed the parameter values at F=0.6 and G=0.15, and measured the ordered recall for the top-50 results in the base result set. Figures 12 and 13 plot the number of these top-50 tuples found in the top-i result sets for optimizations O3 and O4 for i=50, 100, 150, 200.

From the figures we see that K = 4 leads to very bad recall. That is, optimizations O3 and O4 fail to return even 50% of the top-50 tuples with this value of K, even when the range is extended to the first 200 tuples. K = 3 is also not particularly good on recall. Therefore, for applications in which recall in the top tuples is important it is best to stick to lower values of parameters F and G at the cost of increased execution time.

It can be said that the parameters which control the SubQuery are somewhat arbitrary. An improvement that can be made in that regard is to replace F and G in all SubQueries with (F/L) and (G/L) respectively, where L is the length of the search string. The idea is that when the search string is long there are many tokens/terms which can contribute to the final score and we lower the bar a tuple must meet for pre-selection. This adjustment makes the choice of parameters more robust to a different data set.

5 Open Issues

5.1 Functionality

Multi-column flexible matching is important in many practical applications. Our proposed technique for this problem works on columns within a single table. In general, the columns on which we want to enable flexible matching will belong to different tables, with various join paths between them. Efficiently implementing flexible matching across tables (without having to materialize the join of the base tables beforehand) is a topic for future work.

Another open question is the handling of semantic dissimilarities, a.k.a. antonyms. We have come across this problem very frequently in the context of flexible address matching where the same city name may be present in multiple states, e.g., Manhattan KS $\neq$ Manhattan NY. In this setting, the algorithm has to somehow filter out the antonyms corresponding to the search string while still assigning a high score to all the synonyms.

An ad-hoc approach would be to create a separate antonym table and query this table before returning the results. Thus a search string of “Manhattan KS” would match

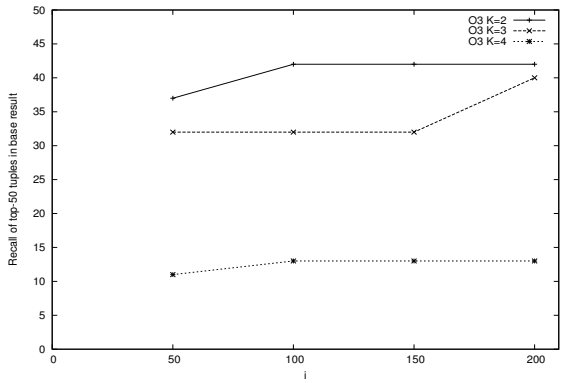

Figure 12: Ordered recall for optimization O3

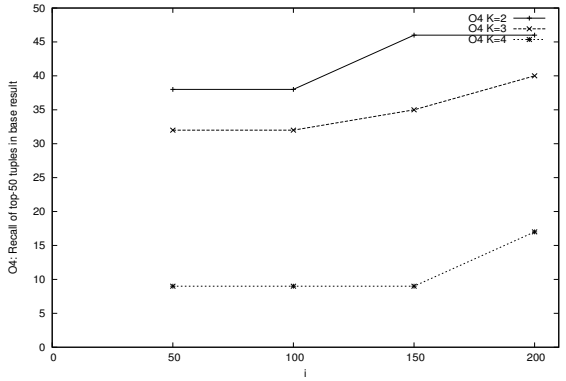

Figure 13: Ordered recall for optimization O4

"Manhattan NY" with a high score but the final step would consult the antonym table and drop that tuple. However, this method is very rigid when it comes to mistakes in the search string and/or the antonym table. Ideally, we would like antonyms to be processed in a robust manner, in much the same way that the cosine similarity metric captures errors in both the search string and the field values. An interesting research problem is the development of principled techniques (as opposed to ad-hoc ones) that can correctly and flexibly process such semantic negations.

So far, we have dealt only with string attributes. However, there are many data types that are commonly encountered in practice. Numeric data is of particular interest. For example, a table might contain an age field of type integer or latitude/longitude fields of type double. The conventional approach of finding fuzzy matches with respect to a given numeric value is to issue a range query against the corresponding field. But this does not take advantage of the data distribution to return a better result. Consider, for example, a database containing the latitude/longitude of all towns in the US. We might want a search on latitude/longitude to have small range in dense population areas like New Jersey and a large range in sparse population areas like Idaho.

Many search engines work with just the string representation of numeric values. This turns out to be inadequate for flexible matching purposes: a google search on "186000" turns up a few pages mentioning the speed of light but a search on "185900" does not find any such pages. Part of the reason is that the string representation of numbers which are very close may not have enough tokens in common. Alternatively, we could define a notion of tf/idf for numbers. The extension of the cosine similarity metric to non-string data types is an intriguing research direction [1].

5.2 Performance

Also of importance is the adaptation of our techniques to a dynamically updated database. So far, we have assumed that the data does not change (or if it does, we can quickly rebuild the flexible string match index). In practice, there are tables that are big enough that rebuilding the index on every change is not feasible. The key difficulty arises with "global" metrics such as tf-idf and cosine similarity. There the weight of a token depends on its inverse document frequency which in turn is a function of the fraction of tuples in which that token appears. Therefore, inserting a new tuple into the table, in principle, changes the weights of all tokens and thereby necessitates an index rebuild. In practice, the token weights will have changed by a non-zero but small amount. Since we are doing flexible matching it is acceptable to not insist on absolute accuracy. The challenge then is to identify criteria for the index rebuilds which work in practice by striking the right balance between accuracy and efficiency.

Index rebuilds can take a long time (a few hours for a table with a few million rows) during which time flexible matching cannot be performed against the table. So once we define a suitable criterion for rebuilding the index we also need to investigate ways to restructure the computation to avoid causing any downtime of the query functionality.

6 Conclusion

In this paper, we related our experiences in deploying flexible string matching on large databases within AT&T. We started with the cosine similarity metric and extended it to handle multi-attribute flexible matching. We enhanced the algorithm to use semantic equivalences that cannot be captured by textual means. We also suggested a number of optimizations that allow the results to be retrieved more quickly. These performance improvements preserve precision and enable a dramatic reduction in the running time while decreasing recall by only a small amount.

References

[1] R. Agrawal and R. Srikant. Searching with numbers. *Proceedings of WWW*, 2002.

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

[3] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.

[4] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. *Proceedings of WWW*, 2003.