Propagating Updates in SPIDER

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

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

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

ABSTRACT

SPIDER, developed at AT&T Labs–Research, is a system that efficiently supports flexible string matching against attribute values in large databases, and is extensively used in AT&T. The scoring methodology is based on $tf.idf$ weighting and cosine similarity, and SPIDER maintains indexes containing string tokens and their weights, for fast matching at query time. Given the “global” nature of the weights maintained in the indexes, even a few updates to the underlying database tables would necessitate a (near-)complete recomputation of the indexes, which can be prohibitively expensive. In this paper, we explore novel techniques to considerably reduce the cost of propagating updates in SPIDER, without a significant degradation of answer accuracy or query performance. We present experimental evidence using real data sets to demonstrate the practical benefits of our techniques.

1. INTRODUCTION

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases (see, e.g., [5, 1]). Data quality issues could instigate a variety of business problems, such as inefficient customer relationship management, billing errors and distribution delays. A very typical problem encountered in such settings is that customer information (individual names, corporation names, addresses) is represented differently across related databases. Similar problems exist with product names, product descriptions, etc. Information consists primarily of strings. As a result, there is a pressing need for technologies that enable flexible (fuzzy) matching of string information in a database [12].

At AT&T Labs–Research, we have been developing a prototype system called SPIDER to efficiently support flexible string matching (FSM) of attribute values in large databases [10, 11]. SPIDER is extensively used in AT&T, both as a key component of an operational portal for matching customer names and addresses, and for a variety of ad hoc data quality analyses. The matching and scoring methodology used by SPIDER is based on $tf.idf$ weighting and cosine similarity [4]. For this purpose, SPIDER maintains indexes over the database tables, containing string tokens and their $tf.idf$ weights, for fast matching at query time (see Section 3 for details of these indexes). Microsoft’s SQL Server also provides similar capabilities as SPIDER [3].

A key issue that all these tools and systems need to deal with is that real-world data is not static, and is continuously updated. To accurately and completely take into account even a few updates to the underlying database tables, SPIDER would need to do a (near-)complete recomputation of the indexes. This is because of the “global” nature of the $tf.idf$ weights maintained in the indexes, which are sensitive to (i) the total number of records in a table, and (ii) the distribution of occurrences of string tokens in the records of the table. Since database tables tend to be large (10s of millions of records), and updated quite frequently (daily), complete recomputation can be prohibitively expensive.

In this paper, we explore novel techniques to considerably reduce the cost of propagating updates in SPIDER, without a significant degradation of answer accuracy or query performance. In particular, we show that the cost of propagating updates can be ameliorated using two interesting design tradeoffs:

- First, there is a cost-accuracy tradeoff when propagating updates from the database table to the indexes needed for answering FSM queries. Essentially, by delaying the propagation of updates using “blocking” and “thresholding”, the cost of propagating updates can be considerably reduced (> 90%), without adversely impacting the accuracy (< 1%) of FSM query answers (both in terms of the answer set and their scores). See Section 5 for more details.

- Second, there is a query-update cost tradeoff, given the relative frequency of updates and FSM queries. Essentially, one can determine the “optimal stage” to which the update propagation need be performed, to optimize the weighted average total cost of updates and FSM queries. See Section 6 for more details.

We investigate the conceptual underpinnings of these two design tradeoffs, and study their interactions, using real data sets. We demonstrate their significant benefits using an experimental study.

2. RELATED WORK

Trends in data quality research and development have evolved in two main directions. The first one is based on the use of proprietary technology that requires exclusive access to the data. A variety of companies provide tools that operate on data to identify quality problems using a multitude of technologies. Such tools commonly extract data out of relational databases and apply proprietary algorithms.
The second direction makes use of declarative specifications of data quality tasks. Data quality algorithms and techniques operate on data directly in the relational database, and are often expressed as SQL statements (see, e.g., [2, 7, 6, 8, 10]). Such an approach has the advantage of not requiring extraction of the data outside the database which can be a serious concern in large enterprises. Moreover it can readily leverage query optimization and execution provided by the database. Our SPIDER tool falls in this category. At a very high level, SPIDER interfaces with any relational database, issues declarative statements to preprocess any information it requires and is capable of expressing operations necessary to enable flexible string matching in a declarative way.

In order to enable flexible matching on strings (customer names, product names, addresses, etc.), there is a need for principled techniques to quantify the closeness of corresponding relational attribute values. A variety of predicates exist for such a purpose, including edit distance and tf*idf cosine similarity. In applying these predicates to our application data, we have found that tf*idf cosine similarity is more versatile and useful than edit distance.

3. TF.IDF FLEXIBLE STRING MATCHING

In this section, we present a detailed description of tf*idf (term frequency, inverse document frequency) and cosine similarity measure used in SPIDER for flexible string matching (FSM) against the values in a single relational attribute, along with the SQL queries that define the indexes used for this computation. Our description is based on the approach mentioned in [8, 10].

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, 3-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 s with a (normalized) weight vector (incorporating both tf and idf) corresponding to the tokens extracted from s.

- **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.

We now describe the SQL used by the pre-processing time steps 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 of exposition, 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 table BaseIDF(token, idf) below.

```sql
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. But this is an un-normalized weight vector. Before computing this vector, the second step computes this normalization term, for each tid, as the L2-norm (length in the Euclidean space) of the un-normalized weight vector. This is computed into the table BaseLength(tid, len) below.

```sql
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 following table BaseWeights(tid, token, weight).

```sql
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
```

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 scores) whose scores exceed a pre-specified similarity threshold T, is given below.

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

4. PROPAGATING UPDATES

In this section, we first present an illustrative example to raise some of the issues involved in propagating updates in SPIDER. We then show a modified sequence of SQL statements that is amenable to a stage-wise incremental computation. Finally, we present our objectives that lead to the design tradeoffs investigated in subsequent sections.

4.1 Building Intuition: An Example Scenario

Consider a small table containing strings, as shown in Table 1. Assume that this table has been indexed to answer FSM queries. We would now like to allow updates on the table while still being able to answer FSM queries. Obviously, this involves updating the index tables for each update to the Base table.

There are 328 tokens (3-grams) in the above set of strings. Most of the tokens occur in just one tuple while only a handful of them
Table 1: Example table with company names.

<table>
<thead>
<tr>
<th>df</th>
<th># of tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>255</td>
</tr>
<tr>
<td>2</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Document frequency (df) of 3-gram tokens

appear in more than 4 tuples. The distribution of the token document frequency is shown in Table 2.

Now imagine that we wanted to change “American Intl Group” to “American International Group”. Then the df of tokens obtained from “International” will increase. Those tokens occur in 2 other tuples and hence the normalized weights of those tuples will also change. But since these tokens have a relatively high df (low idf) to start with such changes will ultimately have a low effect on the final normalized scores.

Alternatively, consider the name change from “Philip Morris” to “Altria”. Here, the tokens involved all have a low df (high idf). Hence, the number of tuples in the corpus affected by this change is small. However, those tuples that are affected are likely to change their normalized weights significantly.

4.2 Propagating Updates Incrementally

Looking at the SQL for computing the weights table in Section 3, we note that the idf for every token depends on the size of the Base table. Therefore, adding or deleting even a single tuple changes the entire BaseDF table. That causes all the tuples in BaseLength to change as well, which cascades into a change of all the tuples in BaseWeights. As we cannot perform a full re-index on every tuple addition/deletion, the original SQL is not amenable for incremental propagation of updates. To handle incremental updates we need to first isolate the effects of size changes and stop the cascading effect. We start by creating 3 new tables:

- BaseDF(token, df).
- BaseRawWeights(tid, token, tf, df).
- BaseRawLength(tid, s1, s2, s3).

BaseDF stores the document frequency corresponding to each token. BaseRawWeights is the join of BaseTF and BaseDF on the token column. To understand how BaseRawLength is populated, let L be the length of some tuple. Then,

\[ L = \sum_{i} t_{f_i}^2 \cdot \text{idf}_i^2 \]
\[ = \sum_{i} t_{f_i}^2 \cdot (\log(size) - \log(df_i))^2 \]
\[ = \log^2(size) \sum_{i} t_{f_i}^2 \]
\[ - 2\log(size) \sum_{i} t_{f_i}^2 \cdot \log(df_i) \]
\[ + \sum_{i} t_{f_i}^2 \cdot \log^2(df_i) \]

where size is the number of tuples in the Base table and the summation is over all the tokens in that particular tuple. Thus, we can express L as \[ \log^2(size)s1 - 2\log(size)s2 + s3 \] where s1, s2 and s3 depend only on the tokens in a tuple and are independent of size. The complete SQL for populating the new set of tables is given below.

\[
\text{insert into BaseDF(token, df)}
\text{select T.token, LOG(COUNT(T.tid))}
\text{from BaseTF T}
\text{group by T.token}
\]

\[
\text{insert into BaseRawWeights(tid, token, tf, df)}
\text{select T.tid, T.token, T.tf, D.df}
\text{from BaseTF T, BaseDF D}
\text{where T.token = D.token}
\]

\[
\text{insert into BaseRawLength(tid, s1, s2, s3)}
\text{select T.tid, SUM(T.tf*T.tf),}
\text{SUM(T.tf*T.tf*D.df), SUM(T.tf*T.tf*D.df*D.df)}
\text{from BaseTF T, BaseDF D}
\text{where D.token = T.token}
\text{group by T.tid}
\]

\[
\text{insert into BaseLength(tid, length)}
\text{select RL.tid,}
\text{sqrt(S.size*S.size*RL.s1 - 2*S.size*RL.s2 + RL.s3)}
\text{from BaseRawLength RL, BaseSize S}
\]

\[
\text{insert into BaseWeights(tid, token, weight)}
\text{select RW.tid, RW.token,}
\text{RW.tf*(S.size-RW.df)/L.length}
\text{from BaseRawWeights RW, BaseSize S, BaseLength L}
\text{where RW.tid=L.tid}
\]

The new set of tables can be arranged in 4 stages, as shown in Table 3, with tables in a stage computed from tables in earlier stages.
We will make use of these stages in Section 6, in discussing the query-update cost tradeoff.

4.3 Objectives

We have a table containing some string valued attributes on which we want to run FSM queries. Unlike previous work, the underlying table in this case is not static but subject to a stream of updates. Each update can add a new tuple or delete/modify an existing tuple. The goal is to answer FSM queries on the table while concurrently processing the updates efficiently. In answering the FSM queries, we want to study two design tradeoffs.

- First, balance the cost of processing the updates against the accuracy of the answers returned.
- In most applications, once the answer accuracy is sufficiently high there would be little value in spending more and more cost in update processing for smaller and smaller improvements in answer accuracy.
- Second, balance the update cost against the query cost.
- If updates are infrequent, then we would like to use a query plan in which individual query cost is minimized. Conversely, if queries are infrequent, then we should minimize the cost of individual updates.

We investigate these two tradeoffs in the next two sections.

5. DELAYED PROPAGATION

5.1 Method

We observed in the previous section that a change in the size of the Base table ends up changing all the tuples in the BaseWeights table. Similarly, a change in the document frequency(df) of a single token can propagate to many tuples in BaseWeights. For example, a change in the df of a token appearing in 10% of the tuples leads to a change in 10% of the tuples in BaseWeights. Clearly, it is not enough to restructure the index tables to isolate the dependency on BaseSize: it is also important to contain the effects of updates to the BaseDF table.

We can do that by using blocking and/or thresholding. In the blocking technique, updates are not processed as they arrive. Instead, updates are held in a buffer and only after we have collected a sufficient number of updates are all of them processed together as a batch. The rationale for this approach is that an individual update is going to change the final weights by a very small amount and hence batching the updates will result in more efficient processing (at the cost of small errors in accuracy).

In the thresholding technique, we consume updates on arrival and change the BaseSize, BaseTF and BaseDF tables. But rather than pushing all the changes to token dfs to later stages in the pipeline, we compare the new df to the old df. Only those tokens whose df has changed by a “significant” amount are considered when updating the downstream tables in stages 2, 3 and 4. The benefit is reduced update processing time, again at the cost of small errors in accuracy of FSM queries.

We quantify the effects of these two techniques in the experimental section.

5.2 Accuracy Issues

In this section we provide a theoretical justification for the blocking and thresholding techniques. To understand why those techniques work, consider how an error in the document frequency of a token might affect the final similarity score.

The idf of a token is defined as $\log(N/df)$ where N is the number of tuples in the Base table and df is the document frequency of that token. A small relative change in the df will lead to a small relative change in the idf, provided that the df is not “too large” compared to N. e.g. if N = 100 and df = 10 then a 10% change in the df results in approximately a 10% change in N/df and therefore a bounded percentage change in the idf. On the other hand, if N = 100 and df = 90 then a 10% change in the df can lead to a 100% change in N/df.

Thus, if a particular tuple does not contain any high df tokens and we don’t propagate df changes below a certain relative threshold then the normalized weights (and hence the similarity scores) for that tuple won’t differ much from the exact values. Similarly, if a tuple contains a low df token then that token is going to have a high idf and will influence the length of that tuple much more than the other tokens. If each token df undergoes a small relative change the length of such a tuple won’t change much and consequently the normalized weights will be almost unchanged.

Since the normalized query weights are computed at run-time using the latest BaseDF table they are always accurate. Thus the only problematic case is one in which the tuple and the query consist only of high df tokens. In this case, the approximate similarity score may differ a lot from the exact similarity score that would have been obtained if we had not thresholded the df changes. For example, consider a tuple consisting of tokens $t_1$, $t_2$ and a query consisting of tokens $t_3$ and $t_4$. Assume that all the three tokens are equally frequent in the corpus. Then the normalized weight vector for the tuple will be $(0.7, 0.7, 0)$ while that for the query will be $(0.7, 0.7, 0.7)$. Suppose that after a batch update token $t_2$ becomes less frequent and it’s normalized weight in the tuple and query increases to 0.99. The true similarity score between the tuple and query is 0.99 so 1.0. However, if we threshold the df changes then the approximate similarity score is 0.99 * 0.7 ≈ 0.7, a 30% difference from the correct value.

The reason we can use the delayed propagation techniques despite the above counterexample is that tuples which consist solely of high df tokens are exceedingly rare. The tables in Table 4 illustrate that rarity by answering, for different real data sets and different values of x, the following question: “what fraction of tuples contain only tokens that appear in at least x% of all the tuples?”

6. BOUNDED PROPAGATION

In addition to delayed propagation, we also have a choice of how far (up to what stage) to propagate an update, based on the assignment of tables to stages in Table 3. The advantage of thinking in terms of stages is that it immediately illustrates one way of decreasing the update cost: by limiting how far updates are propagated.

6.1 Cost Characteristics

The motivation for bounded propagation is the fact that updates

<table>
<thead>
<tr>
<th>Stage</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BaseTF</td>
</tr>
<tr>
<td>2</td>
<td>BaseDF</td>
</tr>
<tr>
<td>3</td>
<td>BaseSize</td>
</tr>
<tr>
<td>4</td>
<td>BaseRawLength</td>
</tr>
</tbody>
</table>

Table 3: Partitioning FSM tables into stages.
at various stages in the pipeline have different characteristics and costs.

- **record-level updates**: a single record change in stage $i$ induces a single record change in stage $i + 1$. For example, a change in the $tf$ of a particular token in a particular tuple affects the single tuple in the $BaseRawWeights$ which corresponds to this tuple-tid combination.

- **group-level updates**: a single record change in stage $i$ can affect a group of records (that share some group-id) in stage $i + 1$. For example, a change to a tuple in the Base table affects all the tuples in the $BaseSize$ table with the same tuple-id.

- **holistic updates**: a single record change in stage $i$ could affect the entire table at stage $i + 1$. For example, a change to the $BaseSize$ table changes all tuples in $BaseWeights$.

### 6.2 Stage-Wise FSM Query Answering

Propagating updates to different stages will work only if we can modify our original FSM query to work off the tables in any stage at which we decide to stop the propagation. In principle, this should always be possible because of the way the table dependencies are structured. The FSM query which uses the $BaseWeights$ table (in stage 4) was shown previously. We now give the SQL that can be used if FSM queries are to be run against stage 2 or stage 3.

- The FSM query to be run against stage 3 is:

  ```sql
  select S.tid, B.tid,
  sum(S.weight*BRW.tf*(BS.size-BRW.df)/BL.length) as sim
  from SearchWeights S, BaseRawWeights BRW,
  BaseLength BL, BaseSize BS
  where S.token = B.token
  and BRW.tid = BL.tid
  group by S.tid, B.tid
  having sim > T
  ```

- The FSM query to be run against stage 2 is:

  ```sql
  select S.tid, B.tid,
  sum(S.weight*BRW.tf*(BS.size-BRW.df)/sqrt((BRL.s1 * BS.s - 2*BRL.s2) * BS.s + BRL.s3)) as sim
  from SearchWeights S, BaseRawWeights BRW,
  BaseRawLength BRL, BaseSize BS
  where S.token = B.token
  and BRW.tid = BRL.tid
  group by S.tid, B.tid
  having sim > T
  ```

Given the varying query and update costs and depending on the query-update ratio, it may make sense to not push updates all the way to the last stage. If updates are much more frequent than queries it could be optimal to push updates only up to the second stage. In this case, we would be trading a decrease in the per-update cost for a smaller increase in the per-query cost. Conversely, if updates are very rare as compared to queries the most efficient plan would be to propagate the infrequent updates all the way to stage 4 and use the tables in the last stage to answer FSM queries.

We show in the experimental section that this approach is not just feasible but also practical.

### 7. EXPERIMENTS

In this section, we present the results of some experiments to evaluate the effectiveness of our proposed techniques. We used the dataset of DBLP authors in a SQL Server 2000 database running on a Dell PE2600 with two Intel Xeon processors and 4GB of memory.

#### 7.1 Performance Gains

The DBLP authors dataset consists of approximately 400K names with an average length of 13 characters. This dataset is easily available and hence other researchers can replicate our experiments and compare the results. Unfortunately, we don’t know the characteristics of the update stream for the set of DBLP authors. Therefore, we decided to create a synthetic stream of updates on which to base our experiments.

We would also like to study how our techniques work on datasets of different sizes. Hence, we created two other datasets from the DBLP author data containing 50% and 25% of the original records (chosen at random). For the rest of this section, we will use the names DBLP, DBLP-50 and DBLP-25 to refer to these datasets. We ran all our experiments on each of these 3 datasets to study how the techniques scale with increasing size of the $Base$ table.

Our first experiment measures the cost of running FSM queries against stages 2, 3 and 4. For each dataset we measure how long an average query takes when working off the tables in a particular stage. Figure 1 shows the results. For all datasets, queries against stage 2 run the slowest and for any particular stage the query time increases with the size of the dataset. Also, the timing difference between stages 2 and 3 is smaller than the difference between stages 3 and 4.

The second experiment evaluates the effectiveness of the blocking strategies on the update processing time. For batch sizes of 1, 10 and 100 and for thresholds of 0.001%, 0.01% and 0.1% we calculate the time taken to process a batch of updates at that threshold. For each (batch size, threshold) pair we further measure the time taken to push the updates only up to a particular stage in the pipeline.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>2.3</td>
</tr>
<tr>
<td>DBLP-50</td>
<td>3.2</td>
</tr>
<tr>
<td>DBLP-25</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Figure 1: Query timings

### Table 4: Rarity in real datasets.

<table>
<thead>
<tr>
<th>Upper bound</th>
<th>Count of records</th>
<th>% of tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>10</td>
<td>0.00</td>
</tr>
<tr>
<td>0.01</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>0.23</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Upper bound</th>
<th>Count of records</th>
<th>% of tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>10</td>
<td>0.00</td>
</tr>
<tr>
<td>0.01</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>1.23</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>
The results are shown in figures 2, 3 and 4. Some trends can be clearly seen: for a given batch size and threshold the update time increases as we push the updates further downstream. However, the increase from stage 2 to stage 3 is much smaller than that from stage 3 to stage 4. This makes intuitive sense as the update at the last stage tends to be a holistic update (a single change in the token $df$ is not necessarily restricted to a set of records in the BaseWeights table with the same group-id).

The second trend is that if the batch size and stage are fixed then processing time increases as the threshold decreases. That is consistent with our understanding that as the threshold is decreased more and more token $df$ changes are pushed forward from stage 2 to stages 3 and 4. In the limiting case at a very small relative threshold a change to even a handful of tuples typically ends up changing a large fraction of the BaseWeights table.

The final observation is that the update time increases with batch size when the threshold and stage are kept constant. Moreover, the increase is more pronounced for the second increase in the batch size (from 10 to 100). This can be explained by considering the tokens that are likely to be part of a batch. As the batch size increases, the probability that the batch contains a high $df$ token tends to 1 and therefore the update time also increases dramatically.

We also ran the same set of experiments for the DBLP-50 and DBLP-25 datasets. For the sake of brevity, the results are not shown here but they are similar to those obtained for DBLP authors.

### 7.2 Accuracy Loss

The previous section gave us an idea of the performance gains due to delayed and bounded propagation of updates to different stages of the FSM index tables.

We now investigate issues related to the loss in accuracy due to delayed propagation of updates. For a batch size of 100 and various values of the threshold, we ran a set of FSM queries against the approximate index and against the exact index (the index tables that would have been obtained with a threshold of 0). We compared the number of answers returned in the two cases and also the maximum score difference between corresponding answers. Finally, we compared the approximate and exact weight tables and computed the maximum difference in the normalized weights between corresponding tuples of the BaseWeights table. The results are shown in Table 5.

We can see that even with a threshold of 0.1% we get a high level of precision and recall. Moreover, the maximum weight difference in the approximate and normalized weights is very small in all cases which is a good indicator that the high precision/recall are not dependent on the particular queries chosen for this experiment.

### 8. CONCLUSION

We described how updates to database tables can be efficiently propagated to the index tables used for flexible string matching in the SPIDER system, developed at AT&T Labs–Research, without significant degradation of answer accuracy or query performance. The key reason this problem was challenging was because of the global nature of the $tf.idf$ weights maintained in the SPIDER indexes, which would require a (near-)complete recomputation of the indexes, to accurately and completely take into account even a few updates to the underlying database tables.

Our update propagation techniques are broadly applicable to any system or tool that uses $tf.idf$ weighting, and needs to efficiently manage a stream of updates, while tolerating a small loss in the accuracy of flexible string matching.

### 9. REFERENCES


<table>
<thead>
<tr>
<th>Threshold</th>
<th>Max Wt Diff</th>
<th>Approx Count</th>
<th>Exact Count</th>
<th>Common Count</th>
<th>Max Sim Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.007</td>
<td>746</td>
<td>744</td>
<td>743</td>
<td>0.004</td>
</tr>
<tr>
<td>0.01</td>
<td>0.001</td>
<td>764</td>
<td>764</td>
<td>764</td>
<td>0.0006</td>
</tr>
<tr>
<td>0.001</td>
<td>0.0003</td>
<td>1282</td>
<td>1282</td>
<td>1282</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

Table 5: Accuracy tradeoffs


