Flexible String Matching Against Large Databases in Practice

Divesh Srivastava
AT&T Labs–Research
http://www.research.att.com/~divesh/

Joint work with Nick Koudas and Amit Marathe
Thesis

- **String data prevalent in databases**
  - customer names, addresses, circuit ids, …
  - need to search, correlate tables based on string data
- **But string data often of poor quality**
  - multiple conventions, errors, embedded comments
  - equality, prefix, substring matching of limited use
- **Solution: Use “flexible” string matching**
  - string edit distance of limited use
  - cosine similarity with tf.idf of great practical use
Outline of Talk

- Thesis
- Background: matching on one string-valued attribute
- Functionality need: match on \( n \) attributes, use equivalences
- Performance need: enhance efficiency, trade off recall
Background: Need for Flexible String Matching

- Poor quality string data
  - “MICROSOFT INC.” vs “MICROSOFT CORPORATION”
  - “11810 WILLS ROAD ALPHARETTA GA30004” vs “BLDG 110 FLR 1 RM 110 11810 WILLIS RD ALPHARETTA GA 30004205”
- Equality, substring, string edit distance matchings of limited use
Background: Single Attribute TF.IDF Matching

- **Cosine similarity with (token freq, inverse DB freq)**
  - with string $s$, associate a weight vector $v_s$
  - $v_s[i]$ is tf.idf weight of token $t_i$ in string $s$
  - similarity between strings $s_1$ and $s_2$, $\text{sim}(s_1, s_2) = \sum_i (v_{s_1}[i] \times v_{s_2}[i])$

- **Correlating two database tables using one string attribute**
  - find all pairs with similarity $\geq$ threshold
  - conceptually: thresholded matrix multiplication
  - practically: use SQL for flexibility, efficiency
Background: SQL for Single Attribute Matching

- **Pre-processing:** extract tokens, compute tf.idf weights, normalize

  \[
  \text{Base}(\text{tid}, \text{sva}, \ldots) \\
  \text{BaseSize}(\text{size}) \\
  \text{BaseTF}(\text{tid}, \text{token}, \text{tf}) \\
  \text{insert into BaseIDF(token, idf)} \\
  \text{select T.token, LOG(S.size) - LOG(COUNT(T.tid)) from BaseTF T, BaseSize S} \\
  \text{group by T.token} \\
  \text{insert into BaseLength(tid, len) from BaseTF T, BaseIDF I} \\
  \text{where T.token = I.token and T.tid = L.tid} \\
  \text{select T.tid, SQRT(SUM(I.idf * I.idf * T.tf * T.tf)) from BaseTF T, BaseIDF I} \\
  \text{where T.token = I.token and T.tid = L.tid} \\
  \text{group by T.tid}
  \]

- **Query time:** above steps for query string, correlate & aggregate

  \[
  \text{Search}(\text{tid}, \text{sva}, \ldots) \\
  \text{SearchTF}(\text{tid}, \text{token}, \text{tf}) \\
  \text{SearchLength(tid, len)} \\
  \text{SearchWeights(tid, token, weight) from SearchWeights S, BaseWeights B} \\
  \text{where S.token = B.token} \\
  \text{group by S.tid, B.tid} \\
  \text{having SUM(S.weight * B.weight) > T}
  \]

Divesh Srivastava

Flexible String Matching
Functionality Need: Multiple Attribute Matching

- Need to match simultaneously on name and address
  - attribute concatenation: loses per-attribute statistical information
  - static weights: how to choose relative weights?
- Solution: dynamic weights for per-tuple attribute importance
  - normalize weight vectors jointly, using \( L(n, a) = \sqrt{L(n)^2 + L(a)^2} \)
  - run flexible matches on disjoint union of name, address
  - similarity scores are still between 0 and 1
Multiple Attribute Matching: Experiments

- Query: ("Worldcom", "Wall St Manhattan NY")
- Results: dynamic weighting has better quality answers
Functionality Need: Semantic Equivalences

- **Problem:** same real-world entity has distinct representations
  - “1 ATT Way Bedminster NJ” vs “900 Route 202/206 Bedminster NJ”
  - ‘MCI” vs “Worldcom”
  - since few tokens shared, not caught by flexible string matching

- **Solution:** enhance flexible string matching
  - specify semantic equivalences in a binary table $S$
  - additionally perform flexible string joins with $S$
  - pre-processing: join with $S$, augment database string tokens
  - query time: augment query string tokens, do flexible string matching
Semantic Equivalences: Experiments

- Synonym table with address equivalences
  - “Route 25 Forest Hills NY” vs “Queens Blvd Queen NY”
- Query: “4001 Rte 25 Forrest Hills NY”
- Results: robust despite poor data quality
  - in synonym table: “Queen NY” vs “Queens NY”
  - in query: “Forrest” vs “Forest”, “Rte 25” vs “Route 25”

<table>
<thead>
<tr>
<th>syn</th>
<th>address</th>
<th>cost</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>0.732493522536904</td>
<td>6</td>
<td>7664</td>
</tr>
<tr>
<td>Query</td>
<td>0.781395125202309</td>
<td>6</td>
<td>7664</td>
</tr>
<tr>
<td>English</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7662</td>
</tr>
<tr>
<td>Query</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7662</td>
</tr>
<tr>
<td>English</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7660</td>
</tr>
<tr>
<td>Query</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7660</td>
</tr>
<tr>
<td>English</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7660</td>
</tr>
<tr>
<td>Query</td>
<td>0.832897332298207</td>
<td>6</td>
<td>7660</td>
</tr>
</tbody>
</table>
Performance Enhancements: Indexing BaseWeights

- Query to find flexible matches above threshold $T$

  $\text{BaseWeights}(\text{tid, token, weight})$
  $\text{SearchWeights}(\text{tid, token, weight})$

  \[
  \text{select } \text{S.tid, B.tid, SUM(S.weight*B.weight)} \\
  \text{from } \text{SearchWeights S, BaseWeights B} \\
  \text{where S.token = B.token} \\
  \text{group by S.tid, B.tid} \\
  \text{having SUM(S.weight*B.weight) > T}
  \]

- Primary key of BaseWeights is (tid, token), join is by token

<table>
<thead>
<tr>
<th>Table size</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NonIndexed</td>
</tr>
<tr>
<td>100,000</td>
<td>2</td>
</tr>
<tr>
<td>7,000,000</td>
<td>48</td>
</tr>
<tr>
<td>13,000,000</td>
<td>105</td>
</tr>
</tbody>
</table>

Divesh Srivastava

Flexible String Matching
Performance Enhancements: Pre-selecting Tuples

- Even with indexing, flexible string matching takes time
  - 42 sec on 13,000,000 row base table
  - Cosine similarity computed with every tuple having a shared token
- Pre-selecting high weight tuples from BaseWeights
  - Add condition “B.tid in (SubQuery)” to where clause
  - Recall may suffer
  - Preserves similarity scores and perfect precision
Performance Enhancements: Pre-selecting Tuples

- Intuition for class of optimizations
  - cosine similarity is sum of terms, each term is product of token weights
  - pick one or many, high weight token or high weight term

01: High Weight Token
select B.tid
from SearchWeights S, BaseWeights B
where B.token = S.token and B.weight > T*F

02: High Weight Term
select B.tid
from SearchWeights S, BaseWeights B
where B.token = S.token and B.weight > T*G/S.Weight

03: Many High Weight Tokens
select B.tid
from SearchWeights S, BaseWeights B
where B.token = S.token and B.weight > T*F
group by B.tid
having count(*) >= K

04: Many High Weight Terms
select B.tid
from SearchWeights S, BaseWeights B
where B.token = S.token and B.weight > T*G/S.Weight
group by B.tid
having count(*) >= K
Pre-selecting Tuples: Experiments

- **Data**: company names table with 13 million rows
- **Results**: much smaller running time with small loss in recall

![Graphs showing Recall and Time as fraction of base running time for different K values (K=2, K=3, K=4) for both F and G datasets.](image)

Divesh Srivastava

Flexible String Matching
Open Issues

• **Functionality: multi-table, multi-attribute flexible matching**
  ○ issue: can one avoid materializing the multi-table join?

• **Functionality: handling semantic dissimilarities**
  ○ issue: principled techniques for semantic negations

• **Functionality: numeric data embedded in strings**
  ○ issue: need to handle string context and numeric order

• **Performance: dynamically updated databases**
  ○ issue: balance tf.idf accuracy with efficiency
Conclusions

• Technical contributions of paper
  ◦ dynamic weighting for tf.idf on multiple attributes
  ◦ extending tf.idf to incorporate semantic equivalences
  ◦ optimizations that trade off recall for efficiency

• Techniques implemented in Spider system, used at AT&T

• Ongoing work: functionality and performance enhancements