Benchmarking Declarative Approximate Selection Predicates

Amit Chandel, Oktie Hassanzadeh,
Nick Koudas, Mohammad Sadoghi
University of Toronto
{amit,oktie,koudas,mo}@cs.toronto.edu

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

ABSTRACT

Declarative data quality has been an active research topic. The fundamental principle behind a declarative approach to data quality is the use of declarative statements to realize data quality primitives on top of any relational data source. A primary advantage of such an approach is the ease of use and integration with existing applications. Over the last few years several similarity predicates have been proposed for common quality primitives (approximate selections, joins, etc) and have been fully expressed using declarative SQL statements. In this paper we propose new similarity predicates along with their declarative realization, based on notions of probabilistic information retrieval. In particular we show how language models and hidden Markov models can be utilized as similarity predicates for data quality and present their full declarative instantiation. We also show how other scoring methods from information retrieval, can be utilized in a similar setting. We then present full declarative specifications of previously proposed similarity predicates in the literature, grouping them into classes according to their primary characteristics. Finally, we present a thorough performance and accuracy study comparing a large number of similarity predicates for data cleaning operations. We quantify both their runtime performance as well as their accuracy for several types of common quality problems encountered in operational databases.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance

Keywords

Declarative data quality, data cleaning, SQL, performance, accuracy

1. INTRODUCTION

The importance of data cleaning and data quality technologies for business practices is well recognized. Data cleaning has been an active research topic in several communities including statistics, machine learning and data management. The quality of data suffers from typing mistakes, lack of standards for recording database fields, integrity constraints that are not enforced, inconsistent data mappings, etc. For years, data quality technology has developed independently from core data management. Data quality tools became part of Extract Transform Load (ETL) technologies, commonly applied during the initial loading phase of data into a warehouse. Although this might be a viable approach for data analytics, where the data processed are static, it is far from acceptable for operational databases, which are dynamic and face proliferating quality problems that degrade common business practices.

Recently, there has been a major focus on tighter integration of data quality technology with database technology. In particular there has been research work on the efficient realization of popular data cleaning algorithms inside database engines as well as studies for the efficient realization of data quality primitives in a declarative way. The approaches are complementary, the former assuring great performance and the latter ease of deployment and integration with existing applications without modification of the underlying database engine. We are concerned with declarative implementations of data quality primitives in this paper. In particular we study declarative realizations of several *similarity predicates* for the popular approximate (flexible) selection operation for data de-duplication [17]. A similarity predicate $sim()$ is a predicate that numerically quantifies the “similarity” or “closeness” of two (string) tuples. Given a relation R, the approximate selection operation using similarity predicate $sim()$, will report all tuples $t \in R$ such that $sim(t_q, t) \geq \theta$, where θ is a specified numerical ”similarity threshold” and t_q is a query tuple. Approximate selections are special cases of the approximate join (record linkage, similarity join) operation [17]. Several efficient declarative implementations of this operation for specific similarity predicates have been proposed [17] both for approximate selections and joins.

In this paper, we conduct a thorough study of declarative realizations of similarity predicates for approximate selections. We introduce and adapt novel predicates, realize them declaratively and compare them with existing ones for accuracy and performance. In particular we make the following contributions:

- Inspired by the success of *tf-idf cosine similarity* from information retrieval [24] as a similarity predicate for approximate selections, we introduce declarative realizations of other successful predicates from information retrieval and in particular the popular BM25 measure.

SIGMOD'07, June 12-14, 2007, Beijing, China
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

- We introduce declarative realizations of probabilistic similarity predicates inspired by Language Models from information retrieval [20] and Hidden Markov Models [19], suitably adapted for the case of approximate selections.

- We present declarative realizations of previously proposed similarity predicates for the approximate selection problem and we propose a categorization of all measures both previously proposed and new according to their characteristics.

- We present a thorough experimental study comparing all similarity predicates for accuracy and performance, under various types of quality problems in the underlying data.

The rest of this paper is organized as follows. In Section 2, we describe related work in the area of data quality and Information Retrieval. We present our overall framework in Section 3, along with the similarity predicates considered and their classification. Declarative realizations of the predicates in each class are discussed in Section 4, where we also provide SQL expressions required for the different predicates. Finally, we experimentally evaluate the performance of each of the similarity predicates and compare their accuracy in Section 5.

2. RELATED WORK

Data quality has been an active research topic for many years. A collection of statistical techniques have been introduced initially for the record linkage problem [9, 8]. The bulk of early work on data quality was geared towards correcting problems in census files [27]. A number of similarity predicates were developed taking into account the specific application domain (i.e., census files) for assessing closeness between person names (e.g., Jaro [15], Jaro-Winkler [27], etc).

An initial work geared towards database tuples was the merge/purge technique [14]. The work of Cohen [6] introduced the use of primitives from information retrieval (namely cosine similarity, utilizing tf-idf[24]) to identify flexible matches among database tuples. A performance/accuracy study conducted by Cohen et al. [7] demonstrated that such techniques outperform common predicates introduced for specific domains (e.g., Jaro, Jaro-Winkler, etc).

Several predicates to quantify approximate match between strings have been utilized for dealing with quality problems, including edit distance and its variants [13]. Hybrid predicates combining notions of edit distance and cosine similarity have also been introduced [4, 1]. Recently, [5, 2] presented SSJOIN, a primitive operator for efficient set similarity joins. Utilizing ideas from [26], such an operator can be used for approximate matching based on a number of similarity functions, including hamming distance, edit-distance and Jaccard similarity. However, the choice of the similarity predicate in this approach is limited [2]. The bulk of the techniques and predicates however have been introduced without a declarative framework in mind. Thus, integrating them with applications utilizing databases in order to enable approximate selections is not very easy.

Gravano et al. [11] and Galhardas et al. [10], introduced a declarative methodology for realizing approximate joins and selections for edit distance. Subsequently a declarative framework for realizing tf-idf cosine similarity was introduced [12, 16, 17].

There has been a great deal of research in the information retrieval literature on weighting schemes beyond cosine similarity with tf-idf weighting. Recent IR research has shown BM25 to be the most effective among the known weighting schemes [23]. This weighting scheme models the distribution of within-document term frequency, document length and query term frequency very accurately. Moreover, in the information retrieval literature, language modeling has been a very active research topic as an alternative scheme to weight documents for their relevance to user queries. Starting with Ponte and Croft [20], language models for information retrieval have been widely studied.

Hidden Markov Models (HMM) have been very successful in machine learning and they have been utilized for a variety of learning tasks such as named entity recognition and voice recognition [21]. They have been utilized for information retrieval as well [19]. An experimental study on TREC data demonstrated that an extremely simple realization of HMM outperforms standard tf-idf for information retrieval [19]. Finally, researchers (e.g., [22]) have also tried to formally reason about the relative goodness of information retrieval weighting schemes.

3. FRAMEWORK

Let Q be a query string and D a string tuple from a base relation $R = \{D_i : 1 \leq i \leq N\}$. We denote by $\mathcal{Q}$, $\mathcal{D}$ the set of *tokens* in Q and D respectively. We refer to substrings of a string as tokens in a generic sense. Such tokens can be words or q-grams (sequence of q consecutive characters of a string) for example. For Q='db lab', $\mathcal{Q}$={'db', 'lab'} for word-based tokenization and $\mathcal{Q}$={'db ' ,'b l',' la', 'lab'} for tokenization using 3-grams. We refer to tokens throughout the paper when referring to words or q-grams. We make the choice specific (word or q-gram) for techniques we present, when absolutely required. In certain cases, we may associate a *weight* with each token. Several weighting mechanisms exist. We present our techniques referring to weights of tokens, making the choice of the weighting scheme concrete when required. In Section 5 we realize our techniques for specific choice of tokens and specific weighting mechanisms.

Our goal is to calculate a *similarity score* between Q and D using a similarity predicate. We group similarity predicates into five classes based on their characteristics, namely:

- **Overlap predicates:** These are predicates that assess similarity based on the overlap of tokens in $\mathcal{Q}, \mathcal{D}$.

- **Aggregate Weighted Predicates:** Predicates that assess similarity by manipulating weights (scores) assigned to elements of $\mathcal{Q}, \mathcal{D}$.

- **Language Modeling Predicates:** Predicates that are based on probabilistic models imposed on elements of $\mathcal{Q}, \mathcal{D}$.

- **Edit Based Predicates:** Predicates based on a set of edit operations applied between Q and D.

- **Combination Predicates:** Predicates combining features from the classes above.

The classes were defined by studying the properties of previously proposed similarity predicates as well as ones newly proposed herein. Within each class we discuss declarative realizations of predicates.

3.1 Overlap Predicates

Suppose $\mathcal{Q}$ is the set of tokens in the query string Q and $\mathcal{D}$ is the set of tokens in the string tuple D. The *IntersectSize* predicate [26] is simply the number of common tokens between Q and D, i.e.:

$$sim_{intersect}(Q, D) = |\mathcal{Q} \cap \mathcal{D}| \tag{1}$$

Jaccard similarity [26] is the fraction of tokens in Q and S that are present in both, namely:

$$sim_{Jaccard}(Q, D) = \frac{|\mathcal{Q} \cap \mathcal{D}|}{|\mathcal{Q} \cup \mathcal{D}|} \tag{2}$$

If we assign a weight $w(t)$[1] to each token t, we can define weighted versions of the above predicates. *WeightedMatch* [26] is the total weight of common tokens in $\mathcal{Q}$ and $\mathcal{D}$, i.e., $\sum_{t \in \mathcal{Q} \cap \mathcal{D}} w(t)$. Similarly, *WeightedJaccard* is the sum of the weights of tokens in $|\mathcal{Q} \cap \mathcal{D}|$ divided by the sum of the weights of tokens in $|\mathcal{Q} \cup \mathcal{D}|$.

3.2 Aggregate Weighted Predicates

The predicates in this class encompass predicates widely adopted from information retrieval (IR). A basic task in IR is, given a query, identifying *relevant documents* to that query. In our context, we would like to identify the *tuples* in a relation that are *similar* to a query string.

Given a query string Q and a string tuple D, the similarity score of Q and D in this class of predicates is of the form $sim(Q, D) = \sum_{t \in \mathcal{Q} \cap \mathcal{D}} w_q(t, Q) w_d(t, D)$, where $w_q(t, Q)$ is the query-based weight of the token t in string Q and $w_d(t, D)$ is the tuple-based weight of the token t in string D.

3.2.1 Tf-idf Cosine Similarity

The tf-idf cosine similarity [24] between a query string Q and a string tuple D is defined as follows:

$$sim_{cosine}(Q, D) \quad = \quad \sum_{t \in \mathcal{Q} \cap \mathcal{D}} w_q(t, Q) w_d(t, D) \tag{3}$$

where $w_q(t, Q), w_d(t, D)$ are the normalized tf-idf weights [24]. The normalized tf-idf for a token t and a string S, $w(t, S)$ is given by:

$$w(t, S) = \frac{w'(t, S)}{\sqrt{\sum_{t' \in \mathcal{S}} w'(t', S)^2}} \quad , \quad w'(t, S) = tf(t, S).idf(t)$$

The idf term makes the weight of a token inversely proportional to its frequency in the database; the tf term makes it proportional to its frequency in S. Intuitively, this assigns low scores to frequent tokens in the database and high scores to rare tokens in the database. More discussion is available elsewhere [6, 12].

3.2.2 BM25 Predicate

The $BM25$ similarity score between a query string Q and a tuple D, is given as:

$$sim_{BM25}(Q, D) = \sum_{t \in \mathcal{Q} \cap \mathcal{D}} w_q(t, Q) w_d(t, D) \tag{4}$$

where

$$\begin{aligned} w_q(t, Q) &= \frac{(k_3 + 1) * tf(t, Q)}{k_3 + tf(t, Q)} \\ w_d(t, D) &= w^{(1)}(t, R) \frac{(k_1 + 1) * tf(t, D)}{K(D) + tf(t, D)} \end{aligned}$$

$w^{(1)}$ is a modified form of Robertson-Sparck Jones weight:

$$\begin{aligned} w^{(1)}(t, R) &= log\left(\frac{N - n_t + 0.5}{n_t + 0.5}\right) \\ K(D) &= k_1 \left((1 - b) + b\frac{|D|}{avgdl}\right) \end{aligned} \tag{5}$$

and N is the number of tuples in the base relation R, n_t is the number of tuples in R containing the token t, $tf(t, D)$ is the frequency of occurrence of the token t within tuple D, $|D|$ is the number of tokens of tuple D, $avgdl$ is the average number of tokens per tuple, i.e. $\frac{\sum_{D \in R} |D|}{N}$ and k_1, k_3, and b are independent parameters. For TREC-4 experiments [23], $k_1 \in [1, 2]$, $k_3 = 8$ and $b \in [0.6, 0.75]$.

3.3 Language Modeling Predicates

A language model is a form of a probabilistic model. To realize things concretely, we base our discussion on a specific model introduced by Ponte and Croft [20]. Given a collection of documents, a language model is inferred for each; then the probability of generating a given query according to each of these models is estimated and documents are ranked according to these probabilities. Considering an approximate selection query, each tuple in the database is considered as a document; a model is inferred for each tuple and the probability of generating the query given the model is the similarity between the query and the tuple.

3.3.1 Language Modeling

The similarity score between query Q and tuple D is defined as:

$$sim_{LM}(Q, D) = \hat{p}(Q|M_D) = \prod_{t \in \mathcal{Q}} \hat{p}(t|M_D) \times \prod_{t \notin \mathcal{Q}} (1 - \hat{p}(t|M_D)) \tag{6}$$

where $\hat{p}(t|M_D)$ is the probability of token t occurring in tuple D and is given as follows:

$$\hat{p}(t|M_D) = \begin{cases} \hat{p}_{ml}(t, D)^{(1.0 - \hat{R}_{t,D})} \times \hat{p}_{avg}(t)^{\hat{R}_{t,D}} & \text{if } tf_{(t,D)} > 0 \\ \frac{cf_t}{cs} & \text{otherwise} \end{cases} \tag{7}$$

$\hat{p}_{ml}(t, D)$ is the maximum likelihood estimate of the probability of the token t under the token distribution for tuple D and is equal to $\frac{tf_{(t,D)}}{dl_D}$ where $tf_{(t,D)}$ is raw term frequency and dl_D is the total number of tokens in tuple D. $\hat{p}_{avg}(t)$ is the mean probability of token t in documents containing it, i.e.,

$$\hat{p}_{avg}(t) = \frac{(\sum_{D_{(t \in D)}} \hat{p}_{ml}(t|M_D))}{df_t} \tag{8}$$

where df_t is the document frequency of token t. This term is used since we only have a tuple sized sample from the distribution of M_D, thus the maximum likelihood estimate is not reliable enough; we need an estimate from a larger amount of data. The term $\hat{R}_{t,d}$ is used to model the risk for a term t in a document D using a geometric distribution:

$$\hat{R}_{t,D} = \left(\frac{1.0}{(1.0 + \bar{f}_{t,D})}\right) \times \left(\frac{\bar{f}_{t,D}}{(1.0 + \bar{f}_{t,D})}\right)^{tf_{t,D}} \tag{9}$$

$\bar{f}_{t,D}$ is the expected term count for token t in tuple D if the token occurred at the average rate, i.e., $p_{avg}(t) \times dl_D$. The intuition behind this formula is that as the tf gets further away from the normalized mean, the mean probability becomes riskier to use as an estimate. Finally, cf_t is the raw count of token t in the collection, i.e. $\sum_{D \in R} tf(t, D)$ and cs is the raw collection size or the total number of tokens in the collection, i.e. $\sum_{D \in R} dl_D$. $\frac{cf_t}{cs}$ is used as the probability of observing a non-occurring token.

3.3.2 Hidden Markov Models

The query generation process can be modeled by a discrete Hidden Markov process. Figure 1 shows a simple yet powerful two-state HMM for this process. The first state, labeled "String" represents the choice of a token directly from the string. The second state, labeled "General English" represents the choice of a token that is unrelated to the string, but occurs commonly in queries.

Suppose Q is the query string and D is a string tuple from the base relation R; the similarity score between Q and D,

[1] Discussion of ways to assign such weights to tokens follows in subsequent sections.

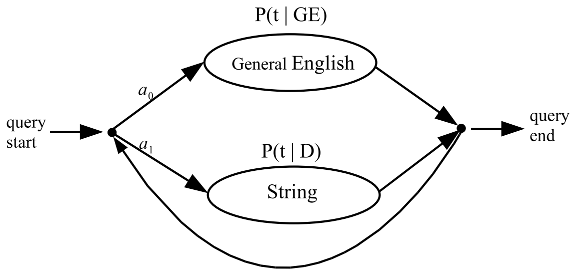


Figure 1: Two State Hidden Markov Model

$sim_{HMM}(Q, D)$, is equal to the probability of generating Q given that D is similar, that is:

$$P(Q|D \text{ is similar}) = \prod_{t\in\mathcal{Q}} (a_0 P(t|GE) + a_1 P(t|D)) \qquad (10)$$

where:

$$P(t|D) = \frac{\text{number of times } t \text{ appears in } D}{\text{length of } D} \qquad (11)$$

$$P(t|GE) = \frac{\sum_{D\in R} \text{number of times } t \text{ appears in } D}{\sum_{D\in R} \text{length of } D} \qquad (12)$$

and a_0 and $a_1 = 1 - a_0$ are transition probabilities of the HMM. The values for these parameters can be optimized to maximize accuracy given training data.

3.4 Edit-based Predicates

An important and widely used class of string matching predicates is the class of edit-based predicates. In this class, the similarity between Q and D is the transformation cost of string Q to D, $tc(Q, D)$. More specifically $tc(Q, D)$ is defined as the minimum cost sequence of edit operations that converts Q to D. Edit operations include *copy, insert, substitute* and *delete* characters in Q and D [13]. Algorithms exist to compute $tc(Q, D)$ in polynomial time [13] but complexity is sensitive to the nature of operations and their operands (individual characters, blocks of consecutive characters, etc). The edit similarity is therefore defined as:

$$sim_{edit}(Q, D) = 1 - \frac{tc(Q, D)}{\max\{|Q|, |D|\}} \qquad (13)$$

Edit operations have an associated cost. In the Levenstein edit-distance [13] which we will refer to as edit-distance, the cost of copy operation is zero and all other operations have unit cost. Other cost models are also possible [13].

3.5 Combination Predicates

We present a general similarity predicate and refer to it as generalized edit similarity (GES) (following [5]). Consider two strings Q and D that are tokenized into *word* tokens and a weight function $w(t)$ that assigns a weight to each word token t. The transformation cost of string Q to D, $tc(Q, D)$ is the minimum cost of transforming Q to D by a sequence of the following transformation operations:

- *token replacement*: Replacing word token t_1 in Q by word token t_2 in D with cost $[1 - sim_{edit}(t_1, t_2)] \cdot w(t_1)$, where $sim_{edit}(t_1, t_2)$ is the edit similarity score between t_1 and t_2.
- *token insertion*: Inserting a word token t into Q with cost $c_{ins} \cdot w(t)$ where c_{ins}, is a constant token insertion factor, with values between 0 and 1.
- *token deletion*: Deleting a word token t from Q with cost $w(t)$.

Suppose $wt(Q)$ is the sum of weights of all word tokens in the string Q. We define the *generalized edit similarity* predicate between a query string Q and a tuple D as follows:

$$sim_{GES}(Q, D) = 1 - \min\left(\frac{tc(Q, D)}{wt(Q)}, 1.0\right) \qquad (14)$$

A related predicate is the SoftTFIDF predicate [7]. In SoftTFIDF, normalized tf-idf weights of word tokens are used along with cosine similarity and any other similarity function $sim(t, r)$ to find the similarity between word tokens. Therefore the similarity score, $sim_{SoftTFIDF}(Q, D)$, is equal to:

$$\sum_{t\in CLOSE(\theta,Q,D)} w(t, Q)\cdot w(\arg\max_{r\in\mathcal{D}}(sim(t, r)), D)\cdot\max_{r\in\mathcal{D}}(sim(t, r)) \qquad (15)$$

where $w(t, Q), w(t, D)$ are the normalized tf-idf weights and $CLOSE(\theta, Q, D)$ is the set of words $t \in \mathcal{Q}$ such that there exists some $v \in \mathcal{D}$ such that $sim(t, v) > \theta$.

4. DECLARATIVE FRAMEWORK

We now describe declarative realizations of predicates in each class. For all predicates, there is a preprocessing phase responsible for tokenizing strings in the base relation, R, and calculating as well as storing related weight values which are subsequently utilized at query time. Tokenization of relation R (`BASE_TABLE`) creates the table `BASE_TOKENS (tid, token)`, where *tid* is a unique tuple identifier for each tuple of `BASE_TABLE` and *token* an associated token (from the set of tokens corresponding to the tuple with identifier *tid* in `BASE_TABLE`). The query string is also tokenized on the fly (at query time) creating the table `QUERY_TOKENS(token)`.

In the rest of this section, we present SQL expressions required for preprocessing and query time approximate selections for the different predicates. In some cases, we re-write formulas to make them amenable to more efficient declarative realization. Due to space limitations, discussion and presentation of SQL is abridged.

4.1 Overlap Predicates

The IntersectSize predicate requires token generation to be completed in a preprocessing step. SQL statements to conduct such a tokenization, which is common to all predicates we discuss, are omitted due to space limitations. The SQL statement for approximate selections with the IntersectSize predicate is shown in Figure 2. The Jaccard coefficient predicate can be efficiently computed by storing the number of tokens for each tuple of the `BASE_TABLE` during the preprocessing step. For this reason we create a table `BASE_DDL(tid, token, len)` where `len` is the number of tokens in tuple with tuple-id `tid`. The SQL statement for conducting approximate selections with the Jaccard predicate is presented in Figure 3.

The weighted overlap predicates require calculation and storage of the related weights for tokens of the base relation during preprocessing. For the WeightedMatch predicate, we store during the preprocessing step the weight of each token redundantly with each `tid, token` pair in a table `BASE_TOKENS_WEIGHTS(tid, token, weight)` in order to avoid an extra join with a table `BASE_WEIGHT(token, weight)` at query time. In order to

```
INSERT INTO INTERSECT_SCORES (tid, score)
SELECT    R1.tid, COUNT(*)
FROM      BASE_TOKENS R1, QUERY_TOKENS R2
WHERE     R1.token = R2.token
GROUP BY  R1.tid
```

Figure 2: SQL Code for IntersectSize

```
INSERT INTO JACCARD_SCORES (tid, score)
SELECT    S1.tid, COUNT(*)/(S1.len+S2.len-COUNT(*))
FROM      BASE_DDL S1, QUERY_TOKENS R2,
          (SELECT COUNT(*) AS len
           FROM QUERY_TOKENS) S2
WHERE     S1.token = R2.token
GROUP BY  S1.tid, S1.len, S2.len
```

Figure 3: SQL Code for Jaccard Coefficient

calculate the similarity score at query time, we use SQL statements similar to that used for the IntersectSize predicate (shown in Figure 2) but replace table BASE_TOKENS by BASE_TOKENS_WEIGHTS and COUNT(*), by SUM(R1.weight).

For the WeightedJaccard predicate, we create during preprocessing a table BASE_DDL(tid, token, weight, len) where weight is the weight of token and len is the sum of weights of tokens in the tuple with tuple-id tid. The SQL statement for approximate selections using this predicate is the same as the one shown in Figure 3 but COUNT(*) is replaced by SUM(S1.weight).

4.2 Aggregate Weighted Predicates

4.2.1 Tf-idf Cosine Similarity

The SQL implementation of the tf-idf cosine similarity predicate has been presented in [12]. During preprocessing, we store normalized tf-idf weights for the base relation in relation BASE_WEIGHTS(tid, token, weight). A normalized weight table QUERY_WEIGHTS(token, weight) for the query string is created on the fly at query time. The SQL statements in Figure 4 will calculate the similarity score for each tuple of the base table.

4.2.2 BM25

Realization of BM25 in SQL involves generation of the table BASE_WEIGHTS(tid, token, weight) storing the weights for tokens in each tuple of the base relation. These weights ($w_d(t, D)$) consist of two parts that could be considered as modified versions of tf and idf. The query weights table QUERY_WEIGHTS(token, weight) can be created on the fly using the following subquery:

```
(SELECT TF.token, TF.tf*(k3+1)/(k3+TF.tf) AS weight
  FROM   ( SELECT T.token, COUNT(*) AS tf
           FROM QUERY_TOKENS T
           GROUP BY   T.token ) TF)
```

The SQL statement shown in Figure 4 will calculate BM25 similarity scores.

```
INSERT INTO SIM_SCORES (tid, score)
SELECT    R1W.tid, SUM(R1W.weight*R2W.weight)
FROM      BASE_WEIGHTS R1W, QUERY_WEIGHTS R2W
WHERE     R1W.token = R2W.token
GROUP BY  R1W.tid
```

Figure 4: SQL Code for Aggregate Weighted Predicates

4.3 Language Modeling Predicates

4.3.1 Language Modeling

In order to calculate language modeling scores efficiently, we rewrite the formulas and finally drop some terms that would not affect the overall accuracy of the metric. Calculating the values in equations (8) and (9) is easy. We build the following relations during preprocessing:
BASE_TF(tid,token,tf) where tf= $tf_{token,tid}$.
BASE_DL(tid,dl) where dl= dl_{tid}.
BASE_PML(tid,token,pml) where pml= $\hat{p}_{ml} = \frac{tf_{token,tid}}{dl_{tid}}$.
BASE_PAVG(token,pavg) where pavg= $\hat{p}_{avg}(token)$.
BASE_FREQ(tid,token,freq) where freq= $\bar{f}_{token,tid}$.
BASE_RISK(tid,token,risk) where risk= $\hat{R}_{token,tid}$.

We omit detailed SQL statements due to space constraints but they mainly involve joins and group by operations. In order to improve the performance of the associated SQL queries, we rewrite the final score formula of equation (6), as follows:

$$\hat{p}(Q|M_D) = \left[\prod_{t\in\mathcal{Q}} \hat{p}(t|M_D)\right] \times \frac{\prod_{\forall t}(1-\hat{p}(t|M_D))}{\prod_{t\in\mathcal{Q}}(1-\hat{p}(t|M_D))} \qquad (16)$$

We slightly change (16) to the following:

$$\hat{p}(Q|M_D) = \left[\prod_{t\in\mathcal{Q}} \hat{p}(t|M_D)\right] \times \frac{\prod_{\forall t\in\mathcal{D}}(1-\hat{p}(t|M_D))}{\prod_{t\in\mathcal{Q}\cap\mathcal{D}}(1-\hat{p}(t|M_D))} \qquad (17)$$

This change results in a large performance gain, since the computation is restricted to the tokens of the query and the tokens of a tuple (as opposed to the entire set of tokens present in the base relation). Experiments demonstrate that accuracy is not significantly affected.

In equation (7), we only materialize the first part (i.e., values of tokens that are present in the tuple D) in the relation BASE_PM during preprocessing (storing the second part would result in unnecessary waste of space). We therefore have to divide all formulas that use $\hat{p}(t|M_D)$ into two parts: one for tokens present in the tuple under consideration and one for all other tokens. So we rewrite the first term in equation (17) as follows:

$$\begin{aligned}\prod_{t\in\mathcal{Q}} \hat{p}(t|M_D) &= \prod_{t\in\mathcal{Q}\cap\mathcal{D}} \hat{p}(t|M_D) \times \prod_{t\in\mathcal{Q}-\mathcal{D}} \hat{p}(t|M_D)\\ &= \prod_{t\in\mathcal{Q}\cap\mathcal{D}} \hat{p}(t|M_D) \times \prod_{t\in\mathcal{Q}-\mathcal{D}} \frac{cf_t}{cs}\\ &= \prod_{t\in\mathcal{Q}\cap\mathcal{D}} \hat{p}(t|M_D) \times \frac{\prod_{t\in\mathcal{Q}} \frac{cf_t}{cs}}{\prod_{t\in\mathcal{Q}\cap\mathcal{D}} \frac{cf_t}{cs}}\end{aligned} \qquad (18)$$

The term $\prod_{t\in\mathcal{Q}} \frac{cf_t}{cs}$ in the above formula is constant for any specific query string, so it can be dropped, since the goal is to find most similar tuples by ranking them based on the similarity scores. Therefore, equation (17) can be written as follows:

$$\hat{p}(Q|M_D) = \frac{\prod_{t\in\mathcal{Q}\cap\mathcal{D}} \hat{p}(t|M_D)}{\prod_{t\in\mathcal{Q}\cap\mathcal{D}} \frac{cf_t}{cs}} \times \frac{\prod_{\forall t\in\mathcal{D}}(1-\hat{p}(t|M_D))}{\prod_{t\in\mathcal{Q}\cap\mathcal{D}}(1-\hat{p}(t|M_D))} \qquad (19)$$

This transformation allows us to efficiently compute similar tuples by just storing $\hat{p}(t|M_D)$ and $\frac{cf_t}{cs}$ for each pair of t and D. Thus, we create table BASE_PM(tid, token, pm, cfcs)

```
INSERT INTO LM_SCORES (tid, score)
SELECT B1.tid2, EXP(B1.score + B2.sumcompm)
FROM (SELECT P1.tid AS tid1, T2.tid AS tid2,
           SUM(LOG(P1.pm)) - SUM(LOG(1.0-P1.pm))
           - SUM(LOG(P1.cfcs)) AS score
       FROM BASE_PM P1, QUERY_TOKENS T2
       WHERE P1.token = T2.token
       GROUP BY P1.tid, T2.tid) B1,
      BASE_SUMCOMPBASE B2
WHERE B1.tid1=B2.tid
```

Figure 5: SQL Code for Language Modeling

where $\mathtt{pm} = \hat{p}(token|M_{tid})$ and $\mathtt{cfcs} = \frac{cf_{token}}{cs}$ as the final result of the preprocessing step. We also calculate and store the term $\prod_{\forall t\in\mathcal{D}}(1 - \hat{p}(t|M_D))$ during preprocessing in relation `BASE_SUMCOMPBASE(tid, sumcompm)`.

The query-time SQL statement to calculate similarity scores is shown in Figure 5. The subquery in the statement computes the three terms in equation 19 that include intersection of query and tuple tokens and therefore needs a join between the two token tables. The fourth term in the equation is read from the table stored during the preprocessing as described above.

4.3.2 Hidden Markov Models

We rewrite equation (10) as follows:

$$\begin{aligned} P(Q|D \text{ is similar}) &= \prod_{t\in\mathcal{Q}} (a_0P(t|GE) + a_1P(t|D)) \\ &= \prod_{t\in\mathcal{Q}} a_0P(t|GE) \times \left[\prod_{t\in\mathcal{Q}}(1+\frac{a_1P(t|D)}{a_0P(t|GE)})\right] \end{aligned} \quad (20)$$

For a specific query, the term $\prod_{t\in\mathcal{Q}} a_0P(t|GE)$ in the above formula is constant for all tuples in the base relation and therefore can be dropped since our goal is to order tuples based on similarity to a specific query string. So the modified similarity score will be:

$$\begin{aligned} sim_{HMM}(Q,D) &= \prod_{t\in\mathcal{Q}}(1+\frac{a_1P(t|D)}{a_0P(t|GE)}) \\ &= \prod_{t\in\mathcal{Q}\cap\mathcal{D}}(1+\frac{a_1P(t|D)}{a_0P(t|GE)}) \end{aligned} \quad (21)$$

In Equation 21, $t \in \mathcal{Q}$ changes to $t \in \mathcal{Q}\cap\mathcal{D}$ because $P(t|D) = 0$ for all $t \notin \mathcal{D}$. Thus we can calculate the term $(1 + \frac{a_1P(t|D)}{a_0P(t|GE)})$ for all `tid, token` pairs during preprocessing and store them as `weight` in relation `BASE_WEIGHTS(tid, token, weight)`. Notice that the term $P(t|D)$ is equal to $\hat{p}_{ml}(t, D)$ in language modeling; we use a relation `BASE_PML(tid, token, pml)` for it. Calculating $P(t|GE)$ and storing it in relation `BASE_PTGE(token, ptge)` is also fairly simple. The final SQL query for preprocessing and the SQL statements for calculating similarity scores, are shown in Figure 6.

4.4 Edit-based Predicates

We use the same declarative framework proposed in [11] for approximate matching based on edit-distance. The idea is to use properties of q-grams created from the strings to generate a candidate set in a way that no false negatives are guaranteed to exist but the set may contain false positives. The set is subsequently filtered by computing the exact edit similarity score between the query and the strings in the candidate set. Computing the edit similarity score is performed using a UDF. The SQL statements for candidate set generation and score calculation are available in [11].

```
                 Preprocessing
INSERT INTO BASE_WEIGHTS(tid,token,weight)
SELECT  M2.tid, M2.token,
          (1 + (a1*M2.pml) / (a0*P2.ptge))
FROM    BASE_PTGE P2, BASE_PML M2
WHERE   P2.token = M2.token
                     Query
INSERT INTO HMM_SCORES (tid, score)
SELECT  W1.tid, EXP(SUM(LOG(W1.weight)))
FROM    BASE_WEIGHTS W1, QUERY_TOKENS T2
WHERE   W1.token = T2.token
GROUP BY W1.tid
```

Figure 6: SQL Code for HMM

4.5 Combination Predicates

Since the calculation of the score function for GES (Equation 14) between a query string and all tuples in a relation could be very expensive, we can first identify a candidate set of tuples similar to the methodology used for edit-distance and then use a UDF to compute exact scores between the query string and the strings in the candidate set. The elements of the candidate set are selected using a threshold θ and the following score formula which ignores the ordering between word tokens. This formula over-estimates $sim_{GES}(Q, D)$ [4]:

$$sim^{Jaccard}_{GES}(Q,D) = \frac{1}{wt(Q)}\sum_{t\in Q} w(t)\cdot\max_{r\in\mathcal{D}}(\frac{2}{q}sim_{Jaccard}(t,r)+d_q) \quad (22)$$

where $wt(Q)$ is the sum of weights of all *word* tokens in Q, $w(t)$ is the *idf* weight for word token t, q is a positive integer indicating the q-gram length extracted from *words* in order to calculate $sim_{Jaccard}(t, r)$ and $d_q = (1 - 1/q)$ is an adjustment term. In order to enhance the performance of the operation, we can employ min-wise independent permutations [3] to approximate $sim_{Jaccard}(t_1, t_2)$ in Equation 22. Description of min-wise independent permutations is beyond the scope of this paper. This would result in substituting $sim_{Jaccard}$ with the min-hash similarity $sim_{mh}(t_1, t_2)$, which is a provable approximation. The resulting metric, GES^{apx}, is shown to be an upper-bound for GES in expectation [4]:

$$sim^{apx}_{GES}(Q,D) = \frac{1}{wt(Q)}\sum_{t\in Q} w(t)\cdot\max_{r\in\mathcal{D}}(\frac{2}{q}sim_{mh}(t,r)+d_q) \quad (23)$$

In order to implement the above predicates, we need to preprocess the relation using the following methodology:

- Tokenization in two levels, first tokenizing into words and then tokenizing each word into q-grams. Word tokens are stored in relation `BASE_TOKENS(tid, token)` and q-grams are stored in `BASE_QGRAMS(tid, token, qgram)`.

- Storing *idf* weights of word tokens in relation `BASE_IDF (token,idf)` as well as the average of *idf* weights in the base relation to be used as *idf* weights of unseen tokens.

- Calculating weights related to the similarity employed to compare tokens, i.e., $sim(t, r)$. For GES^{Jaccard} employing the Jaccard predicate, this includes storing the number of q-grams for each word token in relation `BASE_TOKENSIZE (tid, token, len)`. For GES^{apx}, we have to calculate minhash signatures (required by min-wise independent permutations). SQL statements for generating min-hash signatures and min-hash similarity scores, $sim_{mh}(t, r)$, are omitted due to space constraints.

```
INSERT INTO GESAPX_RESULTS(tid, score)
SELECT  MS.tid, 1.0/SI.sumidf *
        SUM(I.idf*(((2.0/q)*MS.maxsim)+(1-1/q)))
FROM    MAXSIM MS, QUERY_IDF I, SUM_IDF SI
WHERE   MS.token = I.token
GROUP BY MS.tid, SI.sumidf
```

Figure 7: SQL Code for GES^{apx}, GES^{Jaccard}

In order to make the presented statements more readable, we assume that the following auxiliary relations are available to us; in practice, they are calculated on-the-fly as subqueries:

- QUERY_IDF(token, idf) stores idf weights for each token in the query. Weights are retrieved from the base weights relation and the average idf value over all tokens in the base relation is used as the weight of query tokens not present in the base relation. SUM_IDF(token, sumidf) will store sum of idf weights for query tokens.

- MAXSIM(tid, token, maxsim) stores the maximum of the similarity scores between the tokens in tuple tid and each token in the query.

The tables above do not have to be computed beforehand, they are rather computed on the fly at query execution time. Assuming however they are available, the SQL statements for computing the scores for GES^{apx}, GES^{Jaccard} are shown in Figure 7.

SoftTFIDF can also be implemented similar to GES approximation predicates. During preprocessing, we need to first tokenize the string into word tokens and store them in BASE_TOKENS(tid, token). Depending on the function used for similarity score between word tokens, we may need to tokenize each word token into qgrams as well. We then need to store normalized tf-idf weights of tokens in the tuples in the base relation in BASE_WEIGHTS(tid, token, weight).

Here again, at query time, we assess the final score formula of equation (15), in a single SQL statement. For presentation purposes, assume that the following relations have been materialized:

- QUERY_WEIGHTS(token, weight) stores normalized tf-idf weights for each token in the query table.

- CLOSE_SIM_SCORES(tid, token1, token2, sim) stores the similarity score of each token in the query (token2) with each token of each tuple in the base relation, where the score is greater than a threshold θ (θ specified at query time). Such a score could have been computed using a declarative realization of some other similarity predicate or a UDF to compute similarity using a string distance scheme (e.g., Jaro-Winkler [27]).

- MAXSIM(tid, token, maxsim) stores the maximum of the sim score for each query token among all tids in CLOSE_SIM_SCORES relation. MAXTOKEN(tid, token1, token2, maxsim) stores arg max$_{r\in \mathtt{tid}}(sim(token2, r))$ as well, i.e., the token in each tuple in the base relation that has the maximum similarity with a query token token2 in $CLOSE(\theta, Q, D)$

Figure 8 shows the SQL statement for MAXTOKEN table and the final similarity score for SoftTFIDF.

```
INSERT INTO MAXTOKEN(tid,token1,token2,maxsim)
SELECT  CS.tid, CS.token1,
        CS.token2, MS.maxsim
FROM    MAXSIM MS, CLOSE_SIM_SCORES CS
WHERE   CS.tid=MS.tid AND
        CS.token2=MS.token2 AND MS.maxsim=CS.sim

INSERT INTO SoftTFIDF_RESULTS (tid, score)
SELECT  TM.tid, SUM(I.weight*WB.weight*TM.maxsim)
FROM    MAXTOKEN TM,
        QUERY_WEIGHTS I, BASE_WEIGHTS WB
WHERE   TM.token2 = I.token AND TM.tid = WB.tid
        AND TM.token1 = WB.token
GROUP BY TM.tid
```

Figure 8: SQL Code for SoftTFIDF - Query time

5. EVALUATION

We experimentally evaluate the performance of each of the similarity predicates presented thus far and compare their accuracy. The choice of the best similarity predicate in terms of accuracy highly depends on the type of datasets and errors present in them. The choice in terms of performance depends on the characteristics of specific predicates. We therefore evaluate the (a) accuracy of predicates using different datasets with different error characteristics and the (b) performance by dividing the preprocessing and query execution time into various phases to obtain detailed understanding of the relative benefits and limitations. All our experiments are performed on a desktop PC running MySQL server 5.0.16 database system over Windows XP SP2 with Pentium D 3.2GHz CPU and 2GBs of RAM.

5.1 Benchmark

In the absence of a common benchmark for data cleaning, we resort to the definition of our own data generation scheme with controlled error. In order to generate datasets for our experiments, we modify and significantly enhance the UIS database generator which has effectively been used in the past to evaluate duplicate detection algorithms [14]. We use the data generator to inject different types and percentages of errors to a clean database of string attributes. We keep track of the source tuple from which the erroneous tuples have been generated in order to determine precision and recall required to quantify the accuracy of different predicates. The generator allows to create data sets of varying sizes and error types, thus is a very flexible tool for our evaluation. The data generator accepts clean tuples and generates erroneous duplicates based on a set of parameters. Our data generator provides the following parameters to control the error injected in the data:

- the size of the dataset to be generated.

- the fraction of clean tuples to be utilized to generate erroneous duplicates.

- *distribution of duplicates*: the number of duplicates generated for a clean tuple can follow a uniform, Zipfian or Poisson distribution.

- *percentage of erroneous duplicates*: the fraction of duplicate tuples in which errors are injected by the data generator.

- *extent of error in each erroneous tuple*: the percentage of characters that will be selected for injecting character edit error (character insertion, deletion, replacement or swap) in each tuple selected for error injection.

dataset	#tuples	Avg. tuple length	#words/tuple
Company Names	2139	21.03	2.92
DBLP Titles	10425	33.55	4.53

Table 1: Statistics of Clean Datasets

parameter	range
size of dataset	5k - 100k
# clean tuples	500 - 10000
duplicate distribution	uniform, Zipfian
erroneous duplicates	10% - 90%
extent of error per tuple	5% - 30%
token swap error	10% - 50%

Table 2: Range of Parameters Used For Erroneous Datasets

- *token swap error*: the percentage of word pairs that will be swapped in each tuple that is selected for error injection.

We use two different sources of data: a data set consisting of *company names* and a data set consisting of *DBLP Titles*. Statistical details for the two datasets are shown in Table 1. For the company names dataset, we also inject domain specific *abbreviation errors*, e.g., replacing `Inc.` with `Incorporated` and vice versa.

For both datasets, we generate different erroneous datasets by varying the parameters of the data generator as shown in Table 2.

Due to space constraints, we show accuracy results for 8 different erroneous datasets generated from a data set of company names, each containing 5000 tuples generated from 500 clean records, with uniform distribution. We choose to limit the size of the data sets to facilitate experiments and data collection since each experiment is run multiple times to obtain statistical significance. We conducted experiments with data sets of increasing size and we observed that the overall accuracy trend presented remains the same. We consider the results presented highly representative across erroneous data sets (generated according to our methodology) of varying sizes, and duplicate distributions. We classify these 8 datasets into *dirty*, *medium* and *low* error datasets based on the parameters of data generation. We have also generated 5 datasets, each having only one specific type of error, in order to evaluate the effect of specific error types. Table 3 provides more details on the datasets. Table 4 shows a sample of duplicates generated by the data generator from $CU1$ and $CU5$.

5.2 Evaluating Accuracy

Class	Name	Percentage of			
		erroneous duplicates	errors in duplicates	token swap	Abbr. error
Dirty	CU1	90	30	20	50
Dirty	CU2	50	30	20	50
Medium	CU3	30	30	20	50
Medium	CU4	10	30	20	50
Medium	CU5	90	10	20	50
Medium	CU6	50	10	20	50
Low	CU7	30	10	20	50
Low	CU8	10	10	20	50
-	F1	50	0	0	50
-	F2	50	0	20	0
-	F3	50	10	0	0
-	F4	50	20	0	0
-	F5	50	30	0	0

Table 3: Classification of Datasets

	CU1
t_1^1	Stsalney Morgan cncorporsated Group
t_2^1	jMorgank Stanlwey Grouio Inc.
t_3^1	Morgan Stanley Group Inc.
t_4^1	Sanlne Morganj Inocrorpated Group
t_5^1	Sgalet Morgan Icnorporated Group
	CU5
t_1^5	Morgan Stanle Grop Incorporated
t_2^5	Stalney Morgan Group Inc.
t_3^5	Morgan Stanley Group In.
t_4^5	Stanley Moragn Grou Inc.
t_5^5	Morgan Stanley Group Inc.

Table 4: Sample Tuples from CU1 & CU5 Datasets

We measure the accuracy of predicates, utilizing known methods from the information retrieval literature in accordance to common practice in IR [25]. We compute the *Mean Average Precision (MAP)* and *Mean Maximum* F_1 scores of the rankings of each dataset imposed by approximate selection queries utilizing our predicates. Average Precision(AP), is the average of the precision after each similar record is retrieved, i.e.,

$$\frac{\sum_{r=1}^{N}[P(r) \times rel(r)]}{\text{number of relevant records}} \tag{24}$$

where N is the total number of records returned, r is the rank of the record, i.e., the position of the record in the result list sorted by decreasing similarity score, $P(r)$ is the precision at rank r, i.e., the ratio of the number of *relevant* records having rank $\leq r$ to the *total* number of records having rank $\leq r$, and $rel(r)$ is 1 if the record at rank r is relevant to the query and 0 otherwise. This measure emphasizes returning more similar strings earlier. MAP is the mean AP value over a set of queries. Maximum F_1 measure is the maximum F_1 score (the harmonic mean of precision and recall) over the ranking of records, i.e.,

$$\max_r \left[\frac{2 \times Pr(r) \times Re(r)}{Pr(r) + Re(r)}\right] \tag{25}$$

where $Pr(r)$ and $Re(r)$ are precision and recall values for rank r. $Pr(r)$ is as defined above. $Re(r)$ is the ratio of the number of relevant records having rank $\leq r$ to the total number of relevant records. Again, we compute mean maximum F_1 over a set of queries.

Our data generation methodology allows to associate easily a clean tuple with all erroneous versions of the tuple generated using our data generator. A clean tuple and its erroneous duplicates are assigned the same cluster id. Essentially each time we pick a tuple from a cluster, using its string attribute as a query we consider all the tuples in the same cluster (tuples with the same cluster id) as relevant to this query. For each query and a specific predicate, we return a list of tuples sorted in the order of decreasing similarity scores. Thus, it is easy to identify relevant and irrelevant records among the results returned for a specific query and similarity predicate. In order to maintain our evaluation independent of any threshold constants (specified in approximate selection predicates) we do not prune this list utilizing thresholds. For each dataset, we compute the mean average precision and mean maximum F_1 measure over 500 randomly selected queries taken from that data set (notice that our query workload contains both clean as well as erroneous tuples). Thus, our accuracy results represent the *expected* behavior of the predicates over queries and thresholds. We report the values for MAP only since the results were consistently similar for max F_1 measure in all our experiments.

5.3 Settings

5.3.1 Choice of Weights for Weighted Overlap Predicates

Both WeightedMatch (WM) and WeightedJaccard (WJ) predicates require a weighting scheme to assign weights to the tokens. It is desirable to use a weighting scheme which captures the importance of tokens. We experimented with idf and the Robertson-Spark Jones (RS) weighting scheme given in Equation 5 and found that RS weights lead to better accuracy. So in the following discussion, we use RS weights for weighted overlap predicates.

5.3.2 Parameter Settings for Predicates

For all predicates proposed previously in the literature we set any parameter values they require for tuning as suggested in the respective papers. For the predicates presented herein for data cleaning tasks, for the case of BM25, we set k_1=1.5, k_3=8 and b=0.675; for HMM, we set a_0 to 0.2, although our experiments show that the accuracy results are not very sensitive to the value of a_0 as long as a reasonable value is chosen (i.e., a value not close to 0 or 1).

The SoftTFIDF predicate requires a similarity predicate over the word tokens. We experimented with various similarity predicates like Jaccard, IntersectSize, edit distance, Jaro-Winkler, etc. and choose Jaro-Winkler since SoftTFIDF with Jaro-Winkler (STfIdf w/JW) performs the best. This was also observed in [7]. Two words are similar in SoftTFIDF if their similarity score exceeds a given threshold θ. SoftTFIDF with Jaro-Winkler performed the best with θ=0.8. Finally, we set c_{ins} for GES predicate to 0.5 as proposed in [4]. For calculating accuracy, we use the exact GES as shown in Equation 14. We remark that we do not prune the results based on any threshold in order to keep the evaluation independent of the threshold values.

5.3.3 Q-gram Generation

Qgram generation is a common preprocessing step for all predicates. We use an SQL statement similar to that presented in [11] to generate q-grams, with a slightly different approach. We first insert $q-1$ special symbols (e.g. \$) in place of all whitespaces in each string, as well as at the beginning and end of the strings. In this way we can fully capture all errors caused by different orders of words, e.g., “Department of Computer Science” and “Computer Science Department”. For qgram generation we also need to have an optimal value of qgram size (q). A lower value of q ignores the ordering of characters in the string while a higher value can not capture the edit errors. So an optimum value is required to capture the edit errors taking in account the ordering of characters in the string. The table below shows the accuracy comparison of different qgram based predicates (Jaccard, tf-idf (Cosine), HMM and BM25) in the dirty cluster of our data sets:

q	Jaccard	Cosine	HMM	BM25
2	0.736	0.783	0.835	0.840
3	0.671	0.769	0.807	0.805

The trend is similar for other predicates and the accuracy further drops for higher values of q. Thus, we set q=2 as it achieves the best accuracy results.

5.4 Accuracy Results

In this section we present a detailed comparison of the effectiveness of the similarity predicates in capturing the different types of error introduced in the data.

Abbreviation error: Due to abbreviation errors, a tuple AT&T Incorporated gets converted to AT&T Inc. Note that Incorporated and Inc are frequent words in the company names database. For the query AT&T Incorporated, the unweighted overlap predicates Jaccard (Jac.) and IntersectSize (Xect) will assign to the tuple IBM Incorporated greater similarity score than to the tuple AT&T Inc since they just try to match tuples on the basis of common qgrams. Edit distance (ED) will behave similarly since it is cheaper to convert AT&T Incorporated to IBM Incorporated than to AT&T Inc. The weight based predicates are robust to abbreviation errors since they assign high weights to tokens corresponding to rare (important) words e.g. AT&T. Table 5 presents the accuracy of the predicates for the case of a data set with only abbreviation error (dataset F1). All other predicates WeightedMatch(WM), WeightedJaccrd(WJ), tf-idf(Cosine), BM25, HMM, Language Modeling(LM) and SoftTFIDF(STfIdf w/JW) had near perfect accuracy. Similar behavior is observed when the percentage of duplicates and abbreviation error is varied.

Predicate group	F3	F4	F5
GES	1.0	.99	.97
BM25, HMM, LM, STfIdf w/JW	1.0	.97	.91
edit distance	.99	.97	.90
WM , WJ, Cosine	.99	.93	.85
Jaccard (Jac.), IntersectSize (Xect)	.99	.91	.81

Table 6: Accuracy: Only Edit Errors

Token swap errors: Due to token swap errors, a tuple Beijing Hotel gets converted to Hotel Beijing. Suppose there is a tuple Beijing Labs present in the database, where Labs and Hotel are equally important tokens but more frequent than Beijing. For a query Beijing Hotel, edit distance and GES will claim Beijing Labs more similar to the query than Hotel Beijing. We remark that for accuracy calculation, we use exact GES as shown in Equation 14. All other predicates ignore the order of words, and hence will perform well for token swap errors. Table 5 shows the accuracy of the predicates for a data set with only token swap errors (dataset F2). All other predicates had near perfect accuracy. Similar trend is observed when the percentage of duplicates and token swap error is varied.

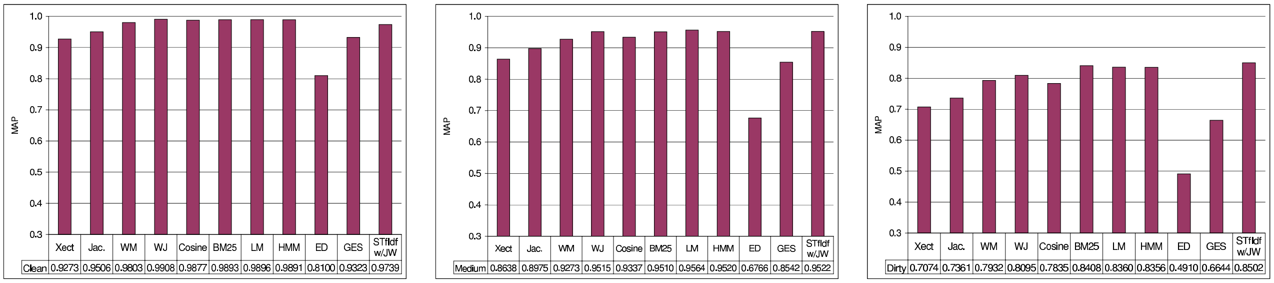
Edit errors: Edit errors involve character insertion/ deletion/ replacement and character swap. The number of positions of a string at which edit error has occurred defines the extent of the edit error. All the predicates discussed above are robust towards low edit errors but their accuracy degrades as the extent of edit error increases. Table 6 shows the accuracy result for various predicates for increasing edit error in the data (datasets F3, F4 and F5). The predicates giving near equal accuracy are grouped together. GES is most resilient to edit errors. Edit distance, designed to capture edit errors has average performance. BM25, STfIdf w/JW, and probabilistic predicates (LM and HMM) are competitive in catching edit errors and perform slightly better than edit distance. The weighted overlap predicates (WM and WJ) with RS weights perform equivalent to tf-idf (Cosine) but not as good as edit distance. Finally the unweighted overlap predicates Jaccard and IntersectSize perform the worst as they ignore the importance of tokens. Similar trend is observed when the percentage of erroneous duplicates is varied.

5.4.1 Comparison of predicates

Figure 9 shows MAP values for different predicates for the 3 classes of erroneous datasets described in Table 3. For the low error datasets, all the predicates perform well except edit distance, GES, IntersectSize and Jaccard. GES performs a little worse due to the presence of token swap errors, IntersectSize and Jaccard perform

Type of Error	Xect	Jac.	WM	WJ	Cosine, BM25, LM, HMM	ED	GES	S TfIdf w/JW
abbr. error (F1)	0.94	0.96	0.98	1.0	1.0	0.89	1.0	1.0
token swap error (F2)	1.0	1.0	1.0	1.0	1.0	0.77	0.94	1.0

Table 5: Accuracy: Abbr. and Token Swap Errors



(a) Low Error Datasets

(b) Medium Error Datasets

(c) Dirty Datasets

Figure 9: MAP values for different predicates on different datasets

worse because of abbreviation errors and edit distance is the worst because of both factors.

When the error increases, the three types of errors occur in combination and edit based predicates experience large accuracy degradation. The edit based predicates are already not good at handling token swap errors and the presence of edit errors deteriorates their effectiveness since the word token weights are no longer valid. This is not the case for the qgram based predicates since edit errors affect only a small fraction of qgrams and the remaining qgram weights are still valid. Consider a query Q=`Morgan Stanley Group Inc.` over dataset $CU5$, where we expect to fetch the tuples shown in Table 4. The qgram based predicates are able to return all the tuples at the top 5 positions in the list according to similarity values. GES is not able to capture the token swap and it ranks t_2^5 and t_4^5 at position 27 and 28 respectively. The edit distance predicate performs worse; both t_2^5 and t_4^5 are absent from the list of top 40 similar tuples. Both edit based predicates give high similarity score to tuples like `Silicon Valley Group, Inc.` for query Q primarily because of low edit distance between `Stanley` and `Valley`.

The unweighted overlap predicates ignore the importance of qgrams and hence perform worse than the predicates that incorporate weights. It is interesting to note that the weighted overlap predicates perform better than the tf-idf (`cosine`) predicate. This is due to the RS weighting scheme (Equation 5) for weight assignment of tokens which has been shown to be more accurate than the idf weighting scheme. The former captures importance of tokens more accurately than the latter. The language modeling predicates (HMM and LM), and BM25 are always the best in all the three datasets. The success of the SoftTFIDF is attributed to the underlying Jaro-Winkler word level similarity predicate which can match the words accurately even in the presence of high errors.

We also experimented with GES^{Jaccard} and GES^{apx}. Both predicates make use of a threshold θ to prune irrelevant records without calculating the exact scores. Depending on the value of θ, relevant records might also be pruned leading to a drop in accuracy. Table 7 shows the variation in accuracy for GES^{Jaccard} and GES^{apx} for threshold values (θ) 0.7, 0.8 and 0.9 for dataset $CU1$ for which GES (with no threshold) has 69.7% accuracy. For GES^{apx} we used 5 min hash signatures in order to approximate the GES^{Jaccard}. We observe that increasing the number of min-hash signatures takes more time without having a significant impact on accuracy (pretty soon it demonstrates diminishing returns). A small number of min hash signatures results in significant accuracy loss.

Predicate	θ =0.7	θ =0.8	θ =0.9
GES^{Jaccard}	0.692	0.683	0.603
GES^{apx}	0.678	0.665	0.608

Table 7: Accuracy of GES Predicates for Different Thresholds

Experimental results show that for suitable thresholds GES^{Jaccard} performs as good as GES and the accuracy drops as the threshold increases. GES^{apx}, being an approximation for GES^{Jaccard}, performs slightly worse than GES^{Jaccard}. Similar results were observed for other datasets.

5.5 Performance Results

In this section, we compare different predicates based on preprocessing time, query time and how well they scale when the size of the base table grows. As expected, the performance depends primarily on the size of the base table. Performance observations and trends remain relatively independent from the error rate of the underlying data sets. Thus, we present the experiments on the DBLP datasets with increasing size and medium amount of errors: 70% of erroneous duplicates, 20% extent of error, 20% token swap error and no abbreviation error.

5.5.1 Preprocessing

We divide preprocessing time for a data set to make it amenable for approximate selection queries into two phases. In the first phase, tokenization is performed. Qgrams are extracted from strings in the way described in section 5.3.3 and stored in related tables. Aggregate weighted (Cosine and BM25) and language modeling predicates (LM and HMM) are fastest in this phase, followed by overlap predicates (Xect and Jac.) with a small difference which is due to storing distinct tokens only. Combination predicates (`GES Jac`, `GES apx` and `STfIdf w/JW`) are considerably slower in this phase since they involve an extra level of tokenization into words.

In the next phase, related weights are calculated and assigned to tokens. In this phase, the fastest predicates are the overlap predicates and edit distance (ED) followed by $GES^{Jaccard}$ and SoftTFIDF that only require weight calculation for word tokens. Aggregate weighted and language modeling predicates are considerably slower since calculating weights in these predicates involves a

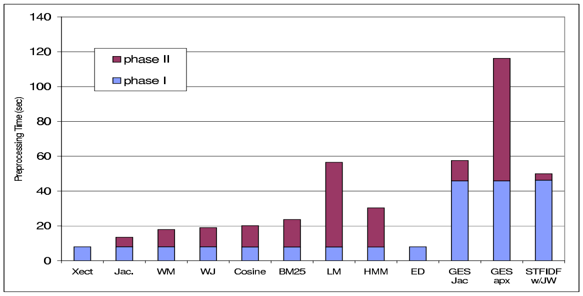


Figure 10: **Preprocessing time of different predicates**

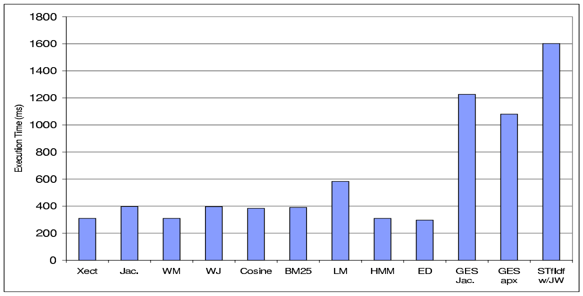


Figure 11: Query time of different predicates

lot of computation and creation of many intermediate tables. Language modeling (LM) is the slowest predicate among probabilistic predicates since it requires the maximum number of intermediate tables to be created and stored. GES^{apx} requires to compute min-hash signatures for the tokens separately for a number of hash functions on top of the two level tokenization and IDF weight calculation, so it is the slowest of all predicates. Figure 10 shows the preprocessing times for all predicates on a dataset of 10,000 records with an average length of 37 characters. GES^{apx} in this Figure employs min-hash computation utilizing 5 hash functions (min hash signature size of 5). Preprocessing time for GES^{apx} increases with increasing number of hash functions employed for min-hash signature calculation.

5.5.2 *Query time*

Query time for a predicate is the time taken to rank the tuples from the base table according to decreasing similarity score. Query time can also be divided into two phases: preprocessing the query string and computing similarity scores. The preprocessing part can itself be divided into tokenization and weights computation phases as done for preprocessing of the base relation. We didn't experience large variability in the time for query preprocessing among all predicates. As described in section 4.3 the score formulas for Language modeling and HMM are suitably modified by dropping query dependent terms which do not alter the similarity score order and hence, the accuracy of the predicates.

Figure 11 shows the average query execution time of different predicates over 100 queries on a table of 10,000 strings with an average length of 37 characters. The experimental results are consistent with our analysis. A comparison of the average query time of the predicates shows that IntersectSize, Jaccard, WeightedMatch, WeightedJaccard, HMM, BM25 should be among the best since first, they just involve one join and second, the query token weights do not depend on idf and are easy to compute. We expect the Cosine predicate to follow these predicates as it has the additional overhead of calculating query weights which depend on idf of tokens. The Language Modeling predicate involves join of 3 tables, so it is comparatively slow. The GES based predicates are slowest of all since they involve identification of the best matching token among the tuples for each query token. GES^{apx} has been designed to efficiently approximate $\mathrm{GES}^{Jaccard}$, so it is expected to be the fastest of all GES based predicates. Note that the filtering step of $\mathrm{GES}^{Jaccard}$, GES^{apx} and edit distance require a suitable threshold θ. Lower value of θ results in poor filtering and high post-processing time, while higher value of θ leads to loss of similar results and hence a drop in accuracy. We used θ=0.8 for the filtering step in $\mathrm{GES}^{Jaccard}$ and GES^{apx} and θ=0.7 for edit distance, since these values balance the trade-off between the performance and precision for these predicates. For GES^{apx}, we use 5 hash functions for min-hash calculation (min hash signature of 5).

5.5.3 *Scalability*

In order to investigate the scalability of our approach, we run experiments on DBLP datasets with sizes varying from 10k to 100k records. The variation in query time as the base table size increases is shown in Figure 12. The predicates with nearly equal query execution times have been grouped together. Group `G1` includes predicates IntersectSize, WeightedMatch and HMM, and the group `G2` includes Jaccard, WeightedJaccard, Cosine and BM25. For predicates other than combination predicates, the results are consistent with our analysis of query execution time presented in Section 5.5.2. The predicates in group G1 can be thought of having a weight of 1 for query tokens and they just require a single join to compute similar tuples. The predicates in group G2 take slightly more time than predicates in G1 since they have to calculate weights for query tokens. LM requires join of three tables to get results so it is considerably slower than predicates in G1 and G2. For the case of combination predicates, query time depends highly on the value of threshold θ used for these predicates and the number of words in the string. We use the same thresholds we used in Section 5.5.2 for these predicates. We also limit the size of the query strings to three words in order to be able to compare the values among different datasets with other predicates. The results show that combination predicates are significantly slower than other predicates since for each query token, we need to determine the best matching token from the base tuple using an auxiliary similarity function such as Jaccard and Jaro-Winkler, apart from the time needed to calculate related weights for word tokens. GES^{apx} is the fastest in this cluster of predicates. Increasing the number of words in query strings considerably slows down these predicates. We excluded edit distance from this experiment because of its significantly poor accuracy.

5.6 Summary of Evaluation

We presented an exhaustive evaluation of approximate selection predicates by grouping them into five classes based on their characteristics: overlap predicates, aggregate weighted predicates, edit-based predicates, combination predicates and language modeling predicates. We experimentally show how predicates in each of these classes perform in terms of accuracy, preprocessing and execution time. Within our framework, the overlap predicates are relatively efficient but have low accuracy. Edit based predicates perform worse in terms of accuracy but are relatively fast due to the filtering step they employ. The aggregate weighted predicates, specifically BM25, perform very well both in terms of accuracy and efficiency. Both the predicates from the language modeling cluster perform well in terms of accuracy. Moreover, HMM is as fast as

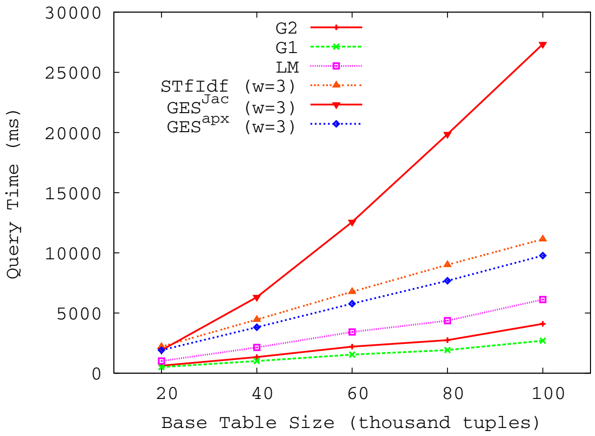


Figure 12: Query Time: Variation in Base Table Size

simple overlap predicates. The combination predicates are considerably slow due to their two levels of tokenization. Among the combination predicates, GES based predicates are robust in handling edit errors but fail considerably in capturing token swap errors. SoftTFIDF with Jaro-Winkler performs nearly equal to BM25 and HMM and is among the best in terms of accuracy, although it is the slowest predicate. This establishes the effectiveness of BM25 and HMM predicates for approximate matching in large databases.

6. CONCLUSIONS

We proposed new similarity predicates for approximate selections based on probabilistic information retrieval and presented their declarative instantiation. We presented an in-depth comparison of accuracy and performance of these new predicates along with existing predicates, grouping them into classes based on their primary characteristics. Our experiments show that the new predicates are both effective as well as efficient for data cleaning applications.

7. REFERENCES

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB '02*.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB '06*.

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 2000.

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD '03*.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE '06*.

[6] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD '98*.

[7] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb '03*.

[8] J. B. Copas and F. J. Hilton. Record linkage: statistical models for matching computer records. *Journal of the Royal Statistical Society*, 1990.

[9] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 1969.

[10] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB '01*.

[11] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB '01*.

[12] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins for web data integration. In *WWW 2003: 90-101*.

[13] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[14] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 1998.

[15] M. A. Jaro. Advances in record linkage methodology as applied to matching the 1985 census of tampa. *Journal of the American Statistical Association*, 1984.

[16] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB '04*.

[17] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. Tutorial in *SIGMOD '06*.

[18] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*, 1999.

[19] D. R. H. Miller, T. Leek, and R. M. Schwartz. A hidden Markov model information retrieval system. In *SIGIR '99*.

[20] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR '98*.

[21] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, 1989.

[22] S. Robertson. Understanding inverse document frequency: on theoretical arguments. *Journal of Documentation*, 2004.

[23] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, M. Gatford, and A. Payne. Okapi at trec-4. In *TREC '95*.

[24] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 1988.

[25] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[26] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD '04*.

[27] W. E. Winkler. The state of record linkage and current research problems. US Bureau of the Census, 1999.