AKYRA: Efficient Keyword-Query Cleaning in Relational Databases

Rares Vernica
rares@ics.uci.edu
Chen Li
chenli@ics.uci.edu

April 12, 2011

Technical Report
Information Systems Group
Department of Computer Science
University of California, Irvine

AKYRA: Efficient Keyword-Query Cleaning in Relational Databases

Rares Vernica, Chen Li

Department of Computer Science, University of California, Irvine, CA 92697, USA

Abstract

Databases often cannot find answers to keyword queries due to inconsistencies in the data or typos in the queries. Solving this problem requires efficient techniques to clean keywords in queries. In this paper we systematically study different approaches to cleaning keyword queries in relational databases. We first consider applications where we can support queries outside a database. We develop a query engine with various important operators for answering queries by relaxing keywords. We focus on an operator that suggests several related keywords for a query keyword. This operator is critical to query cleaning and tends to be computationally expensive. We develop novel algorithms for implementing this operator efficiently, by considering both the similarity between keywords and the importance of a suggested keyword. We also consider applications that require query cleaning inside an existing database. We study different alternatives for leveraging existing DBMS features such as user-defined functions and auxiliary tables for answering keyword queries. We conducted thorough experiments on real data sets and report the observations on the performance, limitations, and tradeoffs of the different approaches.

Keywords: keyword, search, approximate, cleaning, database

1. Introduction

Consider an e-commerce application where users search for products using keywords. Figure 1 shows a snippet of its Products table, which stores the id and description for each product. Consider a query with two keywords “givanchy” and “t-shirt”. There is no record in the table that matches both keywords exactly, so an algorithm based on exact matching will return an empty answer to the user. However, there are several records with keywords similar to the query keywords, which might be of interest to the user. In order to find these interesting records, the system needs to “clean” the query by relaxing the keywords to other possibly related keywords.

<table>
<thead>
<tr>
<th>RID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>...popular givenchy fragrance</td>
</tr>
<tr>
<td>11</td>
<td>...adidas t-shirt</td>
</tr>
<tr>
<td>12</td>
<td>...elegant givenchy shirt</td>
</tr>
<tr>
<td>13</td>
<td>...casual shirt</td>
</tr>
<tr>
<td>14</td>
<td>...fancy tshirt</td>
</tr>
<tr>
<td>15</td>
<td>...givanchy logo tshirt</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: A table about products.

In general, keyword queries in relational databases need to be cleaned due to the gap between user queries and the underlying data. The data can have inconsistencies, and queries can have typos possibly due to users’ limited knowledge about the data. The same real-world entity can have different representations. As another example, the ACM SIGMOD 2001 conference could be represented as “SIGMOD 2001” or “SIGMOD’01”, and we may want to find records with these representations even if a user types in a keyword “SIGMOD”.

To solve similar problems in Web search, many search engines provide a “Did you mean” feature, which offers suggestions for the user’s keywords. In the running example, we could use the same idea by recommending a new keyword for each query keyword. For instance, we could recommend a new keyword “givenchy” for “givanchy”, recommend “tshirt” for “t-shirt”, and then find records that match these two new keywords exactly. However, there are still records in the table that might be interesting to the user, such as record 15, which do not match the new keywords exactly. Note that a disjunctive query might overwhelm the user with records containing on one of the keywords. In fact, a close look at the table suggests that no single recommended query with keywords similar to the original query keywords can find all the records that might be of interests to the user. Thus a better solution is to clean the query by finding the records with keywords similar to the query keywords, even if the records have different similar keywords, i.e., query contains conjunctions of disjunctions.

Contributions: In this paper we focus on two challenges to clean keyword queries based on finding records with similar keywords. The first one is about how to support it in different types of applications. In some applications, we can retrieve the data from its existing database, and build a custom layer to support query cleaning and processing. In this case, we have a lot of flexibility to build a query engine with operators using
new efficient index structures and search algorithms. On the other hand, there are also many applications that are already developed inside a DBMS, which requires the query-cleaning feature be implemented inside the existing DBMS, so that we can minimize the amount of changes on their existing database and programs on top of the database. In this case, data needs to remain inside the DBMS, and query cleaning search is performed by leveraging the existing DBMS query engine as much as possible. We need to study the tradeoffs and performance bottlenecks in these different types of applications.

The second challenge is how to support efficient keyword relaxation. There are many ways a keyword can be relaxed, including prefix matching, synonym substitution or even dropping keywords from the query. In this paper, we focus on query relaxation based on keyword similarity. Specifically, we relax a keyword by finding the most relevant keywords based on their similarity with the original keyword and their importance. In the literature there have been studies on string similarity search assuming a similarity threshold is given. In addition to utilizing these existing techniques, we also need novel indexes and algorithms for finding top-k similar keywords, which have not been studied in the past. Further, our query-cleaning method can find different records with keywords similar to different relaxations of the query keywords. This feature implicitly requires answering multiple conjunctive queries, which have a high demand for efficient algorithms.

We present a query-cleaning framework called AKYRA, which stands for Approximate KeYword search in Relational dAtabases. In Section 2 we formulate the problem of query cleaning using keyword relaxation based on similarity search. In Section 3 we present a keyword query cleaning engine built for the case where data sits outside the DBMS. We present query processing operators and discuss optimization issues. In Section 4 we develop efficient algorithms for keyword relaxation based on similarity and keyword weight. We first present an iterative algorithm that can leverage existing keyword relaxation algorithms developed for the case where a similarity threshold is given. We then develop an algorithm that answers keyword relaxation queries using a single-pass-traversal over an inverted lists index of grams. In Section 5 we discuss various solutions for cleaning keyword queries when the data is inside the database. We focus on how to relax keywords and retrieve relevant records. In Section 6 we look at additional optimizations for the query cleaning problem. We show how the keyword relaxation step can be further optimized by grouping keywords and relaxing them together. We show how we can decide which keywords in the query we should relax. Finally, we also look at how to clean phrase queries. In Section 7 we report our experimental results on real datasets to evaluate the proposed techniques. We present the findings on the performance, limitations, and tradeoffs of the different approaches. The results presented in Section 4 were previously presented in [27].

1.1. Related Work

Exact keyword search inside databases has been well studied in the literature [1, 15, 6]. These systems first locate tuples that contain at least one of the keywords in the query. Then, they use a graph-based approach to find join paths between these tuples so that the joined tuples contain all the keywords in the query. In our setting we do not match the keywords exactly. After a keyword is relaxed we can leverage the existing exact keyword search techniques. We discuss these aspects in more detail in Section 5. When data can reside outside the DBMS, exact-keyword search has been heavily studied in text documents and Web search [3]. Compared with earlier work, our work focuses on query cleaning by relaxing keywords.

Several aspects of the more general query relaxation problem have been studied in the literature [1,25]. For instance, the authors in [19] propose a lattice-based framework and algorithms for relaxing relational join and selection queries on numeric attributes. A similar lattice structure can be used in our setting for relaxing keyword queries. More recently, in [25] the authors looked at the problem of keyword-query cleaning from the perspective of grouping keywords into segments. The authors focus on how an unclean query can be reformulated, without discussing how new queries are then efficiently answered. For each query keyword, their system proposes alternatives and then all the possible keyword combinations are considered. A scoring function decides which combination of keywords and segments of keywords should replace the initial query. Considering all keyword combinations proves to be a step which is time consuming. In our study, we look at the entire pipeline and propose optimizations for the entire system as a whole. Our keyword-relaxation algorithms complement the problem of query cleaning by providing efficient implementations for the relaxation operator.

There have been many studies on ranking queries on relational tables after the pioneering studies by Fagin [10, 11]. See [16] for a recent survey. Our keyword relaxation approach is different from the traditional setting of top-k queries in the following aspects. In the traditional setting, a record has multiple attributes, and each attribute has a list of record ids sorted based on their similarity to a query on that attribute. An aggregation function is used to combine these similarities (possibly with weights) to compute an overall score for each record, and we want to find the k best records. In our setting, we do not have multiple rank-able attributes. Instead, a keyword has multiple grams, and we have multiple lists of the grams in the query keyword. Each list only includes ids of those keywords with the corresponding gram. The similarity of a keyword to the query keyword is closely related to the number of occurrences of the keyword on these lists. In addition, each keyword has a single weight, and we use its similarity to the query keyword and the weight to compute its overall score.

Several recent papers have focused on approximate keyword selection (or range search) [13, 23]. They assume a similarity threshold is given. Our ranking algorithms in Section 4 do not assume this threshold. They also consider the weights of keywords. There are algorithms for the problem of approximate keyword joins based on various similarity functions [2, 5, 7, 9, 12, 26], especially in the context of record linkage [20].
2. Preliminaries

**Record Collection**: Let $D$ be a bag of records, such as all the records from one table in a database. We denote by $d$ a particular record in the table.

**Keyword Collection**: Let $S$ be a set of keywords that contains all the unique keywords that appear in all the records $d$ in $D$, in the attributes of interest to the user. Each keyword $s$ in $S$ has a weight $w(s)$ associated with it, which indicates the relative importance of this keyword. The weight could be the importance of a keyword in a particular application domain. For example, it can be computed using the inverse-document frequency (IDF) of the keyword.

For our running example, a keyword collection $S$ is formed by all the keywords that appear in the *Description* attribute: ("popular", "givenchy", "fragrance", ...).

**Top-k Similar Keywords**: Given a keyword collection $S$, a keyword $r$, a similarity function $\theta$, an aggregation scoring function $F$ (that assigns a score to a given keyword), and an integer $k$, the top-$k$ similar keywords to $r$ are the best $k$ keywords in $S$ in terms of overall score to $r$. We denote by $T$ the set of keywords $(t_1, t_2, \ldots, t_k)$ most similar to $r$.

For a keyword $s$, we use "[s]" to denote the length of $s$, "s[i]" to denote the $i$-th character of $s$ (starting from 1), and "s[i, j]" to denote the substring from its $i$-th character to its $j$-th character. A positional $q$-gram of $s$ is a pair $(i, g)$, where $g$ is the substring of length $q$ starting at the $i$-th character of $s$, i.e., $g = s[i, i + q - 1]$. The set of $q$-grams of $s$, denoted by $G(s, q)$, or simply $G(s)$ when the $q$ value is clear in the context, is obtained by sliding a window of length $q$ over the characters of $s$. For instance, suppose $q = 2$, and $s = "john"$, then $G(s, q) = \{jo, oh, hn\}$. The number of $q$-grams of the keyword $s$ is $|s| - q + 1$.

Given two keywords, the similarity function $\theta$ computes a similarity value $\theta(s_1, s_2)$ between two keywords $s_1$ and $s_2$. Various similarity functions can be used. Commonly used similarity functions include edit distance, Cosine similarity, and Jaccard similarity. For instance, the Jaccard similarity of two keywords $s_1$ and $s_2$ based on $q$-grams is $jaccard(s_1, s_2) = \frac{|G(s_1, q) \cap G(s_2, q)|}{|G(s_1, q) \cup G(s_2, q)|}$.

The query can be relaxed as soon as the query answering engine realizes that the answer is going to be empty. The engine might choose to relax one or more of the keywords. It might start with relaxing one of the keywords, then it might relax two keywords, and so on, eventually relaxing all of the keywords. All these cases form a lattice structure where the top node represents not relaxing any keyword, while the bottom node represents relaxing all the keywords. We call this lattice structure the relaxation lattice. Figure 2 shows the relaxation lattice for keyword query $R$ ("givenchy", "t-shirt") posed over the Product table. In the figure "\(=\)" before the keyword means that the keyword is matched exactly, while "\(!=\)" means that the keyword is relaxed. A similar method for relaxing join and selection queries on numeric attributes has been used in [19]. In this paper we focus on the case at the bottom node of the lattice where all the keywords are relaxed.

**Relaxed Query Results**: We are given a keyword query $R$ as a sequence of keywords $(r_1, r_2, \ldots, r_k)$ and a global score aggregation function $E$. For a keyword $r_i$, $T_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,h}\}$

3. Keyword Query Relaxation Outside DBMS

In this section we present a query engine particularly designed for answering keyword queries assuming that the data is stored in a custom format or can be exported for indexing outside the relational engine.

3.1. Query Processing Operators

We define a set of logical operators that we will then realize for answering keyword queries:

- **Keyword Relaxation**: Given a keyword, the keyword relaxation (\(\approx\)) operator will return a set of similar keywords using one of the top-$k$ relaxation methods that we are going to propose.

- **Record Selection**: Given a keyword and a collection of records, the record selection (\(\sigma\)) operator will return the records in the collection that contain the given keyword exactly.

- **Record Intersection**: Given two or more collections of records, the record intersection (\(\cap\)) operator will return the set intersection of the input collections.

- **Record Union**: Given two or more collections of records, the record union (\(\cup\)) operator will return the set union of the input collections.

For our query example, Figure 3(a) and Figure 3(b) show two possible plans to answer this query. For Plan 1, first each keyword is relaxed using the \(\approx\) operator. For each of the resulting keywords, the records containing the keyword are selected. For \(\sigma(\text{"t-shirt"})\) record 11 is selected, for \(\sigma(\text{"t-shirt"})\) record 14 is selected, and so on. Next, the union of the records corresponding to each of the query keywords

![Figure 2: Relaxation lattice for query R = ("givenchy", "t-shirt")](image-url)

denotes the set of keywords similar to $r_i$, where $i = [1, h]$. Let $T$ be the set $\{T_1, T_2, \ldots, T_h\}$. The results of a relaxed query are the records $d$ from $D$ where each record contains at least one keyword from each set $T_i$, where $i = [1, h]$, sorted decreasing by their global E score with respect to the keyword query $R$.

The global scoring function $E$ computes $E(R, d)$ as an overall score of the record $d$ to the keyword query $R$. It takes into consideration the importance of each keyword $w$ in the record $d$ and the similarity between $w$ and $r_i$. Additionally, the scoring function could also consider the attribute of the record in which the keyword appears.

For example, we can use the sum of TF/IDF scores of each $t_{i,j}$ in $d$ scaled by the $F$ score between $t_{i,j}$ and $r_i$ as the global score of a record:

$$E(R, d) = \sum_{t_{i,j} \in d} tf(t_{i,j}, d) \cdot idf(t_{i,j}) \cdot F(r_i, t_{i,j}).$$

(1)
is computed. For query keyword “t-shirt” the record union is 11, 12, 13, 14, 15, while for query keyword “givanchy” the record union is 10, 12, 15. Finally, the record intersection is computed and the final results is 12, 15, which contains both keywords approximately. In Plan 2, the intersection operation is performed before the union operation.

Each of the logical operators can be physically implemented in several different ways outside the DBMS:

- **Keyword Relaxation (≈)**: In Section 4 we discuss a few possible ways to implement this operator.
- **Record Selection (σ)**: The following are two ways to implement this operator:
  - Scan: We can scan the entire record collection and output the records that contain the given keyword.
  - Index Lookup: If an index from keyword to record is available, we can use the index to identify the records that contain the given keyword. One possible index is an inverted-list structure from keywords to records.
- **Record Intersection (∩)**: During the index construction and at any time during the query execution, we keep the list of identified records sorted by id. In this way the intersection operation can be implemented very efficiently. If we need to intersect more than two record collections, we can use a heap-based intersection algorithm. See [4] for an excellent survey of sorted set intersection algorithms.
- **Record Union (∪)**: Similar to the records intersection operation, we can implement the records union operation very efficiently if the list of identified records is sorted by id.

### 3.2. Query Optimization

Using the predicates’ selectivity and the run-time cost of an operator, we can estimate the total cost of an operator. For estimating the selectivity of relaxed keywords we can use techniques like [18, 24, 22, 14]. The operator run-time cost can be computed experimentally using a relevant query workload (e.g., log history from a search engine).

Following the relational database optimization techniques, we can generate a set of plans and choose a “good” one to execute. If the number of keywords is not large, the number of possible plans is small. Additionally, we can build the “good” plan bottom up using a greedy approach where at each step we choose the least expensive operator.

We experimentally evaluated the cost of each operator. We used a naive implementation for the keyword relaxation operation (≈). For relaxing a keyword we computed the top-3 similar keywords by scanning the keyword collection. For computing the score we use the sum of the similarity (computed using the edit distance normalized by length) and the weight as the E score. We used 100k emails from the Enron email dataset, which had around 240k keywords. We used a query workload of 100 queries where each query had 3 randomly picked keywords. We ran the workload 5 times and averaged the time needed to execute one query. We used a plan similar to the plan in Figure 3(b) as it is a “good” plan because the intersections are pushed down in the plan and many records get discarded early. For implementing the record selection (σ) operator we used a memory inverted-list structure from word to record. On average a query took 484.53ms to execute. Figure 4 shows the absolute and relative total time needed by each operator in the plan. For example, the Keyword relaxation time is the total time needed for relaxing all the 3 keywords. This experiment shows that the keyword relaxation operator is the most expensive operator in the plan and further optimization of this operator is needed. In the next section we study efficient ways of implementing this operator.

### 4. Efficient Keyword Relaxation

A keyword can be relaxed by specifying a threshold on score and retrieving all the keywords that fall within the score threshold (range search) or by retrieving the top-k keywords with respect to their score to the given keyword (top-k search). The first approach has the disadvantage that a threshold needs to be specified. In this section we briefly review the available algorithms for range search and propose algorithms for top-k search.

The scoring function \( F \) computes \( F(r, s) \) as an overall score of the keyword \( s \) to the keyword \( r \) in terms of its weight and similarity to \( r \).

For example, we can use Jaccard as the similarity function, and a linear combination of the similarity and the weight of the keyword as the final score of the keyword:

\[
F(\theta(r, s), w(s)) = \alpha \cdot \theta(r, s) + \beta \cdot w(s).
\]

There are many different index structures that can be used to relax keywords. In this paper, we use a 2-gram inverted-list index, built on the keyword collection. For each gram in \( S \), we have an inverted list of ids of keywords containing this gram. For instance, Figure 5(a) shows four keywords with their weights and Figure 5(b) shows the corresponding inverted lists of 2-grams. The ids on each list are sorted in ascending order. Without loss of generality, we assume that the ascending order of the keyword ids is identical to the descending order of their weights. If not, we can map each keyword id to a new id (a
one-to-one mapping), so that the new keyword ids have this property.

<table>
<thead>
<tr>
<th>ID</th>
<th>Keyword</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ab</td>
<td>0.80</td>
</tr>
<tr>
<td>2</td>
<td>ccd</td>
<td>0.70</td>
</tr>
<tr>
<td>3</td>
<td>cd</td>
<td>0.60</td>
</tr>
<tr>
<td>4</td>
<td>abcd</td>
<td>0.50</td>
</tr>
<tr>
<td>5</td>
<td>bcc</td>
<td>0.40</td>
</tr>
</tbody>
</table>

![Example of gram inverted-list index.](image)

**Range Search.** Several recent papers have focused on approximate-keyword range search algorithms [13, 23]. In [13], Hadjieleftheriou et al. presented a solution to the approximate-keyword range search problem by extending Fagin’s algorithms [10, 11]. Given a similarity threshold \( \tau \), they find the keywords in a keyword collection that have a similarity to the query \( \geq \tau \). Keywords are decomposed in grams and an IDF score is computed for each gram. The weight of a keyword is computed by aggregating the individual gram weights. In [23], Li et al. proposed a suite of algorithms for the approximate-keyword range search problem based on an efficient traversal of q-gram inverted lists. Given a similarity threshold \( \tau \), they find the keywords in a keyword collection that have a similarity to the query \( \geq \tau \). These techniques can be used to implement the keyword relaxation (\( \approx \)) operator.

In this section, we develop three algorithms for top-\( k \) approximate-keyword search. The first algorithm Iterative Range-Search answers a ranking query by using range selection queries. The algorithm iteratively issues queries with decreasing thresholds until \( k \) results are retrieved. The second algorithm Single-Pass Search uses the gram inverted list index (Figure 5(b)) and traverses the lists by using a heap. One important feature of the algorithm is that it can skip many elements that cannot be in the top-\( k \) answers. The final algorithm Two-Phase combines the first two algorithms. Figure 6 gives an overview of the three proposed algorithms.

![Overview of the top-\( k \) keyword-relaxation algorithms.](image)

### 4.1. Iterative Range-Search-Based Algorithm

We study how to answer a ranking query by answering (possibly multiple) range selection queries. Each selection query has a threshold on the similarity between the given keyword and a keyword in the collection. In this way, we can leverage existing approximate-keyword-selection techniques [26, 13, 23] without modifying their implementations. We develop an algorithm called “Iterative Range Search” (“IRS” for short). Algorithm 1 shows the pseudo-code of the algorithm. We start with an initial similarity threshold, \( \tau \), which could be a fixed value (e.g., 0.9 for Jaccard similarity) or a value computed based on the query (line 5). The algorithm has two steps.

**Algorithm 1 : IRS Algorithm for a top-\( k \) query**

1: Let \( k \) be the number of results requested;
2: Let \( w_{\text{max}} \) be the maximum weight of a keyword in the dataset;
3: Let \( f \geq 1 \) be a multiplication factor;
4: Let \( R \leftarrow \phi \) be the range-search-result set;
5: Let \( \tau \) be the initial similarity threshold;
6: 
   **[Step 1: Computing initial candidates]**
   
   while \( \text{size}(R) < f \cdot k \) do
   7:     \( R \leftarrow \text{ApproxRangeSearch}(\tau) \);
   8:     if \( \text{size}(R) < f \cdot k \) then
   9:         Decrease \( \tau \);
   10:     end while
   **[Step 2: Finalizing results]**
   11:     Compute scores for elements in \( R \) and keep the first \( k \);
   12:     Let \( \tau_1 \) be the minimum similarity for which \( \text{Score}(\tau_1, w_{\text{max}}) > \text{Score}(R[k]) \);
   13:     if \( \tau_1 < \tau \) then
   14:         \( R \leftarrow \text{ApproxRangeSearch}(\tau_1) \);
   15:     Compute scores for elements in \( R \) and keep the first \( k \);
   16:     end if
   17:     Return \( R[1..k] \);

**Step 1: Computing initial candidates** (lines 6 – 10). The goal of this step is to compute at least \( f \cdot k \) results, where \( f \geq 1 \) is a multiplication factor. We call a function “ApproxRangeSearch” to run an approximate-keyword-range-search algorithm of our choice, and find the keywords that pass the similarity threshold \( \tau \). Next, we decrease the similarity threshold, depending on the number of results we got. This step ends when we get at least \( f \cdot k \) results.

Suppose the given keyword \( r \) is \( \text{abcd} \), the similarity function is Jaccard, and the scoring function is Equation 2, where parameters \( \alpha \) and \( \beta \) are 1, and the \( k \) value is 2. Figure 7 shows a set of similar keywords to the given keyword. The figure also shows the weight of each keyword, its Jaccard similarity to the given keyword \( r \), and its aggregated score. Assume the multiplication factor \( f \) is 1. If we set the initial threshold \( \tau \) to 0.50, the approximate-range-search function returns the keyword with ids between 1 and 4. Because we got more than \( f \cdot k = 2 \) results, we can go directly to step 2. Otherwise, we decrease the similarity threshold \( \tau \), and rerun the approximate-range-search algorithm.

**Step 2: Finalizing results** (lines 11 – 16). We compute the score for each element computed in step 1, and keep the first \( k \) elements ordered by their scores. Next, we want to be certain that these \( k \) elements are indeed the best results. Consider one element \( e \) that was not seen before, and it has the maximum possible weight in the dataset. We compute how similar \( e \) needs
to be to the query in order to have a better score than the current \(k\)th element. We use this similarity as the new similarity threshold \(\tau_1\) (line 12). If \(\tau_1 < \tau\), we call the approximate-range-search function one more time, using \(\tau_1\) as the threshold (line 14). The more results we got from step 1, the better the \(k\)th element will be, and the tighter the similarity threshold \(\tau_1\) will be.

Continuing our example in Figure 7, the top-2 elements so far are 1 and 3. Assume the maximum weight in the dataset is 0.70. In order for an element \(e\) to have a score better than 0.96 (the score of element with id 3), it should satisfy the condition \(F(e) = \mu + 0.70 > 0.96\). Thus the Jaccard similarity \(\mu\) between \(e\) and the given keyword needs to be at least 0.26. In this way, our new similarity threshold \(\tau_1\) is 0.26. Because \(\tau_1 = 0.26\) was not covered by the previous approximate range searches, we run the approximate search function one more time using this threshold. During this run we find the element with id 5, which has a score 1.03. The final top-2 elements are 1 and 5.

Advantages and Limitations: The IRS algorithm has the advantage that it can utilize any of the existing algorithms for approximate-keyword range search. It is easy to implement as it uses the range-search algorithm as a black-box function.

One main limitation of the algorithm is that it needs to run multiple search queries, which may take a lot of time. In addition, it is not easy to choose a good initial similarity threshold \(\tau\) (line 5) and decrease \(\tau\) properly for the next query (line 9). There are recent studies on the problem of estimating the selectivity of SQL LIKE substring queries [8, 17, 21], and approximate keyword queries [24, 18, 22]. Notice that a key difference is that earlier studies estimate the number of answers to an approximate query that specifies a range. In our setting, we need to estimate the approximate range in order to find \(k\) elements. The methodologies used in those papers could be used in our setting.

### 4.2. Single-Pass Search Algorithm

We study how to answer a query by accessing the gram inverted lists only once. (We assume an answer should share at least one common gram with the given keyword.) A naive way to traverse the lists would be to loop over the lists, reading one element at a time from each list. During the traversal, we maintain the information about the visited elements (candidates) and a top-\(k\) buffer of the best \(k\) elements seen so far. For each candidate keyword, we compute a lower bound and an upper bound of its similarity to the given keyword. The bounds are computed based on the number of common grams between the candidate and the given keyword, i.e., the number of lists on which the candidate appears. The lower bound is computed based on the number of lists on which the candidate appeared so far. The upper bound is the lower bound plus the number of lists on which the candidate could appear later in the traversal. The algorithm stops when the top-\(k\) candidate set cannot be improved.

A better way of traversing the lists is to use a heap. The algorithm is called “Single-Pass Search” (“SPS” for short). It traverses the lists in a sorted order using a heap of the current top elements of the lists. This traversal order has two advantages: we do not have the overhead of maintaining the candidate set, and we have more chances to skip elements. Algorithm 2 shows the pseudo-code of the algorithm.

#### Algorithm 2: SPS Algorithm for a top-\(k\) query

1. Let \(n\) be the number of grams in the query;
2. Let \(l[0..n-1]\) be the lists of ids for the query grams;
3. Let \(g \leftarrow 1\) be the frequency threshold;
4. Insert the top element on each list to a heap, \(H\);
5. \(\text{Topk} \leftarrow \emptyset\);
6. while \(H\) is not empty do
5.1 Let \(T\) be the top element on \(H\);
5.2 Pop from \(H\) those elements equal to \(T\);
5.3 Let \(p\) be the number of popped elements;
5.4 if \(p \geq g\) then
5.4.1 if \(\text{Score}(T) > \text{Score}(\text{Topk})\) then
5.4.2 Insert \(T\) into \(\text{Topk}\) and pop the last one;
5.4.3 Recompute threshold \(g\);
5.4.4 if \(g > n\) then break;
5.5 end if
5.6 end while
5.7 Push next element (if any) of each popped list to \(H\);
5.8 else
5.9 Pop additional \(g - p - 1\) elements from \(H\);
5.10 Let \(T'\) be the current top element on \(H\);
5.11 for each of the \(g - 1\) popped lists do
5.11.1 Locate its smallest element \(E \geq T'\) (if any);
5.11.2 Push \(E\) to \(H\);
5.12 end for
5.13 end if
5.14 end while
5.15 Return the elements in \(\text{Topk}\);

We first initialize the frequency threshold \(g\) (line 3). The algorithm maintains a cursor for each list pointing to the current element. The cursor is initially set to the first element on the list. We maintain a heap, \(H\), of the elements pointed by the cursors on the lists (line 4). We pop an element from the heap, process it, and push another element from the same list to the heap. We traverse the lists by pushing and popping elements to and from the heap (lines 6 – 26). If a keyword id appears on multiple lists, the heap has multiple copies of that id. Whenever we pop one element from the heap, we pop all its copies (line 8). In this way we know the total frequency, \(p\), of the keyword id on the lists (line 9). Then, the element is processed, and retained in the top-\(k\) buffer, or discarded. There is no need for maintaining a candidate set (lines 10 – 25).

The first \(k\) elements visited during the traversal become the
top-k candidates, and are added to the top-k buffer (line 12). Next, we compute what new frequency threshold \( g \) a new element needs to have in order to have a better score than the \( k^{th} \) element in the buffer (line 13). We describe how to compute the frequency threshold \( g \) later in this section. If a new element has a frequency \( p \) less than the frequency threshold \( g \), we pop additional \( g - p - 1 \) elements from the heap, and move the cursor on each popped list to the first element greater than or equal to the current top on the heap (lines 19–24). (Similar intuition has been used in the MergeSkip algorithm proposed in [23].) The algorithm stops when reaching the last element on each list, or when the frequency threshold \( g \) is greater than the number of inverted lists \( n \).

We use the example in Figure 8 to show the intuition behind the skipping step. For simplicity, we use the number of common grams as similarity between two keywords and a scoring function based only on the similarity. The figure shows a snapshot of the element-id lists after element 10 has been read from lists 1 and 2, and retained as the top-1 candidate. Currently the frequency threshold \( g \) is 3, thus an element needs to appear at least 3 times in order to be better than the current top-1. Next, the algorithm reads element 20 from the heap. As its frequency \( p \) is 1, the algorithm reads one more element from the heap. After that, the current head of the heap is 40, and the algorithm jumps on lists 1 and 2 to the element 40. In this way the algorithm skips many elements on lists 1 and 2. All the skipped elements could appear at most 2 times on the lists, while the frequency threshold is 3. Thus they cannot be the top-1 answer.

**Computing Frequency Threshold:**

Given the score of the \( k^{th} \) candidate in the buffer, \( y_k \), we need to compute a frequency threshold, \( g \), that an element needs to have in order to have a better score. First, the largest weight of the last seen weights on the lists is the maximum weight that an unseen element could have. We denote it by \( \eta_{max} \). Next, using the definition of the scoring function, \( y_k \) and \( \eta_{max} \), we derive the minimum similarity an element needs to have in order to be better than the current \( k^{th} \) element. We denote that by \( \tau_{min} \). For example, if the scoring function is Equation 2, then \( \tau_{min} \) can be computed as:

\[
\tau_{min} = \frac{\gamma_k - \beta \cdot \eta_{max}}{\eta_{r}}.
\]

Given \( \tau_{min} \), we can compute \( g \) using a formula derived from the definition of the similarity function. Formulas for deriving the frequency threshold for a given similarity have been defined in [23] for the common similarity functions. For example, if the similarity function is Jaccard, \( g \) can be computed as follows:

\[
g = \max(\tau_{min} \cdot r, \frac{g_r + g_{min}}{1 + 1/\tau_{min}}),
\]

where \( g_r \) is the number of grams in the query, and \( g_{min} \) is the minimum number of grams of a keyword in the collection.

**Improvement by Separating Lists**

The algorithm can be improved by partitioning the lists into a set of long lists and a set of short lists. It treats the long lists separately, since these lists could take a lot of time to traverse. The algorithm only searches in them for elements found on those short lists. A similar idea of treating the long lists differently was proposed in algorithms in [26, 23]. We use a heap to traverse the short lists. Whenever an element in the short lists has a frequency at least \( g - n_{long} \), where \( n_{long} \) is the number of long lists, we can search for this element in each of the long lists by doing a binary search. We can use a formula derived in [23] for deciding \( n_{long} \).

### 4.3. Two-Phase Algorithm

We can combine the IRS algorithm and the SPS algorithm. This new algorithm is called “two-phase” algorithm (2PH). In the first phase, we execute a single range search with a tight similarity threshold, \( \tau \). In the second phase, we run the SPS algorithm, but the initial bound on the number of common grams (\( g \) in Algorithm 2) is computed based on the score of the records retrieved in phase 1. The algorithm is based on the following two observations. (1) Retrieving the records very similar to the query could be done efficiently using existing range-search algorithms. (2) The SPS algorithm is efficient since it can skip many elements. Still, a low initial frequency threshold makes the algorithm process a lot of elements at the beginning. The initial top-\( k \) candidates computed in phase 1 could give as a higher initial frequency threshold. Moreover, the traversal might stop earlier since the records very similar to the query have already been considered.

**Utilizing a User Similarity Threshold:**

In some cases the user might provide an initial similarity threshold so that we will not consider keywords that are very different from the given keyword, no matter how important they are. The algorithms presented so far can benefit from such a threshold and become more efficient. Take the SPS algorithm as an example. From the user similarity threshold, we can compute a bound on the number of common grams using the formulas in [23]. We use the computed bound as the initial frequency threshold in the SPS algorithm, and the rest of the algorithm remains the same. In this way, the tighter initial frequency bound can make the algorithm more efficient since it can skip more elements.

### 5. Keyword Query Relaxation Inside DBMS

In this section we present techniques for answering keyword queries assuming that the data is stored in a DBMS. The ability of relaxing keyword queries inside the DBMS is important for applications where the records cannot be exported outside the DBMS due to security or privacy issues, or the application level
changes necessary for maintaining the records updated are too high. The statistical information or partial information about the data can be stored outside the database and a thin custom layer can be used to complement the query processing done by the DBMS. The code layer can take the form of a User Defined Function (UDF), which are available in most of the commercial or open-source relational engines. We first present how to enable the keyword relaxation operator inside the DBMS system and then the records retrieval operators.

The techniques presented in this section can be adapted for the case where data is stored across multiple tables. Various techniques have been proposed in the literature [1, 6, 15] which deal with answering exact keyword queries across multiple tables. A result to such a query is usually formed by joining records from several tables. After the keywords in the original query are relaxed, the relaxed query can be viewed as a conjunction of keyword disjunctions. Each relaxed keyword produces a disjunction of similar keywords. Next, the relaxed query can be processed using one of the exact keyword search techniques across multiple tables. If the exact keyword search technique does not work for disjunctive queries, we have to split the query into multiple conjunctive queries and compute the union of the results.

5.1. Efficient Top-k Keyword Relaxation

Given a keyword, the keyword relaxation operator returns a set of keywords similar to it. For each record in the collection, the values of all the attributes of interest to the user have been tokenized and all the unique keywords are stored in an auxiliary relational table Keywords inside the DBMS. A second SQL table is created to mimic the keyword inverted list structure Keywords-to-Records, by storing which keyword appears in which record. For the example in Figure 1 the corresponding Keywords table is in Figure 9(a), while the Keywords-to-Records table is in Figure 9(b). The Keywords table can also store the weight associated with each keyword.

<table>
<thead>
<tr>
<th>kid</th>
<th>Keyword</th>
<th>Wht</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>popular</td>
<td>0.70</td>
</tr>
<tr>
<td>2</td>
<td>givenchy</td>
<td>0.80</td>
</tr>
<tr>
<td>3</td>
<td>fragrance</td>
<td>0.60</td>
</tr>
<tr>
<td>4</td>
<td>adidas</td>
<td>0.50</td>
</tr>
<tr>
<td>5</td>
<td>t-shirt</td>
<td>0.40</td>
</tr>
<tr>
<td>6</td>
<td>elegant</td>
<td>0.60</td>
</tr>
<tr>
<td>7</td>
<td>shirt</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kid</td>
<td>rid</td>
<td>Gram</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>br</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>ro</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>ow</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>wn</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>co</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>ol</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>lo</td>
</tr>
</tbody>
</table>

Figure 9: Example of Keywords, Keywords-to-Records, and Grams tables.

Range Search. Inside the DBMS, keywords that are within a certain similarity range from a given keyword are found using a UDF that computes the score between two keywords. A naive approach is to use the UDF in a SQL selection query that scans the Keywords table and selects the keywords that fall within the similarity range. A better approach has been proposed in [12]. The authors proposed a solution for finding approximately matching strings between two relational tables (or a self-join on a single table). The technique uses an additional SQL table Grams that stores the q-grams that appear in each keyword. Using the fact that two similar keywords need to share a number of common grams, the similar keywords can be found using a join SQL query between the grams and the Keywords table. For the Keywords table in Figure 9(a), Figure 9(c) shows the corresponding Gram table (for q = 2). Additionally, grams can be hashed to integer values for better performance.

We present four alternatives of finding the top-k approximate-keywords to a given keyword. The order in which the approaches are presented corresponds to the amount of changes necessary to the database infrastructure, with the first approach requiring the least amount of changes. The table in Figure 10 summarizes the four approaches by showing which part of the index and code is stored inside the DBMS and which part is stored outside the DBMS. Please note that in all the cases the original records are not exported outside the DBMS.

<table>
<thead>
<tr>
<th>Method</th>
<th>Inside DBMS</th>
<th>Outside DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Keywords</td>
<td>UDF</td>
</tr>
<tr>
<td>Gram Filter</td>
<td>Keywords, Grams</td>
<td>UDF</td>
</tr>
<tr>
<td>Iterative</td>
<td>Keywords, Grams, Stored Procedure</td>
<td>UDF</td>
</tr>
<tr>
<td>External</td>
<td>-</td>
<td>Keywords, Grams</td>
</tr>
</tbody>
</table>

Figure 10: Summary of the Top-k keyword relaxation approaches.

1) Scan. Using the UDF that computes the F score between two keywords we can retrieve the top-k similar keywords in the following way. We run an SQL select query that scans the Keywords table, orders the keywords in decreasing order of their score, and returns the first k keywords using a limit-like operator. For example, for the MySQL database engine, the SQL query for retrieving the top-3 similar keywords to keyword “givenchy” is in Figure 11(SQL1). “Score” is a UDF that computes the similarity score between two keywords. This approach does not require an extra grams table. The performance bottleneck of this approach is that it has to scan the entire keywords table and make a UDF call for each keyword in the table.

2) Gram Filter. A better way than scanning the entire keywords table is to use the Grams table. In order to use the Grams table, the given keyword needs to be decomposed into grams. We decompose the keyword into grams and store the grams in a temporary SQL table Keyword_Grams. The table has a single column, the gram. This can be achieved using a combination of an UDF and a stored procedure. The SQL query becomes a join query between the Keywords table, the Grams table, and the Keyword_Grams table. The query assumes that the user is only interested in keywords that share at least one gram in common with the given keywords. The SQL query for retrieving the top-3 similar keywords “givenchy” is in Figure 11(SQL2). The advantage of this approach is that we make UDF calls only for a subset of keywords. As a drawback, the solution needs the grams table. The bottleneck of this approach is the join opera-
5.2. Record Retrieval

We can leverage the DBMS engine to retrieve the records that contain the relaxed keywords. We focus on two approaches. The first one uses the Keywords_to_Records relational table, while the second one uses the Full-Text search capabilities of the DBMS engine. The table in Figure 12 summarizes the two approaches by specifying what additional data structure is needed for each of them. For each of the query keywords, we store the similar keywords in a temporary table Keyword_KID. If the query has three keywords, we have three temporary tables.

1) **Join.** The Keywords_to_Records table stores which keyword appears in which record. We use a join query between the Keywords_KID tables and the Keywords_to_Records table. The SQL query for a 2 keyword query is in Figure 11(SQL3).

The relational table Keyword_KID stores the keyword ids of the keywords similar to the first keyword in the query, while Keyword_KID2 stores the keyword ids for the second keyword in the query. Finally, for each selected record, we can compute the global score and order the records in decreasing order by score. For example, if we use a TF/IDF-based score formula, we need to append a new column to the Keywords table with the IDF score for each keyword, a new column to the Keywords_to_Records table with the TF score for each keyword in each record, and new columns to the Keyword_KID tables with the TF score for the keyword and the IDF score. The complete SQL query for a 2 keyword query is listed in Figure 11(SQL4).

The main advantage of this approach is that it can be implemented on any relational DBMS. It does not rely on any auxiliary code or special feature of the DBMS. The bottleneck of this approach is the join operation. For each additional keyword in the query we need to add 2 more tables to the join query. So, the number of keywords in the query greatly affects the performance of this approach.

2) **Full-Text** We can leverage the full-text index feature of the DBMS to locate the records that contain the relaxed keywords. We first build a full-text index on the attributes of interest to the user from the records table. Next we build a Boolean full-text query. For each of the keywords in the original query, a matching record needs to contain at least one of the similar keywords. For example, for the product description table in
Figure 1 (stored in the relational table Products), assume that the query keyword is \( R = (\text{givanchy}, \text{t-shirt}) \). The similar keywords for keyword “givanchy” are “givency” and “givanch”, while the similar keywords for “t-shirt” are “t-shirt” and “tshirts”. Then the MySQL full-text SQL query is in Figure 11(SQL5).

The above SQL query can be prepared inside the DBMS, using SQL statements, SQL string concatenation methods, and SQL prepared statements. The main advantage of this approach is the use of a customized index, instead of table joins, for locating the records. The approach requires a full-text index feature to be implemented in the DBMS. Most common DBMS systems provide such a feature, but the feature is not standardized and has its own particularities in each system. One important drawback of the full-text index approach is that we cannot easily compute the score. In order to compute the score we need to have another pass over the records in the result set and for each record identify which of the relaxed keywords it contains.

### 6. Additional Optimizations

In this section we discuss additional optimizations which can be applied both outside and inside the database.

#### 6.1. Keyword Grouping

If we use the inverted lists of grams to find similar keywords for a given keyword and if the query has more than one keyword, then during the keyword relaxation phase we might traverse the same inverted lists multiple time. For example, assume the keyword query is \( R = (\text{mike}, \text{pike}) \) and that the gram length is 2. For the first keyword we have to traverse the keyword id lists corresponding to grams “mi”, “ik”, and “ke”. For the second keyword we have to traverse the lists corresponding to “pi”, “ik”, and “ke”. So, we end up traversing the lists for the “ik” and “ke” grams twice.

We optimize this process by grouping the keywords and finding the similar keywords for two or more keywords at the same time. In order to achieve this we have to traverse the lists of keywords ids corresponding to the union of grams of the grouped keywords. For instance, for the SPS algorithm, for each keyword in the group we have a minimum frequency threshold \( g \). An element is considered a candidate for the top-\( k \) buffer of any of the keywords in the group if its frequency is greater than the smallest of the minimum frequencies. On the other hand, due to the fact that we might have more lists than grams in one of the keywords, we might seem to encounter candidates that have more grams in common with the query keyword than actual grams in the query keyword. For example, if the keyword “mikepick” is in the keyword collection, it appears in all of the 4 lists of the gram union set. So, it seems that this keyword has 4 grams in common with each of the given keywords, when in fact each of the given keywords has only 3 grams. This inconsistency is fixed when the real similarity is computed. Additionally, we add another filtering condition which checks that a candidate does not “share” too many grams with the given keywords.

For the outside the DBMS setting, the gram-list traversal algorithms can be adapted to compute the similar keywords for a group of keywords. Take for example the SPS algorithm in Algorithm 2. Before line 6 we need to initialize the upper bound on the number of common grams a candidate can share with each of the keywords in the group. Line 11 needs to be changed to loop over the keywords in the group, and for each keyword check if the candidate does not exceed the upper bound on the number of common grams. Finally, we check the candidate if it can be inserted in the current keyword’s top-\( k \) buffer. If the candidate is inserted in the top-\( k \) buffer, the minimum threshold for the current keyword might increase and so might the minimum overall threshold \( g \).

For the inside the DBMS setting, the gram filtering approach can also benefit from grouping the keywords. Essentially, the SQL query looks for grams of multiple keywords, leaving the filtering of false positives to the final step that computes the real similarity. A keyword is selected if it shares at least one gram with any of the keywords in the group. Afterwards, the outputted keywords need to be separated and assigned to one or more of the original keywords.

#### 6.2. Keyword Relaxation Lattice

There are many situations in real applications where some of the keywords are mistyped neither in the query, nor in the data. Because of that it is important for the system to have the option of relaxing some of the keywords. Consider for example the query keyword \( R = (\text{givanchy}, \text{t-shirt}) \). Figure 2 shows different ways in which the keywords can be relaxed, forming the relaxation lattice. Using the relaxation lattice the system starts at the top node in the lattice and executes the query by matching all the keywords exactly. If the results set is not empty it stops. Otherwise, it needs to go lower in the lattice. If counts on how many records are returned for each keyword are available, we relax the keywords that do not appear in any records. Even in the case in which there are matching records for all the keywords, there might be no records that contain both keywords and so one of the keywords or both need to be relaxed. We can take this a step further and decide to drop a keyword from the query. In this way we return the records that have all the other keywords besides the dropped one.

For the outside the DBMS setting, the relaxation lattice can be integrated in the query execution module, and as soon as we realize that the final result set will be empty, we can advance to the next level in the lattice. In a similar fashion, the relaxation lattice can be integrated for the inside the DBMS setting in the stored procedure that relaxes the keywords and retrieves the results or in a thin layer outside the database.

#### 6.3. Answering Phrase Queries

Often keyword queries can contain phrases implicitly or explicitly. We can modify our system to handle phrase queries approximately. A common way to handle exact phrase queries is to use a nextword index [28]. The nextword index is a two level index structure. At the first level we store the first word of a phrase, firstword. For each firstword there is another list
of words, nextword, that appear after it, forming phrases of two words. For each nextword, there is a list of record ids that contain the firstword-nextword phrase. Figure 13 shows an example of a nextword index.

```
FirstWord  NextWord  Record IDs
...        page      15, 21, ...
...        paul      10, 31, ...
...        mccartney 17, 24, ...
...        marage...
```

Figure 13: Nextword index example.

We can leverage the next word index for answering phrase queries approximately. If the query has only two keywords, we relax the first keyword in the query by finding similar keywords in the firstword list. Afterwards, for each of the matching firstword keywords, we relax the second keyword in the query using the firstword’s nextword list. If the query has more than two keywords, \( n > 2 \), we can apply the same algorithm by assuming that the phrase can start at any of the first \( n-1 \) keywords. Consider for example the following keyword query \( R = (\text{“paul”,”marage”}) \). If we treat this query as a phrase query, we first relax “paul”, using the firstword collection of keywords. Among the similar keywords we find “paul” and so we use the nextword keyword collection for “paul” to relax the second keyword in the query, “marage”. One possible answer to the query is the phrase “paul marage”.

For the outside the DBMS setting, the nextword index can be integrated into the system as a secondary index. The query plan generator will generate plans that treat the keyword queries as phrase queries and use the nextword index. The query optimizer will choose whether to use a phrase query plan or not. For the inside the DBMS setting, the next word index can be integrated as another relational table. The table will contain two columns of keyword ids, one column for the firstword and the other column for the nextword. The plan choosing logic can be specified in a stored procedure or in a thin layer of code outside the DBMS.

7. Experimental Evaluation

In this section we present our experimental evaluation of the proposed algorithms for keyword-query cleaning. We used two real datasets.

- **IMDB Actor Names**: It consisted of actor names, and the numbers of movies they played in. The data was downloaded from the IMDB website\(^1\). There were 1.2 million names, with the average length of 15. We log-normalized the number of movies an actor played in and used it as the weight of the actor.

- **WEB Corpus Word Grams**: It came from the LDC Corpus set at the University of Pennsylvania\(^2\). This dataset, contributed by Google Inc., contained sequences of English words and their observed frequency counts on the Web. The raw data was around 30GB. We randomly chose 2.4 million sequences, with the average length of 20. We log-normalized the frequency of the sequence and used it as its weight.

We evaluated the complete keyword query relaxation system on the Enron email dataset. The dataset consisted of 0.5 million emails from the employees of the Enron corporation\(^3\). We tokenized the content and extracted a list of 630 thousands words.

We used the Jaccard similarity and the normalized edit similarity. Each similarity is in the range \([0, 1]\). Each weight is also in the range \([0, 1]\), where 1 is the highest weight. Figure 14 shows the distributions of the percentage of elements that share a weight. We used \(q = 3\) for the gram length. For each dataset, we constructed 100 queries by randomly selecting strings from the dataset. All the algorithms were implemented using C++ (GNU compiler) and run on a Intel 2.40GHz PC with 2GB main memory, running a Ubuntu Linux operating system.

7.1. Efficiency of Keyword Query Relaxation Outside DBMS

We first evaluate the performance of the proposed techniques for the outside the DBMS setting.

7.1.1. Efficiency of Keyword Relaxation Algorithms

We evaluated the performance of the keyword relaxation algorithms for top-10 queries using the IMDB and Web Corpus datasets. We considered the scoring function in Equation 2, with \( \alpha = 1 \) and \( \beta = 1 \). We ran each query 5 times and used its average running time in order to compute accurate performance numbers.

Figure 15(a) shows the average query time of the SPS and 2PH top-\(k\) keyword relaxation algorithms on the IMDB dataset using the Jaccard similarity. The IRS algorithm performed very poorly and we did not plot its running time. Even for the optimal initial threshold, the IRS algorithm needed around 5 seconds to compute the top-10 results for 1.2 million entries. This result was the best performance of the algorithm as we used the optimal initial similarity threshold and ran a single range search. The reason for this poor performance is that the range search retrieves too many candidates and it takes a lot of time to

\(^1\)http://www.imdb.com/interfaces
\(^2\)http://www.ldc.upenn.edu/Catalog, number LDC2006T13
\(^3\)http://www.cs.cmu.edu/~enron
verify them. The performance of the algorithm was even worse if we did not use the optimal initial threshold. The SPS and 2PH algorithms have similar performance. For the 2PH algorithm we heuristically computed the initial threshold as being equal to the number of grams in the query. They needed around 5 ms to answer a top-10 query for 1.2 million entries. The time increased slowly as the dataset size increased.

Figure 15: Average running time for top-10 keyword relaxation (IMDB).

Figure 15(b) shows the results for the same setting but using the normalized edit similarity. The relative performance order of the algorithms is preserved, but all of them needed more time to compute the top-10 results than for the Jaccard case. This behavior is due to the fact that for Jaccard, from the number of common grams between two strings we could compute their exact similarity. For the normalized edit similarity, we computed bounds on the similarity, since computing the exact similarity is expensive. We observed similar results on the WEB Corpus dataset.

7.1.2. Benefits of Skipping Elements

In this experiment, we analyzed how the skipping operation affects the running time of the single-pass search algorithms. We ran the SPS algorithm for various subsets of the IMDB dataset with different sizes, with the skipping feature disabled. Figure 16(a) shows the average running time for the SPS algorithm with and without the skipping feature, using the Jaccard similarity. In the figure, the asterisk (*) near an algorithm name means that the algorithm had the skipping feature disabled. We can see that the skipping operation improves the performance because a smaller number of records need to be processed. For example, the SPS algorithm needed around 40 ms if no skipping was performed, while it only needed around 5 ms with skipping. Figure 16(b) shows the results when we used the normalized edit similarity. The skipping operation helps to improve the performance of the algorithms for this function as well.

7.1.3. Effect of the Initial Threshold for the 2PH Algorithm

We evaluated how the initial threshold affects the performance of the 2PH algorithm. We randomly generated 3 top-10 queries and ran the 2PH algorithm for different initial thresholds. We used the WEB Corpus dataset and normalized edit similarity. We converted the initial similarity threshold needed by the 2PH algorithm to a bound on the number of common grams, g, using the formulas in [23]. Figure 17(a) shows the average running times for the three queries, using different bounds for g. For each query, the first bar represents the execution time for only the second phase of the 2PH algorithm, i.e., we mainly ran the SPS algorithm. The subsequent bars represent the running times with g between the number of grams in the query and 1. We can see how the initial threshold affected the performance of the algorithm, and why the algorithm can have a better time than the SPS algorithm. In particular, for the third query, when we decreased the similarity threshold, the overall time first decreased, then increased.

For each query in our workload, we computed the optimal g bound for the 2PH algorithm as follows. We ran each query with a g value between the number of grams in the query and 1, and selected the bound with the minimum running time. Figure 17(b) shows the average running time of the 2PH algorithm with a heuristically computed initial threshold (equal to the number of grams in the query) and the optimal initial threshold (shown as “2PH Opt” in the figure). This experimental result shows that the 2PH algorithm can indeed decrease the running time when a good initial threshold is used.
7.1.4. Number of Results

We analyzed how the performance of the 2PH algorithm changed when the required number of results, \( k \), changes. We assigned to \( k \) the following values: 10, 100, and 1000. Figure 18(a) shows the average running time of the algorithm on the IMDB dataset with 1.2 million entries. We used Jaccard similarity and normalized edit similarity (NES). We can see that when \( k \) increased, the execution time for NES increased more than that of Jaccard similarity. This is mainly because of the gram based filters are not very effective for the NES. Figure 18(b) shows the experimental result on the Web Corpus dataset. We notice the same trend, only that the difference is more significant.

7.1.5. Keyword Relaxation Scoring Functions

To test the robustness of our algorithms we ran the experiments using a different scoring function:

\[
F(\theta(r,s), w(s)) = \theta(r,s) \cdot w(s).
\]

We used the IMDB dataset and answered top-10 queries using the SPS algorithm. Figure 19(a) shows the average query time of the two scoring functions using the Jaccard similarity function. In the figure “Score 1” means the function in Equation 2, while “Score 2” means the function in Equation 3. The algorithm required more time when we used the new scoring function. The reason is the following. In the new scoring function, the weight has a greater impact on the total score than in the scoring function in Equation 2. As a consequence, the frequency threshold computed in line 13 in Algorithm 2 is looser for the new scoring function. Thus the algorithm skipped fewer elements and the total running time increased.

Figure 19(b) shows the results for the same setting but using the normalized edit similarity function. The algorithm needed approximately the same amount of time for both scoring functions. The reason is the following. For the normalized edit similarity, we computed bounds on the similarity, and the frequency threshold computed in line 13 in Algorithm 2 is looser than that of the Jaccard similarity. As a consequence, the new scoring function did not affect the frequency threshold as much as it did in the case of Jaccard similarity. Thus the algorithm needed a similar amount of time for both scoring functions.

7.1.6. Efficiency of Records Retrieval

We evaluated the performance of the entire query processing engine for the case where data resides outside the database. We used the SPS keyword relaxation algorithm for top-3 keyword relaxation. We used a query workload of 100 queries of 3 keywords each. We executed the a query plan similar to the one in Figure 3(b). We evaluated the engine using the Enron email dataset.

Figure 20(a) shows the average running time for each query as we increase the dataset size. The running time increases linearly with the dataset size. For 500 thousands Enron emails, the system answers a query in under 40ms. Figure 20(b) shows the same experiment, but gives a breakdown of the time needed for each operation. We can see how most of the time is spent for the keyword relaxation operation.

7.2. Keyword-Query Cleaning Inside DBMS

We evaluated the performance of the keyword-query cleaning techniques for the case where data resides inside the DBMS. We evaluated the performance of the keyword relaxation algorithms for top-3 queries using the IMDB dataset. We considered the scoring function in Equation 2, with \( \alpha = 1 \) and \( \beta = 1 \). We used the normalized edit similarity function for computing the similarity between keywords. Figure 21 shows the average running time of the “Scan” and “Gram Filter” query relaxation algorithms with increasing dataset size. We can see how the techniques needs around 1.7s to relax one keyword for a dataset
of 1.2 millions records. There is only a small running time improvement if we use the “Gram Filter” approach because of the overhead introduced by the join between the grams and the words tables. The Grams table required 10 times the space of the Keywords table. The “Iterative” algorithm had very poor performance is the fact that in most cases the algorithm needs to execute an approximate range-search where the gram threshold is 1. So, the algorithm ultimately has to execute the same query as the Gram Filter approach having already spent time on computing intermediate results.

We also evaluated the record retrieval efficiency using our solutions for the case where the data resides inside the database. We implemented the “Join” and “Full-Text” approaches and evaluated their performance using the Enron email dataset. We used the “Gram Filter” keyword relaxation algorithm for top-3 keyword relaxation. We used a query workload of 100 queries having 3 keywords each. Figure 21(b) shows the average running time of each query with increasing dataset size. We can see how the time necessary for the “Join” techniques increases quadratically due to the increased size of the joined tables. The Keywords to Records table required 1.5 times the space of the records table. The “Full-Text” index technique scales sub-linearly with increasing dataset size. Still, as mentioned earlier, the “Full-Text” technique does not rank the results.

7.3. Benefits from Grouping Keywords

We evaluated the performance gain that can be obtained by grouping keywords. We modified the SPS algorithm to retrieve top-k similar keywords for more than one keyword at a time. We used the IMDB dataset. We considered the scoring function in Equation 2, with \( \alpha = 1 \) and \( \beta = 1 \).

Figure 22(a) shows the average running time for finding the top-10 similar keywords for 2 keywords. We plotted the running time for the case where we did not group keywords and we run the SPS algorithm for one keyword at a time (“1 Keyword” in the figure). We also plotted the time for the case where we group two keywords that had 1 edit operation difference between them and 2 edit operations, respectively (“2 Keywords, ED 1” and “2 Keywords, ED 2” in the figure). As the gram length \( q \) is 3, 1 edit distance operation difference means that the each keyword had at most 3 extra grams which the other keywords did not have. We can see from the figure that the running time decreases if we group queries, especially in the case were the edit difference between them is small. Figure 22(b) shows the same experiment but for relaxing 3 keywords. We can see that grouping similar keywords can improve the performance even further.

8. Conclusions

In this paper we studied the problem of keyword-query cleaning in relational databases. We clean a query by relaxing its keywords. We developed techniques for query cleaning for the case where data resides outside or inside the database. We presented efficient algorithms for computing most relevant keywords for a query keyword. We systematically studied various approaches to query cleaning inside databases. We conducted thorough experiments on real data sets and reported the observations on the performance, limitations, and tradeoffs of these approaches.