Approximate String Joins in a Database (Almost) for Free

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Panagiotis G. Ipeirotis
Columbia University
pirot@cs.columbia.edu

H. V. Jagadish
University of Michigan
jag@eecs.umich.edu

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

S. Muthukrishnan
AT&T Labs–Research
muthu@research.att.com

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

Abstract

String data is ubiquitous, and its management has taken on particular importance in the past few years. Approximate queries are very important on string data especially for more complex queries involving joins. This is due, for example, to the prevalence of typographical errors in data, and multiple conventions for recording attributes such as name and address. Commercial databases do not support approximate string joins directly, and it is a challenge to implement this functionality efficiently with user-defined functions (UDFs).

In this paper, we develop a technique for building approximate string join capabilities on top of commercial databases by exploiting facilities already available in them. At the core, our technique relies on matching short substrings of length q, called q-grams, and taking into account both positions of individual matches and the total number of such matches. Our approach applies to both approximate full string matching and approximate substring matching, with a variety of possible edit distance functions. The approximate string match predicate, with a suitable edit distance threshold, can be mapped into a vanilla relational expression and optimized by conventional relational optimizers. We demonstrate experimentally the benefits of our technique over the direct use of UDFs, using commercial database systems and real data. To study the I/O and CPU behavior of approximate string join algorithms with variations in edit distance and q-gram length, we also describe detailed experiments based on a prototype implementation.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

1 Introduction

String data is ubiquitous. To name only a few commonplace applications, consider product catalogs (for books, music, software, etc.), electronic white and yellow page directories, specialized information sources such as patent databases, and customer relationship management data.

As a consequence, management of string data in databases has taken on particular importance in the past few years. Applications that collect and correlate data from independent data sources for warehousing, mining, and statistical analysis rely on efficient string matching to perform their tasks. Here, correlation between the data is typically based on joins between descriptive string attributes in the various sources. However, the quality of the string information residing in various databases can be degraded due to a variety of reasons, including human typing errors and flexibility in specifying string attributes. Hence the results of the joins based on exact matching of string attributes are often of lower quality than expected. The following example illustrates these problems:

Example 1.1 [String Joins] Consider a corporation maintaining various customer databases. Requests for correlating data sources are very common in this context. A specific customer might be present in more than one database because the customer subscribes to multiple services that the corporation offers, and each service may have developed its database independently. In one database, a customer's name may be recorded as `John A. Smith`, while in another database the name may be recorded as `Smith, John`. In a different database, due to a typing error, this name may be recorded as `Jonh Smith`. A request to correlate these databases and create a unified view of customers will fail to produce the desired output if exact string matching is used in the join. □

Unfortunately, commercial databases do not directly support approximate string processing functionality. Specialized tools, such as those available from Trillium Software[1], are useful for matching specific types of values such as addresses, but these tools are not integrated with

[1] `www.trillium.com`

databases. To use such tools for information stored in databases, one would either have to process data outside the database, or be able to use them as user-defined functions (UDFs) in an object-relational database. The former approach is undesirable in general. The latter approach is quite inefficient, especially for joins, because relational engines evaluate joins involving UDFs whose arguments include attributes belonging to multiple tables by essentially computing the cross-products and applying the UDFs in a post-processing fashion.

To address such difficulties, we need techniques for efficiently identifying all pairs of *approximately* matching strings in a database of strings. Whenever one deals with matching in an approximate fashion, one has to specify the approximation metric. Several proposals exist for strings to capture the notion of "approximate equality." Among those, the notion of *edit distance* between two strings is very popular. According to this notion, deletion, insertion, and substitution of a character are considered as unit cost operations and the edit distance between two strings is defined as the lowest cost sequence of operations that can transform one string to the other.

Although there is a fair amount of work on the problem of approximately matching strings (see Section 6), we are not aware of work related to approximately matching *all* string pairs based on edit distance (or variants of it), as is needed in approximate string *joins*. Moreover, we are not aware of any work related to this problem in the context of a relational DBMS.

In this paper, we present a technique for computing approximate string joins efficiently. At the core, our technique relies on matching short substrings of length q of the database strings (also known as q*-grams*). We show how a relational schema can be augmented to directly represent q-grams of database strings in auxiliary tables within the database in a way that will enable use of traditional relational techniques and access methods for the calculation of approximate string joins. By taking into account the total number of such matches and the positions of individual q-gram matches we guarantee no false dismissals under the edit distance metric, as well as variations of it, and the identification of a set of candidate pairs with a few false positives that can be later verified for correctness.

Instead of trying to invent completely new join algorithms from scratch (which would be unlikely to be incorporated into existing commercial DBMSs), we opted for a design that would require minimal changes to existing database systems. We show how the approximate string match predicate, with a suitable edit distance threshold, can be mapped into a vanilla SQL expression and optimized by conventional optimizers. The immediate practical benefit of our technique is that approximate string processing can be widely and effectively deployed in commercial relational databases without extensive changes to the underlying database system. Furthermore, by not requiring any changes in the DBMS internals, we can re-use existing facilities, like the query optimizer, join ordering algorithms and selectivity estimation.

The rest of the paper is organized as follows: In Section 2 we give the notation and the definitions that we will use. Then, in Section 3 we introduce formally the problem of approximate string joins and we present our proposal. In Section 4 we present the results of an experimental study comparing the proposed approach to other applicable methods, demonstrating performance benefits and presenting performance trends for several parameters of interest. Finally, in Section 5 we describe how we can adapt our techniques to address further problems of interest. In particular we show how to incorporate an alternate string distance function, namely the *block* edit distance (where edit operations on contiguous substrings are inexpensive), and we address the problem of approximate *substring* joins.

2 Preliminaries

2.1 Notation

We use R, possibly with subscripts, to denote tables, A, possibly with subscripts, to denote attributes, and t, possibly with subscripts, to denote records in tables. We use the notation $R.A_i$ to refer to attribute A_i of table R, and $R.A_i(t_j)$ to refer to the value in attribute $R.A_i$ of record t_j.

Let Σ be a finite alphabet of size $|\Sigma|$. We use lowercase Greek symbols, such as σ, possibly with subscripts, to denote strings in Σ^*. Let $\sigma \in \Sigma^*$ be a string of length n. We use $\sigma[i \ldots j]$, $1 \leq i \leq j \leq n$, to denote a substring of σ of length $j - i + 1$ starting at position i.

Definition 2.1 [Edit Distance] The *edit distance* between two strings is the minimum number of edit operations (i.e., *insertions*, *deletions*, and *substitutions*) of single characters needed to transform the first string into the second. □

2.2 Q-grams: A Foundation for Approximate String Processing

Below, we briefly review the notion of positional q-grams from the literature, and we give the intuition behind their use for approximate string matching [16, 15, 13].

Given a string σ, its *positional* q*-grams* are obtained by "sliding" a window of length q over the characters of σ. Since q-grams at the beginning and the end of the string can have fewer than q characters from σ, we introduce new characters "#" and "$" *not* in Σ, and conceptually extend the string σ by prefixing it with $q - 1$ occurrences of "#" and suffixing it with $q - 1$ occurrences of "$". Thus, each q-gram contains exactly q characters, though some of these may not be from the alphabet Σ.

Definition 2.2 [Positional q-gram] A *positional* q*-gram* of a string σ is a pair $(i, \sigma[i \ldots i+q-1])$, where $\sigma[i \ldots i+q-1]$ is the q-gram of σ that starts at position i, counting on the extended string. The set G_σ of all positional q-grams of a string σ is the set of all the $|\sigma| + q - 1$ pairs constructed from all q-grams of σ. □

The intuition behind the use of q-grams as a foundation for approximate string processing is that when two strings σ_1 and σ_2 are within a small edit distance of each other, they share a large number of q-grams in common [15, 13]. The following example illustrates this observation.

Example 2.1 [Positional q-gram] The positional q-grams of length q=3 for string `john_smith` are {`(1,##j)`, `(2,#jo)`, `(3,joh)`, `(4,ohn)`, `(5,hn_)`, `(6,n_s)`, `(7,_sm)`, `(8,smi)`, `(9,mit)`, `(10,ith)`, `(11,th$)`, `(12,h$$)`}. Similarly, the positional q-grams of length q=3 for string `john_a_smith`, which is at an edit distance of two from `john_smith`, are {`(1,##j)`, `(2,#jo)`, `(3,joh)`, `(4,ohn)`, `(5,hn_)`, `(6,n_a)`, `(7,_a_)`, `(8,a_s)`, `(9,_sm)`, `(10,smi)`, `(11,mit)`, `(12,ith)`, `(13,th$)`, `(14,h$$)`}. If we ignore the position information, the two q-gram sets have 11 q-grams in common. Interestingly, only the first five positional q-grams of the first string are also positional q-grams of the second string. However, an additional six positional q-grams in the two strings differ in their position by just two positions. This illustrates that, in general, the use of positional q-grams for approximate string processing will involve comparing positions of "matching" q-grams within a certain "band." □

In the next section we describe how we exploit the concept of q-grams to devise effective algorithms for approximate string *joins* (as opposed to the individual approximate string matches described above).

3 Approximate String Joins

In the context of a relational database, we wish to study techniques and algorithms enabling efficient calculation of approximate string joins. More formally, we wish to address the following problem:

Problem 1 (Approximate String Joins) *Given tables R_1 and R_2 with string attributes $R_1.A_i$ and $R_2.A_j$, and an integer k, retrieve all pairs of records $(t, t') \in R_1 \times R_2$ such that edit_distance($R_1.A_i(t)$, $R_2.A_j(t')$) $\leq k$.*

Our techniques for approximate string processing in databases share a principle common in multimedia and spatial algorithms. First, a set of candidate answers is obtained using a cheap, approximate algorithm that guarantees *no false dismissals*. We achieve this by performing a join on the q-grams along with some additional filters that are guaranteed not to eliminate any real approximate match. Then, as a second step, we use an expensive, in-memory algorithm to check the edit distance between each candidate string pair and we eliminate all *false positives*.

In the rest of this section we describe in detail the algorithms used, and how they can be mapped into vanilla SQL expressions. More specifically, the rest of the section is organized as follows. In Section 3.1 we describe the naive solution, which involves the direct application of user-defined functions (UDFs) to address the problem. In Section 3.2 we describe how to augment a database with q-gram information that is needed to run the approximate string joins. Finally, in Section 3.3 we describe a set of filters that we use to ensure a small set of candidates and we describe how to map these filters into SQL queries that can be subsequently optimized by regular query optimizers.

3.1 Exploiting User-Defined Functions

Our problem can be expressed easily in any object-relational database system that supports UDFs, such as Oracle or DB2. One could register with the database a ternary UDF `edit_distance(s1, s2, k)` that returns true if its two string arguments $s1, s2$ are within edit distance of the integer argument k. Then, the approximate string join problem for edit distance k could be represented in SQL as:

```
Q1:
  SELECT    R1.Ai, R2.Aj
  FROM      R1, R2
  WHERE     edit_distance(R1.Ai, R2.Aj, k)
```

To evaluate this query, relational engines would essentially have to compute the cross-product of tables R_1 and R_2, and apply the UDF comparison as a post-processing filter. However, the cross-products of large tables are huge and the UDF invocation, which is an expensive predicate, on every record in the cross-product makes the cost of the join operation prohibitive. For these reasons, we seek a better solution and we describe our approach next.

3.2 Augmenting a Database with Positional q-Grams

To enable approximate string processing in a database system through the use of q-grams, we need a principled mechanism for augmenting the database with positional q-grams corresponding to the original database strings.

Let R be a table with schema $(A_0, A_1, \ldots, A_m)$, such that A_0 is the key attribute that uniquely identifies records in R, and some attributes A_i, $i > 0$, are string-valued. For each string attribute A_i that we wish to consider for approximate string processing, we create an auxiliary table $RA_iQ(A_0, Pos, Qgram)$ with three attributes. For a string σ in attribute A_i of a record of R, its $|\sigma| + q - 1$ positional q-grams are represented as separate records in the table RA_iQ, where $RA_iQ.Pos$ identifies the position of the q-gram contained in $RA_iQ.Qgram$. These $|\sigma| + q - 1$ records all share the same value for the attribute $RA_iQ.A_0$, which serves as the foreign key attribute to table R.

Since the auxiliary q-gram tables are used *only* during the approximate join operation, they can be created on-the-fly, when the database wants to execute such an operation, and deleted upon completion. In the experimental evaluation (Section 4) we will show that the time overhead is negligible compared to the cost of the actual join. The space overhead for the auxiliary q-gram table for a string field A_i of a relation R with n records is:

$$S(RA_iQ) = n(q-1)(q+C) + (q+C)\sum_{j=1}^{n} |R.A_i(t_j)|$$

where C is the size of the additional fields in the auxiliary q-gram table (i.e., id and pos). Since $n(q-1) \leq \sum_{j=1}^{n} |R.A_i(t_j)|$, for any reasonable value of q, it follows that $S(RA_iQ) \leq 2(q+C)\sum_{j=1}^{n} |R.A_i(t_j)|$. Thus, the size of the auxiliary table is bounded by some linear function of q times the size of the corresponding column in the original table.

After creating an augmented database with the auxiliary tables for each of the string attributes of interest, we can

efficiently calculate approximate string joins using simple SQL queries. We describe the methods next.

3.3 Filtering Results Using q-gram Properties

In this section, we present our basic techniques for processing approximate string joins based on the edit distance metric. The key objective here is to efficiently identify candidate answers to our problems by taking advantage of the q-grams in the auxiliary database tables and using features already available in database systems such as traditional access and join methods.

For reasons of correctness and efficiency, we require *no false dismissals* and *few false positives* respectively. To achieve these objectives our technique takes advantage of three key properties of q-grams, and uses the three filtering techniques described below.

Count Filtering:

The basic idea of COUNT FILTERING is to take advantage of the information conveyed by the sets G_{σ_1} and G_{σ_2} of q-grams of the strings σ_1 and σ_2, *ignoring positional information*, in determining whether σ_1 and σ_2 are within edit distance k. The intuition here is that strings that are within a small edit distance of each other share a large number of q-grams in common.

This intuition has appeared in the literature earlier [14], and can be formalized as follows. Consider a string σ_1, and let σ_2 be obtained by a substitution of a single character in σ_1. Then, the sets of q-grams G_{σ_1} and G_{σ_2} differ by at most q (the length of the q-gram). This is because q-grams that do not overlap with the substituted character must be common to the two sets, and there are only q q-grams that can overlap with the substituted character. A similar observation holds true for single character insertions and deletions. In other words, in these cases, σ_1 and σ_2 must have at least $(\max(|\sigma_1|, |\sigma_2|)+q-1)-q = \max(|\sigma_1|, |\sigma_2|)-1$ q-grams in common. When the edit distance between σ_1 and σ_2 is k, the following lower bound on the number of matching q-grams holds.

Proposition 3.1 *Consider strings σ_1 and σ_2, of lengths $|\sigma_1|$ and $|\sigma_2|$, respectively. If σ_1 and σ_2 are within an edit distance of k, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_2}$, ignoring positional information, must be at least* $\max(|\sigma_1|, |\sigma_2|) - 1 - (k-1) * q$. □

Position Filtering:

While COUNT FILTERING is effective in improving the efficiency of approximate string processing, it does not take advantage of q-gram position information.

In general, the interaction between q-gram match positions and the edit distance threshold is quite complex. Any given q-gram in one string may not occur at all in the other string, and positions of successive q-grams may be off due to insertions and deletions. Furthermore, as always, we must keep in mind the possibility of a q-gram in one string occurring at multiple positions in the other string.

We define a positional q-gram (i, τ_1) in one string σ_1 to *correspond* to a positional q-gram (j, τ_2) in another string σ_2 if $\tau_1 = \tau_2$ and (i, τ_1), after the sequence of edit operations that convert σ_1 to σ_2, "becomes" q-gram (j, τ_2) in the edited string.

```
SELECT    R1.A0, R2.A0, R1.Ai, R2.Aj
FROM      R1, R1AiQ, R2, R2AjQ
WHERE     R1.A0 = R1AiQ.A0 AND
          R2.A0 = R2AjQ.A0 AND
          R1AiQ.Qgram = R2AjQ.Qgram AND
          |R1AiQ.Pos - R2AjQ.Pos| ≤ k AND
          |strlen(R1.Ai) - strlen(R2.Aj)| ≤ k
GROUP BY  R1.A0, R2.A0, R1.Ai, R2.Aj
HAVING    COUNT(*) ≥ strlen(R1.Ai) - 1 - (k-1) * q AND
          COUNT(*) ≥ strlen(R2.Aj) - 1 - (k-1) * q AND
          edit_distance(R1.Ai, R2.Aj, k)
```

Figure 1: Query Q2: Expressing COUNT FILTERING, POSITION FILTERING, and LENGTH FILTERING as an SQL expression.

Example 3.1 [Corresponding q-grams] Consider the strings σ_1 = `abaxabaaba` and σ_2 = `abaabaaba`. The edit distance between these strings is 1 (delete `x` to transform the first string to the second). Then $(7, aba)$ in σ_1 corresponds to $(6, aba)$ in σ_2 but not to $(9, aba)$. □

Notwithstanding the complexity of matching positional q-grams in the presence of edit errors in strings, a useful filter can be devised based on the following observation [13].

Proposition 3.2 *If strings σ_1 and σ_2 are within an edit distance of k, then a positional q-gram in one cannot correspond to a positional q-gram in the other that differs from it by more than k positions.* □

Length Filtering:

We finally observe that string length provides useful information to quickly prune strings that are not within the desired edit distance.

Proposition 3.3 *If two strings σ_1 and σ_2 are within edit distance k, their lengths cannot differ by more than k.* □

SQL Expression and Evaluation:

What is particularly interesting is that COUNT FILTERING, POSITION FILTERING, and LENGTH FILTERING can be naturally expressed as an SQL expression on the augmented database described in Section 3.2, and efficiently implemented by a commercial relational query engine. The SQL expression `Q2`, shown in Figure 1, modifies query `Q1` in Section 3.1 to return the desired answers.

Consequently, if a relational engine receives a request for an approximate string join, it can directly map it to a conventional SQL expression and optimize it as usual. (Of course, k and q are constants that need to be instantiated before the query is evaluated.)

Essentially, the above SQL query expression joins the auxiliary tables corresponding to the string-valued attributes $R_1.A_i$ and $R_2.A_j$ on their *Qgram* attributes, along with the foreign-key/primary-key joins with the original

database tables R_1 and R_2 to retrieve the string pairs that need to be returned to the user.

The POSITION FILTERING is implemented as a condition to the `WHERE` clause of the SQL expression above. The `WHERE` clause will prune out any pair of strings in $R_1 \times R_2$ that share many q-grams in common but that are such that the positions of the identical q-grams differ substantially. Hence, such pairs of strings will be eliminated from consideration before the `COUNT(*)` conditions in the `HAVING` clause are tested. Furthermore, this filter reduces the size of the *q-gram join*, hence it makes the computation of the query faster, since fewer pairs of q-grams have to be examined by the `GROUP BY` and the `HAVING` clause. The simplicity of this check when coupled with the ability of relational engines to use techniques like band-join processing [6] makes this a worthwhile filter.

The LENGTH FILTERING is implemented as an additional condition to the `WHERE` clause of the SQL expression above, which compares the lengths of the two strings. Again, like the POSITION FILTERING technique, this filter reduces the size of the *q-gram join*, and subsequently the size of the candidate set.

Finally the COUNT FILTERING is implemented mainly by the conditions in the `HAVING` clause. The string pairs that share only a few q-grams (and not significantly many) will be eliminated by the `COUNT(*)` conditions in the `HAVING` clause. Any string pairs in $R_1 \times R_2$ that do not share any q-grams are eliminated by the conditions in the `WHERE` clause.

However, even after the filtering steps the candidate set may still have false positives. Hence, the expensive UDF invocation `edit_distance(`$R_1.A_i, R_2.A_j, k$`)` still needs to be performed, but hopefully on just a small fraction of all possible string pairs.

We have included all the three filtering mechanisms in `Q2`. Of course any one of these filtering mechanisms may be left out of query `Q2`, and resulting queries will still perform our task albeit perhaps less efficiently. In Section 4, we quantify the benefits of each of the filtering mechanisms individually.

In Section 4, we quantify this performance difference using commercial database systems and real data sets. By examining the query evaluation plans generated by commercial database systems, under varying availability of access methods, we observed that relational engines make good use of traditional access methods and join methods in efficiently evaluating the above SQL expression.

4 Experimental Evaluation

In this section we present the results of an experimental comparison analyzing various trends in the approximate string processing operations. We start in Section 4.1 by describing the data sets that we used in our experiments. Then, in Section 4.2 we discuss the baseline experiments that we conducted using a commercial DBMS to compare our approach for approximate string joins against an implementation that uses SQL extensions in a straightforward way. Finally, in Section 4.3 we report additional experimental results for our technique using a prototype relational system we developed.

4.1 Data Sets

All data sets used in our experiments are real, with string attributes extracted by sampling from the AT&T WorldNet customer relation database. We have used three different data sets *set1*, *set2*, and *set3* for our experiments with different distributional characteristics.

Set1 consists of the first and last names of people. *Set1* has approximately 40K tuples, each with an average length of 14 characters. The distribution of the string lengths in *set1* is depicted in Figure 2(a): the lengths are mostly around the mean value, with small deviation. *Set2* was constructed by concatenating three string attributes from the customer database. *Set2* has approximately 30K tuples, each with an average length of 38 characters. The distribution of the string lengths in *set2* is depicted in Figure 2(b): the lengths follow a close-to-Gaussian distribution, with an additional peak around 65 characters. Finally, *set3* was constructed by concatenating two string attributes from the customer database. *Set3* has approximately 30K tuples, each with an average length of 33 characters. The distribution of the string lengths in *set3* is depicted in Figure 2(c): the length distribution is almost uniform up to a maximum string length of 67 characters.

4.2 DBMS Implementation

The first experiment we performed was to compare our approach with a straightforward SQL formulation of the problem with a function to compute the edit distance of two strings as a UDF, and performing a join query by essentially using the UDF invocation as the join predicate. This is a baseline comparison to establish the benefits of our approach. We implemented the function to assess the edit distance of two strings as a UDF[2] and we registered it in a commercial DBMS (Oracle 8i) running on a SUN 20 Enterprise Server.

We started by issuing the `Q1` query (see Section 3.1) to the DBMS, to evaluate a self-join on *set1*. As expected, the DBMS chose a nested loop join algorithm to evaluate the join. We tried to measure the execution times over this data set, but unfortunately the estimated time to finish the processing was extremely high (more than 3 days). Therefore, to compare our approach with the direct use of UDFs we decided to compare the methods for a random subset of *set1* consisting of 1,000 strings. Hence, we issued the `Q1` self-join query to determine string pairs in the small data set within edit distance of k. Moreover, to assess the utility of the proposed filters when applied as UDF functions, we registered an additional UDF that first applies the filtering techniques we proposed on *pairs of strings* supplied in the input, and if the string pair passes the filter, then determines if the strings are within distance k. Each of these queries took about 30 minutes to complete for this small data set. Applying filtering and edit distance computation *within* the UDF requires slightly longer time compared to `Q1`. Finally, we issued query `Q2`, which implements our technique (Section 3.3). The execution times in this case are in the order of one minute. The execution time increases as edit distance

[2] We implemented the $O(nk)$ decision algorithm to decide whether two strings match or not within edit_distance k.

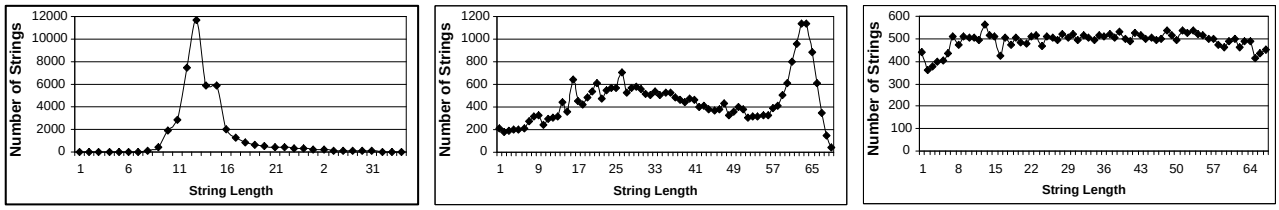


Figure 2: Distribution of String Lengths for the (a) *set1*, (b) *set2*, and (c) *set3* Data Sets.

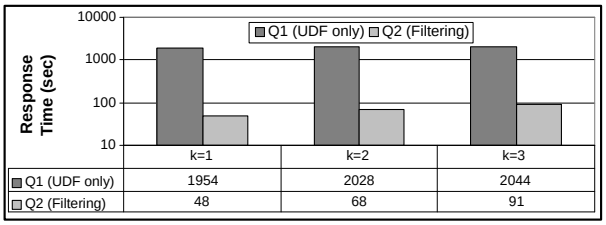


Figure 3: Executing Queries `Q1` and `Q2` over the Sample Database.

increases, since more strings are expected to be within the specified edit distance, and we have to verify more string pairs. The results are reported in Figure 3. It is evident that using our relational technique offers very large performance benefits, being more than 20 times faster than the straightforward UDF implementation.

Using query `Q2`, we also experimented with various physical database organizations for the commercial DBMS and observed the plans generated. When there were no indexes available on the q-gram tables, the joins are executed using hash-join algorithms and the group-by clause is executed using hashing. When there is an index on one or on both q-gram tables joins use sort-merge-join algorithms and the group-by clause is executed using hashing.

4.3 Performance of Approximate String Processing Algorithms

Based on the intuition obtained using the commercial DBMS, we developed a home-grown relational system prototype to conduct further experiments in a more controlled and flexible fashion, disassociating ourselves and our observations from component interactions between DBMS modules.

We emphasize that our objective is to observe performance trends under the parameters that are associated with our problem (i.e., q-gram size, number of errors allowed). These experiments are not meant to evaluate the relative performance of the join algorithms. Choosing which algorithm to use in each case is the task of the query optimizer and modern optimizers are effective for this task.

We conducted experiments using our prototype and the data sets of Section 4.1. In our prototype, LENGTH FILTERING and POSITION FILTERING are applied before creating the join on the q-gram relations. Then, COUNT FILTERING takes place using hashing on the output of the join operations between q-gram relations. We used two performance metrics: the size of the candidate set and the total running time of the algorithm decomposed into processor time and I/O time. The processor time includes the time to validate the distance between candidate pairs, and the I/O time includes the time for querying the auxiliary tables.

The results below do not include the time to generate and index the auxiliary tables. For all the data sets the time spent to generate the auxiliary tables was less than 100 seconds and the time to create a B-tree index on them, using bulk loading, was less than 200 seconds. Hence, it seems feasible to generate these tables on the fly before an approximate string join.

We now analyze the performance of approximate string join algorithms under various parameters of interest.

Effect of Filters

In the worst case (like in query `Q1`), the cross product of the relations has to be tested for edit distance. The aim of introducing filters was to reduce the number of candidate pairs tested. The perfect filter would eliminate all the false positives, giving the exact answer that would need no further verification. To examine how effective each filter and each combination of filters is, we ran different queries, enabling different filters each time, and measured the size of the candidate set. Then, we compared its size against that of the cross product and against the size of the real answer with no false positives.

We examined first the effectiveness of LENGTH FILTERING for the three data sets. As expected, LENGTH FILTERING was not so effective for *set1*, which has a limited spread of string lengths (Figure 2(a)). LENGTH FILTERING gave a candidate set that was between 40% to 70% of the cross-product size (depending on the edit distance). On the other hand, LENGTH FILTERING was quite effective for *set2* and *set3*, which have strings of broadly variable lengths (Figure 2(b) and (c)). The candidate set size was between 1.5% to 10% of the cross-product size. The detailed results are shown in Figure 4.

Enabling COUNT FILTERING in conjunction with LENGTH FILTERING causes a dramatic reduction on the number of candidate pairs: on average (over the various combinations of k and q tested) the reduction is more than 99% for all three data sets. On the other hand, enabling POSITION FILTERING with LENGTH FILTERING reduces the number of candidate pairs, but the difference is not so dramatic. On average it shrinks the size of the candidate set by 50%. Finally, enabling all the filters together worked best, as expected, with only 50% as many candidate pairs as those without POSITION FILTERING, confirming our previous measurement that position filtering reduces the candidate set by a factor of two. The comparative results for the three data sets are depicted in Figure 4.

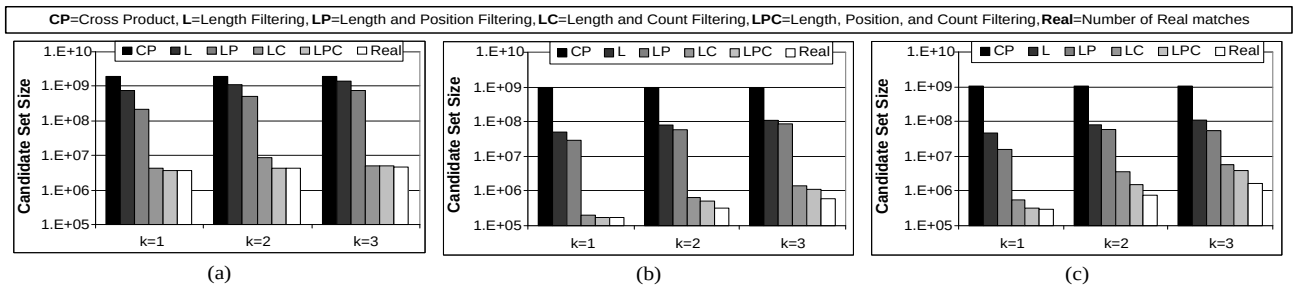
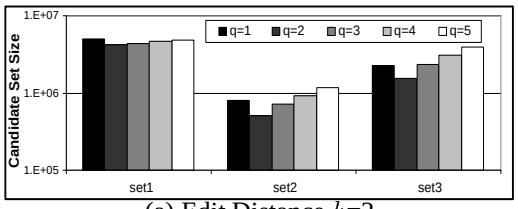
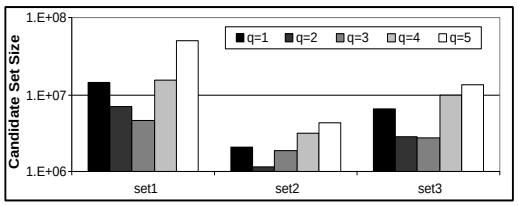


Figure 4: Candidate Set Size for Various Filter Combinations and the (a) *set1*, (b) *set2*, and (c) *set3* Data Sets (q=2).

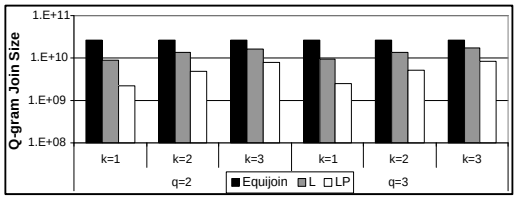


(a) Edit Distance k=2

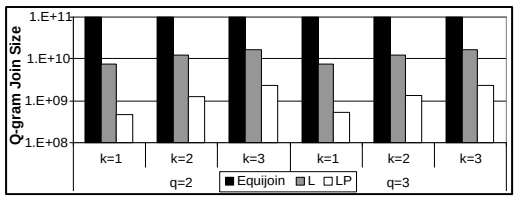


(b) Edit Distance k=3

Figure 5: Candidate Set Size for Various q-gram Lengths with All Filters Enabled.



(a) Size of the q-gram Join for the *set1* Data Set



(b) Size of the q-gram Join for the *set2* Data Set

Figure 6: Size of the q-gram Joins for Various Filter Combinations.

Our experiments indicate that a small value of q tends to give better results. We observe that values of q greater than three give consistently worse results compared to smaller values. This is due to the threshold for COUNT FILTERING, which gets less tight for higher q's. Furthermore, the value of q=1 gives worse results than q=2, because the value q=1 does not allow for q-gram overlap. When q=2 or q=3, the results are inconclusive. However, since a higher value of q results in increased space overhead (Section 3.2), q=2 seems preferable. The increased efficiency for $q \approx 2$ confirms approximate theoretical estimations in [10] about the optimal value of q for approximate string matching with very long strings ($q = \log_{|\Sigma|}(m)$, where m is the length of the string). In Figure 5 we plot the results for k=2 and k=3.

Finally, we examined the effect of LENGTH FILTERING and POSITION FILTERING on *the size of the q-gram join* (i.e., the number of tuples in the join of the q-gram tables *before* the application of the `GROUP BY, HAVING` clause). The effectiveness of these filters plays an important role in the execution time of the algorithm. If the filters are effective, the q-gram join is small and the calculation of COUNT FILTERING is faster, because the `GROUP BY, HAVING` clauses have to examine fewer q-gram pairs. Our measurements show that LENGTH FILTERING decreases the size of the q-gram join by a factor of 2 to 12 compared to the naive equijoin on the q-gram attribute (the decrease was higher for *set2* and *set3*). Furthermore, POSITION FILTERING, combined with LENGTH FILTERING, gives even better results, resulting in join sizes that are up to two orders of magnitude smaller than that from the equijoin. In Figure 6 we illustrate the effectiveness of the filters for *set1* and *set2* (the results for *set3* were similar to the ones for *set2*). These results validate our intuition that POSITION FILTERING is a useful filter, especially in terms of time efficiency.

Effect of Different Query Plans

We first report the trends for algorithms that do not make use of indexes on the q-gram relations, and then show the trends for the algorithms that use indexes on the q-gram relations to perform the join operations. We describe our observations in the sequel. Due to space constraints, we present only results for self-joins using *set1*. The performance trends are similar for the other data sets and for joins that are not self-joins.

No Index Available: In the absence of indexes on relations RA_iQ, the applicable algorithms are Nested Loops (NL), Hash Join (HJ), and Sort-Merge Join (SM). We omit NL from the plots, as this algorithm takes approximately 14 hours to complete for edit distance k=1. Figures 7(a)(b) show the results as the edit distance threshold is increased for q-gram size of 3 (Figure 7(a)). We observe that as the distance threshold is increased, the overall execution time increases both in processor and I/O time. The trends match our expectations: I/O time increases as the number of candidate pairs increases, because more pairs are hashed

during the hash-based counting phase, for both algorithms. Processor time increases since the candidate set has more string pairs, thus more strings have to be tested. As k and q increase, the overall time increases (Figure 7(b)). Both algorithms become heavily processor bound for k=3, q=5 as COUNT FILTERING becomes less effective and large numbers of false positive candidates are generated that subsequently have to be verified.

Indexes Available: We differentiate between two cases: (a) one of the two relations joined is indexed, and (b) both relations have B-tree indexes on them. In the first case, we present results for Indexed Nested Loops (INL) and SM. When both relations have indexes on them, SM is tested. When there is only one index, then SM performs much better than INL. Figures 7(c)(d) present the results for this case. INL performs multiple index probes and incurs a high I/O time. The performance trends are consistent with those observed above for the no index case, both for varying q-gram size and for varying the edit distance threshold. When the size of the q-gram relations involved varies (e.g., when one relation consists only of a few strings), the trends are the same both for increasing the q-gram size as well as for increasing k. In this case, however, INL performs fewer index probes and might be chosen by the optimizer. In practice, the DBMS picks INL as the algorithm of choice only when one of the q-gram relations is very small compared to the other. For all the other cases, SM was the algorithm of choice and this is also confirmed by our measurements with the prototype implementation.

Figures 7(e)(f) present the results for the case when indexes are available on both relations for increasing number of errors and two q-gram sizes (Figure 7(e)) and increasing q-gram size for two values of k (Figure 7(f)). The performance trends are consistent with those observed so far, both for varying q-gram size and for varying the edit distance threshold.

5 Extensions

We now illustrate the utility of our techniques for two extensions of our basic problems: (i) approximate *substring* joins, and (ii) approximate string joins when we allow *block moves* to be an inexpensive operation on strings.

5.1 Approximate Substring Joins

The kinds of string matches that are of interest are often based on one string being a substring of another, possibly allowing for some errors. For example, an attribute `CityState` of one table may contain city and state information for every city in the United States, while another attribute `CustAddress` (of a different table) may contain addresses of customers. One might be interested in correlating information in the two tables based on values in the `CityState` attribute being substrings of the `CustAddress` attribute, allowing for errors based on an edit distance threshold. The formal statement of the approximate substring join problem is:

Problem 2 (Approximate Substring Join) *Given tables R_1 and R_2 with string attributes $R_1.A_i$ and $R_2.A_j$, and an integer k, retrieve all pairs of records $(t, t') \in R_1 \times R_2$ such that for some substring σ of $R_2.A_j(t')$, edit_distance($R_1.A_i(t), \sigma$) $\leq k$.*

In order to use our approach, we reexamine what filtering techniques can be applied for this problem. For a string σ_1 to be within edit distance k of a substring σ_3 of σ_2, it must be the case that σ_1 and σ_3 (and hence σ_2) must have a certain minimum number of matching q-grams. Additionally, the positions of these matches must be in the right order and cannot be too far apart.

Clearly, LENGTH FILTERING is not applicable in this case. However, it follows from the first observation above that COUNT FILTERING is still applicable. Proposition 3.1 needs to be replaced by the following (weaker) proposition:

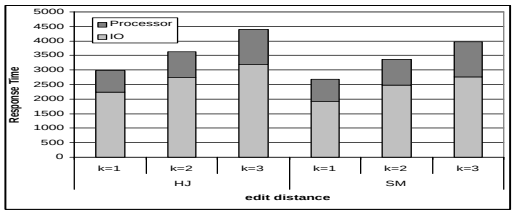
Proposition 5.1 *Consider strings σ_1 and σ_2. If σ_2 has a substring σ_3 such that σ_1 and σ_3 are within an edit distance of k, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_2}$, ignoring positional information, must be at least $|\sigma_1| + 1 - (k+1) * q$.* □

The applicability of POSITION FILTERING is complex. While it is true from the second observation above that the positions of the q-gram matches cannot be too far apart, the q-gram at position i in σ_1 may match at any arbitrary position in σ_2 and not just in $i \pm k$. Hence, POSITION FILTERING is not directly applicable for approximate substring matching.

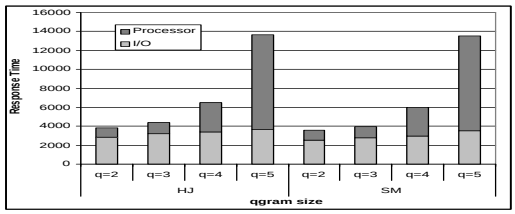
The SQL query expression for computing an approximate substring join between R_1 and R_2 incorporating "substring style" can be easily devised from query `Q2`, if we remove the clauses that perform the position and length filtering and we replace the *edit_distance* UDF with the appropriate one.

The standard algorithm for determining all approximate occurrences of a string σ_1 in σ_2 is rather expensive, taking time $O(n^3)$ in the worst case. Here we develop an alternative filtering algorithm, called *Substring Position Filtering* (SPF), that is based on q-grams and their relative positions, and quickly (in quadratic time) provides a check whether one string σ_1 is an approximate substring of another string σ_2. We will briefly describe the SPF algorithm here; for given strings σ_1 and σ_2, SPF certifies σ_2 to be a candidate for approximate match of σ_1 as an approximate substring in one or more places within threshold k. As before, this is a filter with no false dismissals.

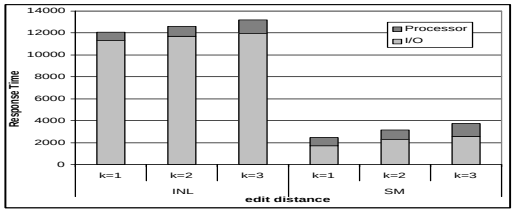
SPF works by finding any one place where σ_1 potentially occurs in σ_2, if any. Let γ_i be the i^{th} q-gram in string σ_1, $1 \leq i \leq i_{\max} = |\sigma_1| + q - 1$. Let $pos(\gamma_i, \sigma_2)$ be the set of positions in σ_2 at which q-gram γ_i occurs; this set may be empty. The algorithm, shown in Figure 8, may be thought of as using standard dynamic programming for edit-distance computation, but savings are achieved by (i) applying the algorithm sparsely only at a subset of positions in σ_2 guided by the occurrences of certain q-grams (line 3 of SPF), and (ii) applying only part of the dynamic programming, again guided by certain q-grams (line 5 of SPF). Algorithm SubMatch is the dynamic programming part, which is described here in a top-down recursive way where the table SubMatchArray is filled in as it is computed and read as needed (this is needed since which entries of the SubMatchArray will be computed depends on



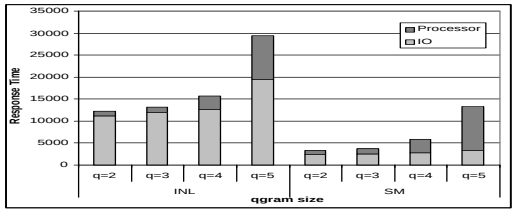
(a) Increasing Edit Distance k for q-gram Length q=3



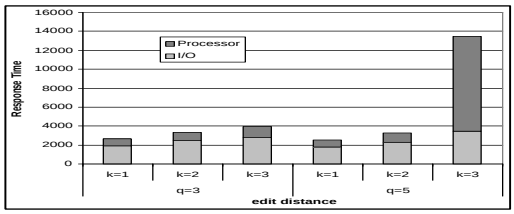
(b) Increasing q-gram Length q for Edit Distance k=3
No Index Available.



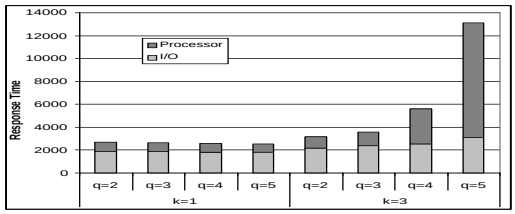
(c) Increasing Edit Distance k for q-gram Length q=3



(d) Increasing q-gram Length q for Edit Distance k=3
One Index Available.



(e) Increasing Edit Distance k for q-gram Length q=3,5



(f) Increasing q-gram Length q for Edit Distance k=1,3
Two Indexes Available.

Figure 7: Response Time (in seconds) for Various Physical Database Organizations.

```
Algorithm SPF(σ1,σ2,k) {
    missing = 0;
    for (i = 1; i ≤ i_max; i++)
        for j ∈ pos(γ_i, σ2)
            cost = SubMatch(σ2, j + 1, i + 1);
            if ((missing + cost) ≤ k * q)
                return {σ2};
        missing++;
    return ∅;
}

SubMatch(σ, j, i) {
    if (i > i_max) return 0;
    if SubMatchArray[j, i] already computed, return it.
    if (j ∈ pos(γ_i, σ))
        return SubMatch(σ, j + 1, i + 1);
    insertion_cost = 1 + SubMatch(σ, j + 1, i);
    deletion_cost = 1 + SubMatch(σ, j, i + 1);
    substitution_cost = 1 + SubMatch(σ, j + 1, i + 1);
    SubMatchArray[j, i] = min{insertion_cost,
        deletion_cost, substitution_cost}
    return SubMatchArray[j, i];
}
```

Figure 8: Substring Position Filtering (SPF) Algorithm.

relative occurrences of q-grams, and cannot be ascertained a priori).

5.2 Allowing for Block Moves

Traditional string edit distance computations are for single character insertions, deletions, and substitutions. However, in many applications we would like to allow block move operations as well. A natural example is in matching names of people; we would like to be able to match "first-name last-name" with "last-name, first-name" using an error metric that is independent of the length of first-name or last-name. It turns out that the q-gram method is well suited for this enhanced metric. For this purpose, we begin by extending the definition of edit distance.

Definition 5.1 [Extended Edit Distance] The *extended edit distance* between two strings is the minimum cost of edit operations needed to transform the first string into the second. The operations allowed are single-character insertion, deletion, and substitution at unit cost, and the movement of a block of contiguous characters at a cost of β units. □

The extended edit distance between two strings σ_1 and σ_2 is symmetric and $0 \leq$ extended_edit_distance$(\sigma_1, \sigma_2) \leq max(|\sigma_1|, |\sigma_2|)$.

Theorem 5.1 *Let G_{σ_1} and G_{σ_2} be the sets of q-grams (of length q) for strings σ_1 and σ_2 in the database. If σ_1 and σ_2 are within an extended edit distance of k, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_2}$, ignoring positional information, is at least $\max(|\sigma_1|, |\sigma_2|) - 1 - 3(k-1)q/\beta'$, where $\beta' = \min(3, \beta)$.*

Intuitively, the bound arises from the fact that the block move operation can transform a string of the form $\alpha\nu\delta\mu$ to $\alpha\delta\nu\mu$, which can result in up to $3q - 3$ mismatching q-grams.

Based on the above observations, it is easy to see that one can apply COUNT FILTERING (with a suitably modified threshold) and LENGTH FILTERING for approximate string processing with block moves. However, incorporating POSITION FILTERING is difficult as described earlier because block moves may end up moving q-grams arbitrarily. Nevertheless, we can design an enhanced filtering mechanism (just as we did with the SPF algorithm in the previous section) and incorporate it together with count filtering into a SQL query as before. Due to space limitation we do not list the details.

6 Related Work

A large body of work has been devoted to the development of efficient solutions to the approximate string matching problem. For two strings of length n and m, available in main memory, there exists a folklore dynamic programming algorithm to compute the edit distance of the strings in $O(nm)$ time and space [12]. Improvements to the basic algorithm have appeared, offering better average and worst case running times as well as graceful space behavior. Due to space limitations, we do not include a detailed survey here, but we refer the reader to [10] for an excellent overview of the work as well as additional references.

Identifying strings approximately in secondary storage is a relatively new area. Indexes such as Glimpse [9] store a dictionary and use a main memory algorithm to obtain a set of words to retrieve. Exact text searching is applied thereafter. These approaches are rather limited in scope due to the static nature of the dictionary, and they are not suitable for dynamic environments or when the domain of possible strings is unbounded. Other approaches rely on suffix trees to guide the search for approximate string matches [4, 11]. In [1], Baeza-Yates and Gonnet solve the problem of exact substring joins, using suffix arrays and outside the context of a relational database.

In the context of databases, several indexing techniques proposed for arbitrary metric spaces [3, 2] could be applied for the problem of approximately retrieving strings. However such structures have to be supported by the database management system.

Cohen [5] presented a framework for the integration of heterogeneous databases based on textual similarity and proposed WHIRL, a logic that reasons explicitly about string similarity using TF-IDF term weighting, from the vector-space retrieval model, rather than the notions of edit distance on which we focus in this paper.

Grossman et al. [7, 8] presented techniques for representing text documents and their associated term frequencies in relational tables, as well as for mapping boolean and vector-space queries into standard SQL queries. In this paper, we follow the same general approach of translating complex functionality not natively supported by a DBMS (approximate string queries in our case) into operations and queries that a DBMS can optimize and execute efficiently.

7 Conclusions

String processing in databases is a very fertile and useful area of research, especially given the proliferation of web based information systems. The main contribution of this paper is an effective technique for supporting approximate string processing *on top of a database system*, by using the *unmodified* capabilities of the underlying system. We showed that significant performance benefits are to be had by using our techniques.

Acknowledgments

L. Gravano and P. Ipeirotis were funded in part by the National Science Foundation (NSF) under Grants No. IIS-97-33880 and IIS-98-17434. P. Ipeirotis is also partially supported by Empeirikeio Foundation. The work of H.V. Jagadish was funded in part by NSF under Grant No. IIS-00085945.

References

[1] R. Baeza-Yates and G. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proceedings of String Processing and Information Retrieval Symposium (SPIRE'99)*, pages 16–23, 1999.

[2] T. Bozkaya and Z. M. Ozsoyoglu. Distance based indexing for high dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD Conference on Management of Data*, pages 357–368,1997.

[3] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[4] A. Cobbs. Fast approximate matching using suffix trees. In *Combinatorial Pattern Matching, 6th Annual Symposium (CPM'95)*, pages 41–54, 1995.

[5] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD Conference on Management of Data*, pages 201–212, 1998.

[6] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB'91)*, pages 443–452, 1991.

[7] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. In *Journal of the American Society for Information Science (JASIS)*, 48(2):122–132, 1997.

[8] C. Lundquist, O. Frieder, D. O. Holmes, and D. A. Grossman. A parallel relational database management system approach to relevance feedback in information retrieval. In *Journal of the American Society for Information Science (JASIS)*, 50(5):413–426, 1999.

[9] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

[10] G. Navarro. A guided tour to approximate string matching. To appear in *ACM Computing Surveys*, 2001.

[11] S. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science*, pages 320–328, 1996.

[12] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. In *Journal of Molecular Biology*, 147:195–197, 1981.

[13] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of Third Annual European Symposium (ESA'95)*, pages 327–340, 1995.

[14] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Combinatorial Pattern Matching, 7th Annual Symposium (CPM'96)*, pages 50–63, 1996.

[15] E. Ukkonen. Approximate string matching with q-grams and maximal matches. In *Theoretical Computer Science (TCS)*, 92(1):191–211, 1992

[16] J. Ullman. A binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words. In *The Computer Journal* 20(2):141–147, 1977.