Using q-grams in a DBMS for Approximate String Processing

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

Lauri Pietarinen
ATBusiness Communications
lauri.pietarinen@atbusiness.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. 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 queries 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 processing capabilities on top *of commercial databases by exploiting facilities already available in them. At the core, our technique relies on generating short substrings of length q, called q-grams, and processing them using standard methods available in the DBMS. The proposed technique enables various approximate string processing methods in a DBMS, for example approximate (sub)string selections and joins, and can even be used 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.*

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. However, the quality of the string information residing in various databases can be degraded due

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

to a variety of reasons, including human typing errors and flexibility in specifying string attributes. Hence, the results of operations based on exact matching of string attributes are often of lower quality than expected.

For example, 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 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 of the tables and applying the UDFs in a post-processing fashion.

Although there is a fair amount of work on the problem of approximate string matching (see, for example, [3]), these results are not used in the context of a relational DBMS. In this paper, we present a technique for incorporating approximate string processing capabilities to a database. At the core, our technique relies on using 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 performing approximate string matching operations. 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 to the DBMS internals, we can re-use existing facilities, like the query optimizer, join ordering algorithms and selectivity estimation.

The rest of the paper, which reports and expands on work originally presented in [2], is organized as follows. In Section 2, we present notation and definitions. In Section 3, we develop a principled mechanism for augmenting a database with q-gram tables. We describe the conceptual techniques for approximate string processing using q-grams in Section 4. Finally, in Section 5, we show how these conceptual techniques can be realized using SQL queries.

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 lower-case 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.

[1] `www.trillium.com`

To match strings *approximately* in a database, we need to specify the approximation metric. Several proposals exist for strings to capture the notion of "approximate equality." Among them, the notion of *edit distance* between two strings is very popular.

Definition 1: 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 one string into the other. □

Although we will mainly focus on the *edit distance* metric in this paper, we note that our proposed techniques can be used for a variety of other distance metrics as well.

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 [7, 6, 4]. 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: 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 [6, 4]. Consider the following example. 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 `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 each. 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."

3 Augmenting a Database with Positional q-Grams

To enable approximate string processing in a database system based on 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, 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 $|\sigma|+q-1$ separate records in the table RA_iQ, where $RA_iQ.Pos$ identifies the position of the q-gram $RA_iQ.Qgram$. 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.

Interestingly, these tables can be created in current database systems, using simple SQL statements. To do so, we use a table N that contains a single attribute I with the numbers from 1 to M (where M is the maximum

```
INSERT  INTO RA_iQ
        SELECT R.A_0, N.I,
           SUBSTR(SUBSTR('#...#',1,q-1) || UPPER(R.A_i) || SUBSTR('%...%',1,q-1), N.I, q)
        FROM R, N
        WHERE N.I <= LENGTH(R.A_i)+q-1;
```

Figure 1: Creating the auxiliary q-gram table RA_iQ

length of a string) [1]. Then, we join this table with the column $R.A_i$, and we take all the q-grams of each string in $R.A_i$ that start at position x, where x is the value stored in field I of a tuple of N. The result of this join is then used to create the auxiliary table RA_iQ. The exact SQL query is presented in Figure 1.

The space overhead for the auxiliary q-gram table for a string attribute 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 attributes in the auxiliary q-gram table (i.e., A_0 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.

Depending on the frequency of the approximate string operations, the database administrator can choose whether or not to have the tables permanently materialized. If the space overhead is not an issue, then the cost of keeping the auxiliary tables updated is relatively small. After creating an augmented database with the auxiliary tables for each of the string attributes of interest, we can efficiently perform approximate string processing using simple SQL queries. We describe the methods next.

4 Filtering Results Using q-gram Properties

In this section, we present our basic techniques for approximate string processing based on the *edit distance metric*. Later we will describe appropriate modifications to these filters to accommodate alternative distance metrics. 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.

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 [5], and can be formalized as follows.

Proposition 3: 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 $(\max(|\sigma_1|, |\sigma_2|) + q - 1) - k * q$. □

Intuitively, this holds because one edit distance operation can modify *at most* q q-grams, so k edit distance operations can modify at most kq q-grams.

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.

Intuitively, a positional q-gram (i, τ_1) in one string σ_1 is said 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 and affect *only the position* of the q-gram τ_1, "becomes" q-gram (j, τ_2) in the edited string. 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 [4].

Proposition 4: 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 5: If strings σ_1 and σ_2 are within edit distance k, their lengths cannot differ by more than k. □

5 Approximate String Processing in a Database

Below we describe how we can use the previously described properties of q-grams to perform approximate string processing tasks inside a database system. Additional details, including an experimental evaluation, are presented in [2].

5.1 Approximate String Selections

This problem can be formalized as follows: Given a table R with a string attribute $R.A_i$ and a string query σ, retrieve all records $t \in R$ such that edit_distance$(\sigma, R.A_i(t)) \leq k$.

To perform this operation it is first necessary to create the q-gram set for the query string σ. This can be done easily in SQL, in a manner similar to the SQL statement of Figure 1. These q-grams are stored in a small auxiliary table TQ. After this step, it is possible to find all the strings in $R.A_i$ that are possible candidate answers. This can be achieved on the augmented database using the SQL statement of Figure 2 that implements the filters described in Section 4. Consequently, if a relational engine receives a request for an approximate string operation, 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.) However, even after the filtering steps, the candidate set may still have false positives. Hence, a UDF invocation `edit_distance(`$R.A_i, \sigma, k$`)` still needs to be performed, but hopefully on just a small fraction of the strings.

5.2 Approximate String Joins

In a similar manner, we can efficiently implement approximate string joins: given two tables R_1 and R_2 with string attributes $R_1.A_i$ and $R_2.A_j$ respectively, report all pairs of strings that are within edit distance k.

In this case, we directly join the auxiliary q-gram tables, and we report pairs of strings with enough corresponding q-grams in common. Essentially, the SQL query expression in Figure 3 joins the auxiliary tables corresponding to the string-valued attributes $R_1.A_i$ and $R_2.A_j$ on their $Qgram$ and Pos 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.

```
SELECT    R.A_0, R.A_i
FROM      R, TQ, RA_iQ
WHERE     R.A_0 = RA_iQ.A_0 AND RA_iQ.Qgram = TQ.Qgram AND
          RA_iQ.Pos <= TQ.Pos + k AND RA_iQ.Pos >= TQ.Pos - k AND
          LENGTH(R.A_i) <= LENGTH(σ) + k AND LENGTH(R.A_i) >= LENGTH(σ) - k
GROUP BY  R.A_0, R.A_i
HAVING    COUNT(*) >= LENGTH(R.A_i) - 1 - (k - 1) * q AND COUNT(*) >= LENGTH(σ) - 1 - (k - 1) * q
```

Figure 2: Performing approximate string selections in an augmented DBMS using SQL

```
SELECT    R_1.A_0, R_2.A_0, R_1.A_i, R_2.A_j
FROM      R_1, R_1A_iQ, R_2, R_2A_jQ
WHERE     R_1.A_0 = R_1A_iQ.A_0 AND R_2.A_0 = R_2A_jQ.A_0 AND
          R_1A_iQ.Qgram = R_2A_jQ.Qgram AND
          R_1A_iQ.Pos <= R_2A_jQ.Pos + k AND R_1A_iQ.Pos >= R_2A_jQ.Pos - k AND
          LENGTH(R_1.A_i) <= LENGTH(R_2.A_j) + k AND LENGTH(R_1.A_i) >= LENGTH(R_2.A_j) - k
GROUP BY  R_1.A_0, R_2.A_0, R_1.A_i, R_2.A_j
HAVING    COUNT(*) >= LENGTH(R_1.A_i) - 1 - (k - 1) * q AND COUNT(*) >= LENGTH(R_2.A_j) - 1 - (k - 1) * q
```

Figure 3: Performing approximate string joins in an augmented DBMS using SQL

5.3 Approximate Substring Processing

A different type of approximate string match of interest is based on one string being a substring of another, possibly allowing for some errors. We can formalize the approximate substring selection problem as follows. Given a table R with a string attribute $R.A_i$ and a query string σ, retrieve all records t from R, such that for some substring σ_R of $R.A_i(t)$, edit_distance(σ_R, σ) $\leq k$. For this edit distance metric, we have to revise the filters described in Section 4. Specifically, LENGTH FILTERING and POSITION FILTERING are not applicable, since the q-gram at position i in σ may match at any arbitrary position in $R.A_i(t)$ and not just in $i \pm k$. Also $R.A_i(t)$ might be of arbitrary length and still have a substring match with σ. Finally, COUNT FILTERING has a different threshold, reflecting the fact that the q-grams at the beginning and at the end of σ (with the "extended" characters '#' and '%') might not match the respective q-grams of $R.A_i(t)$.

Proposition 6: Consider strings σ_1 and σ_2. If σ_2 has a substring σ_S such that σ_1 and σ_S are within an edit distance of k, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_S}$, ignoring positional information, must be at least $|\sigma_1| - (k + 1)q + 1$. □

Using this result, it is possible to write the respective SQL queries to perform selections and joins based on approximate substring matches. The SQL expressions are very similar to the ones described in Figures 2 and 3, but with a different threshold for COUNT FILTERING and without the conditions that perform the POSITION and LENGTH FILTERING.

5.4 Allowing for Block Moves

Traditional string edit distance computations are for single character insertions, deletions and substitutions. If a whole block of characters is modified or moved, the cost charged is proportional to the length of the block. In many applications, we would like to keep a fixed charge for block move operations, independent of block length.

It turns out that the q-gram method is suited to this enhanced metric, and in this section we consider the issues involved in so doing. For this purpose, we begin by extending the definition of edit distance.

Definition 7: The *extended edit distance* between two strings is the minimum cost of edit operations needed to transform one string into the other. 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. $\square$

Theorem 8: Let $G_{\sigma_1}, G_{\sigma_2}$ be the set of q-grams for strings σ_1 and σ_2 in the database. If the extended edit distance between σ_1 and σ_2 is less than 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 $\beta' = min(3, \beta)$. $\square$

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 not possible as described earlier because block moves may end up moving q-grams arbitrarily.

Again, it is possible to write the appropriate SQL queries to perform selections and joins based on the extended edit distance. The statements will apply only the correct filters and will return a set of candidate answers than can be later verified for correctness using a suitable UDF.

6 Conclusions

The ubiquity of string data in a variety of databases, and the diverse population of users of these databases, has brought the problem of string-based querying and searching to the forefront of the database community. Given the preponderance of errors in databases, and the possibility of mistakes by the querying agent, returning query results based on approximate string matching is crucial. In this paper, we have demonstrated that approximate string processing can be widely and effectively deployed in commercial relational databases without extensive changes to the underlying database system.

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. The work of H. V. Jagadish was funded in part by NSF under Grant No. IIS-00085945.

References

[1] Hugh Darwen. A constant friend. In *Relational Database Writings 1985-1989 by C. J. Date*, pages 493–500. Prentice Hall, 1990.

[2] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, pages 491–500, 2001.

[3] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

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

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

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

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