LexEQUAL: Supporting Multiscript Matching in Database Systems *

A. Kumaran and Jayant R. Haritsa

Database Systems Lab, SERC/CSA
Indian Institute of Science, Bangalore 560012, INDIA
{kumaran,haritsa}@dsl.serc.iisc.ernet.in

Abstract. To effectively support today's global economy, database systems need to store and manipulate text data in multiple languages simultaneously. Current database systems do support the storage and management of multilingual data, but are not capable of querying or matching text data across different scripts. As a first step towards addressing this lacuna, we propose here a new query operator called LexEQUAL, which supports multiscript matching of proper names. The operator is implemented by first transforming matches in multiscript text space into matches in the equivalent phoneme space, and then using standard approximate matching techniques to compare these phoneme strings. The algorithm incorporates tunable parameters that impact the phonetic match quality and thereby determine the match performance in the multiscript space. We evaluate the performance of the LexEQUAL operator on a real multiscript names dataset and demonstrate that it is possible to simultaneously achieve good *recall* and *precision* by appropriate parameter settings. We also show that the operator run-time can be made extremely efficient by utilizing a combination of *q-gram* and database indexing techniques. Thus, we show that the LexEQUAL operator can complement the standard lexicographic operators, representing a first step towards achieving complete multilingual functionality in database systems.

1 Introduction

The globalization of businesses and the success of mass-reach *e-Governance* solutions require database systems to store and manipulate text data in many different natural languages simultaneously. While current database systems do support the storage and management of multilingual data [13], they are not capable of *querying* or *matching* text data across languages that are in *different scripts*. For example, it is not possible to automatically match the English string *Al-Qaeda* and its equivalent strings in other scripts, say, Arabic, Greek or Chinese, even though such a feature could be immensely useful for news organizations or security agencies.

We take a first step here towards addressing this lacuna by proposing a new query operator – LexEQUAL – that matches *proper names* across different scripts, hereafter referred to as *Multiscript Matching*. Though restricted to proper names, multiscript matching nevertheless gains importance in light of the fact that *a fifth of normal text*

* A poster version of this paper appears in the Proc. of the 20th IEEE Intl. Conf. on Data Engineering, March 2004.

corpora is generic or proper names [16]. To illustrate the need for the LexEQUAL operator, consider *Books.com*, a hypothetical e-Business that sells books in different languages, with a sample product catalog as shown in Figure 1[1].

Author	Author_FN	Title	Price	Language
Descartes	René	Les Méditations Metaphysiques	€ 49.00	French
நேரு	ஜவஹர்லால்	ஆசிய ஜோதி	INR 250	Tamil
Σαρρη	Κατερινα	Παιχνίδια στο Πιάνο	€ 15.50	Greek
Nero	Bicci	The Coronation of the Virgin	$ 99.00	English
بهنسي ، د	عفيف	العمارة عبر التاريخ	SAR 75	Arabic
Nehru	Jawaharlal	Discovery of India	$ 9.95	English
寺井正博	著	秋の風 普及版	¥ 7500	Japanese
नेहरु	जवाहरलाल	भारत एक खोज	INR 175	Hindi

Fig. 1. Multilingual *Books.com*

```
select Author, Title from Books
where Author = 'Nehru'
   or Author = 'नेहरु' or Author = 'நேரு' or Author = 'Νηρυ'
```

Fig. 2. SQL:1999 **Multiscript Query**

In this environment, an SQL:1999 compliant query to retrieve all works of an author (say, `Nehru`), across multiple languages (say, in `English`, `Hindi`, `Tamil` and `Greek`) would have to be written as shown in Figure 2. This query suffers from a variety of limitations: Firstly, the user has to specify the search string `Nehru` in all the languages in which she is interested. This not only entails the user to have access to lexical resources, such as fonts and multilingual editors, in each of these languages to input the query, but also requires the user to be profi cient enough in all these languages, to provide all close variations of the query name. Secondly, given that the storage and querying of proper names is signifi cantly error-prone due to lack of dictionary support during data entry even in monolingual environments [10], the problem is expected to be much worse for multilingual environments. Thirdly, and very importantly, it would not permit a user to retrieve all the works of `Nehru`, *irrespective* of the language of publication. Finally, while selection queries involving multi-script *constants* are supported, queries involving multi-script *variables*, as for example, in join queries, cannot be expressed.

The LexEQUAL operator attempts to address the above limitations through the specifi cation shown in Figure 3, where the user has to input the name in only one language, and then either explicitly specify only the *identities* of the target match languages, or even use `*` to signify a wildcard covering *all languages* (the `Threshold` parameter in the query helps the user *fine-tune* the quality of the matched output, as discussed later in the paper). When this LexEQUAL query is executed on the database of Figure 1, the result is as shown in Figure 4.

[1] Without loss of generality, the data is assumed to be in Unicode [25] with each attribute value tagged with its language, or in an equivalent format, such as *Cuniform* [13].

```
select Author, Title from Books
where Author LexEQUAL 'Nehru' Threshold 0.25
      inlanguages { English, Hindi, Tamil, Greek }
```

Fig. 3. LexEQUAL **Query Syntax**

Author	Title	Price
Nehru	Discovery of India	$ 9.95
நேரு	ஆசிய ஜோதி	INR 250
नेहरु	भारत एक खोज	INR 175

Fig. 4. Results of LexEQUAL **Query**

Our approach to implementing the LexEQUAL operator is based on transforming the match in *character space* to a match in *phoneme space*. This phonetic matching approach has its roots in the classical Soundex algorithm [11], and has been previously used successfully in monolingual environments by the information retrieval community [28]. The transformation of a text string to its equivalent *phonemic* string representation can be obtained using common linguistic resources and can be represented in the canonical IPA format [9]. Since the phoneme sets of two languages are seldom identical, the comparison of phonemic strings is *inherently fuzzy*, unlike the standard uniscript lexicographic comparisons, making it only possible to produce a likely, but not perfect, set of answers with respect to the user's intentions. For example, the record with English name `Nero` in Figure 1, could appear in the output of the query shown in Figure 3, based on threshold value setting. Also, in theory, the answer set may not include all the answers that would be output by the equivalent (if feasible) SQL:1999 query. However, we expect that this limitation would be virtually eliminated in practice by employing high-quality Text-to-Phoneme converters.

Our phoneme space matches are implemented using standard *approximate string matching* techniques. We have evaluated the matching performance of the LexEQUAL operator on a real multiscript dataset and our experiments demonstrate that it is possible to simultaneously achieve good *recall* and *precision* by appropriate algorithmic parameter settings. Specifically, a recall of over 95 percent and precision of over 85 percent were obtained for this dataset.

Apart from output quality, an equally important issue is the *run-time* of the LexEQUAL operator. To assess this quantitatively, we evaluated our first implementation of the LexEQUAL operator as a User-Defined Function (UDF) on a commercial database system. This straightforward implementation turned out to be extremely slow – however, we were able to largely address this inefficiency by utilizing one of Q-Gram filters [6] or Phoneme Indexing [27] techniques that inexpensively weed out a large number of *false positives*, thus optimizing calls to the more expensive UDF function. Further performance improvements could be obtained by internalizing our "outside-the-server" implementation into the database engine.

In summary, we expect the phonetic matching technique outlined in this paper to effectively and efficiently complement the standard lexicographic matching, thereby representing a first step towards the ultimate objective of achieving complete multilingual functionality in database systems.

1.1 Organization of this Paper

The rest of the paper is organized as follows: The scope and issues of multiscript matching, and the support currently available, are discussed in Section 2. Our implementation of the LexEQUAL operator is presented in Section 3. The match quality of LexEQUAL operator is discussed with experimental results in Section 4. The run-time performance of LexEQUAL and techniques to improve its efficiency are discussed in in Section 5. Finally, we summarize our conclusions and outline future research avenues in Section 6.

2 Multiscript Query Processing

In multiscript matching, we consider the matching of text attributes across multiple languages arising from different scripts. We restrict our matching to attributes that contain *proper names* (such as attributes containing names of *individuals, corporations, cities, etc.*) which are assumed not to have any semantic value to the user, other than their vocalization. That is, we assume that when a name is queried for, the primary intention of the user is in retrieving all names that match *aurally*, in the specified target languages. Though restricted to proper names, multiscript matching gains importance in light of the fact that *a fifth of normal text corpora is generic or proper names* [16].

A sample multiscript selection query was shown earlier in Figure 3. The LexEQUAL operator may also be used for *equi-join* on multiscript attributes, as shown in the query in Figure 5, where all authors who have published in multiple languages are retrieved.

```
select Author from Books B1, Books B2
where B1.Author LexEQUAL B2.Author Threshold 0.25
  and B1.Language <> B2.Language
```

Fig. 5. LexEQUAL **Join Syntax**

The multiscript matching we have outlined here is applicable to many user domains, especially with regard to *e-Commerce* and *e-Governance* applications, web search engines, digital libraries and multilingual data warehouses. A real-life *e-Governance* application that requires a join based on the phonetic equivalence of multiscript data is outlined in [12].

2.1 Linguistic Issues

We hasten to add that multiscript matching of proper names is, not surprisingly given the diversity of natural languages, fraught with a variety of linguistic pitfalls, accentuated

by the attribute level processing in the database context. While simple lexicographic and accent variations may be handled easily as described in [14], issues such as language-dependent vocalizations and context-dependent vocalizations, discussed below, appear harder to resolve – we hope to address these issues in our future work.

Language-dependent Vocalizations A single text string (say, `Jesus`) could be different phonetically in different languages ("Jesus" in English and "Hesus" in Spanish). So, it is not clear when a match is being looked for, which vocalization(s) should be used. One plausible solution is to take the vocalization that is appropriate to the language in which the base data is present. But, automatic language identification is not a straightforward issue, as many languages are not uniquely identified by their associated Unicode character-blocks. With a large corpus of data, IR and NLP techniques may perhaps be employed to make this identification.

Context-dependent Vocalizations In some languages (especially, Indic), the vocalization of a set of characters is dependent on the surrounding context. For example, consider the Hindi name `Rama`. It may have different vocalizations depending on the gender of the person (pronounced as `Rāmā` for males and `Ramā` for females). While it is easy in running text to make the appropriate associations, it is harder in the database context, where information is processed at the attribute level.

2.2 State of the Art

We now briefly outline the support provided for multiscript matching in the database standards and in the currently available database engines.

While Unicode, the multilingual character encoding standard, specifies the semantics of comparison of a pair of multilingual strings at three different levels [3]: using *base characters*, *case*, or *diacritical marks*, such schemes are applicable only between strings in languages that share a common script – comparison of multilingual strings across scripts is only *binary*. Similarly, the SQL:1999 standard [8, 17] allows the specification of *collation sequences* (to correctly sort and index the text data) for individual languages, but comparison across collations is *binary*.

To the best of our knowledge, none of the commercial and open-source database systems currently support multiscript matching. Further, with regard to the specialized techniques proposed for the LexEQUAL operator, their support is as follows:

Approximate Matching Approximate matching is not supported by any of the commercial or open-source databases. However, all commercial database systems allow *User-defined Functions* (*UDF*) that may be used to add new functionality to the server. A major drawback with such addition is that UDF-based queries are not easily amenable to optimization by the query optimizer.

Phonetic Matching Most database systems allow matching text strings using pseudo-phonetic *Soundex* algorithm originally defined in [11], primarily for Latin-based scripts.

In summary, while current databases are effective and efficient for monolingual data (that is, within a collation sequence), they do not currently support processing multilingual strings across languages in any unified manner.

2.3 Related Research

To our knowledge, the problem of matching multiscript strings has not been addressed previously in the database research literature. Our use of a phonetic matching scheme for multiscript strings is inspired by the successful use of this technique in the *mono-script* context by the information retrieval and pharmaceutical communities. Specifically, in [23] and [28], the authors present their experience in phonetic matching of uniscript text strings, and provide measures on correctness of matches with a suite of techniques. Phonemic searches have also been employed in pharmaceutical systems such as [15], where the goal is to find *look-alike sound-alike* (*LASA*) drug names.

The approximate matching techniques that we use in the phonetic space are being actively researched and a large body of relevant literature is available (see [19] for a comprehensive survey). We use the well known *dynamic programming* technique for approximate matching and the standard *Levenshtein* edit-distance metric to measure the *closeness* of two multiscript strings in the phonetic space. The dynamic programming technique is chosen for its flexibility in simulating a wide range of different edit distances by appropriate parameterization of the cost functions.

Apart from being multiscript, another novel feature of our work is that we not only consider the *match quality* of the LexEQUAL operator (in terms of recall and precision) but also quantify its *run-time efficiency* in the context of a commercial state-of-the-art database system. This is essential for establishing the viability of multilingual matching in online e-commerce and e-governance applications. To improve the efficiency of LexEQUAL, we resort to Q-Gram filters [6], which have been successfully used recently for approximate matches in monolingual databases to address the problem of names that have many variants in spelling (example, `Cathy` and `Kathy` or variants due to input errors, such as `Catyh`). We also investigate the phonetic indexes to speed up the match process – such indexes have been previously considered in [27] where the phonetic closeness of English lexicon strings is utilized to build simpler indexes for text searches. Their evaluation is done with regard to in-memory indexes, whereas our work investigates the performance for persistent on-disk indexes. Further, we extend these techniques to multilingual domains.

3 LexEQUAL: Multiscript Matching Operator

In this section, we first present the strategy that we propose for matching multilingual strings, and then detail our multiscript matching algorithm.

3.1 Multiscript Matching Strategy

Our view of ontology of text data storage in database systems is shown in Figure 6. The semantics of *what* gets stored is outlined in the top part of the figure, and *how* the information gets stored in the database systems is provided by the bottom part of the figure. The important point to note is that a *proper name*, which is being stored currently as a character string (shown by the dashed line) may be complemented with a phoneme string (shown by the dotted line), that can be derived on demand, using standard linguistic resources, such as *Text-To-Phoneme (TTP)* converters.

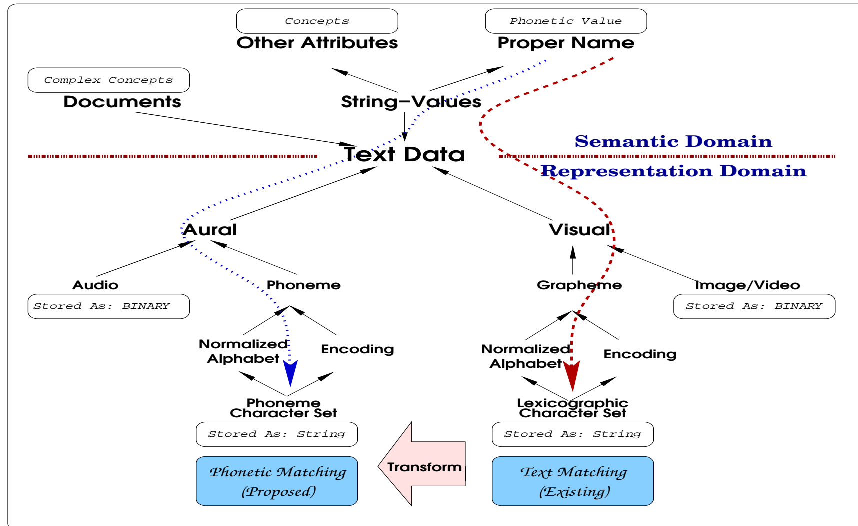

Fig. 6. Ontology for Text Data

As mentioned earlier, we assume that when a name is queried for, the primary intention of the user is in retrieving all names that match *aurally*, irrespective of the language. Our strategy is to capture this intention by matching the equivalent *phonemic* strings of the multilingual strings. Such phoneme strings represent a normalized form of proper names across languages, thus providing a means of comparison. Further, when the text data is stored in multiple scripts, this may be the *only* means for comparing them. We propose complementing and enhancing the standard lexicographic equality operator of database systems with a matching operator that may be used for approximate matching of the equivalent phonemic strings. Approximate matching is needed due to the inherent fuzzy nature of the representation and due to the fact that phoneme sets of different languages are seldom identical.

3.2 LexEQUAL Implementation

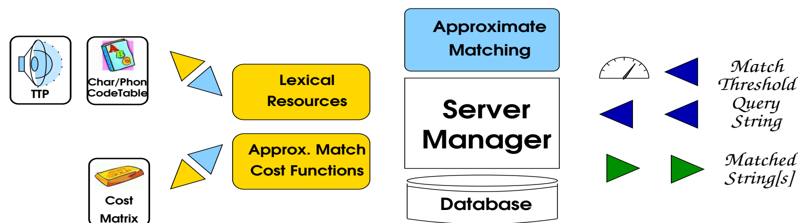

Fig. 7. Architecture

Our implementation for querying multiscript data is shown as shaded boxes in Figure 7. Approximate matching functionality is added to the database server as a UDF.

Lexical resources (e.g., script and IPA code tables) and relevant TTP converters that convert a given language string to its equivalent phonemes in IPA alphabet are integrated with the query processor. The cost matrix is an installable resource intended to tune the quality of match for a specific domain.

Ideally the LexEQUAL operator should be implemented inside the database engine for optimum performance. However, as a pilot study and due to lack of access to the internals in the commercial database systems, we have currently implemented LexEQUAL as a *user-defined function (UDF)* that can be called in SQL statements. As shown later in this paper, even such an *outside-the-server* approach can, with appropriate optimizations, be engineered to provide viable performance. A related advantage is that LexEQUAL can be easily integrated with current systems and usage semantics while the more involved transition to an inside-the-engine implementation is underway.

3.3 LexEQUAL Matching Algorithm

LexEQUAL (S_l, S_r, e)
Input: Strings S_l, S_r, *Match Threshold* e
Languages with IPA transformations, $\mathcal{S_L}$ (*as global resource*)
Output: `TRUE`, `FALSE` or `NORESOURCE`
1. $L_l \leftarrow$ Language of S_l;$L_r \leftarrow$ Language of S_r;
2. **if** $L_l \in \mathcal{S_L}$ and $L_r \in \mathcal{S_L}$ **then**
3. $T_l \leftarrow$`transform`(S_l,L_l); $T_r \leftarrow$`transform`(S_r,L_r);
4. $Smaller \leftarrow (\mid T_l \mid \leq \mid T_r \mid ? |T_l| : |T_r|)$;
5. **if** `editdistance`$(T_l, T_r) \leq (e * Smaller)$ **then**
return `TRUE` **else return** `FALSE`;
6. **else return** `NORESOURCE`;

editdistance(S_L, S_R)
Input: String S_L, String S_R
Output: Edit-distance k
1. $L_l \leftarrow \mid S_L \mid$; $L_r \leftarrow \mid S_R \mid$;
2. Create *DistMatrix*$[L_l, L_r]$ and initialize to $Zero$;
3. **for** i **from** 0 **to** L_l **do** *DistMatrix*$[i, 0] \leftarrow i$;
4. **for** j **from** 0 **to** L_r **do** *DistMatrix*$[0, j] \leftarrow j$;
5. **for** i **from** 1 **to** L_l **do**
6. **for** j **from** 1 **to** L_r **do**
7. $$DistMatrix[i, j] \leftarrow \textbf{Min} \begin{Bmatrix} DistMatrix[i-1, j] + InsCost(S_{L_i}) \\ DistMatrix[i-1, j-1] + SubCost(S_{R_j}, S_{L_i}) \\ DistMatrix[i, j-1] + DelCost(S_{R_j}) \end{Bmatrix}$$
8. **return** *DistMatrix*$[L_l, L_r]$;

Fig. 8. The LexEQUAL Algorithm

The LexEQUAL algorithm for comparing multiscript strings is shown in Figure 8. The operator accepts two multilingual text strings and a match threshold value as input. The strings are first transformed to their equivalent phonemic strings and the edit distance between them is then computed. If the edit distance is less than the threshold value, a positive match is flagged.

The `transform` function takes a multilingual string in a given language and returns its phonetic representation in IPA alphabet. Such transformation may be easily implemented by integrating standard TTP systems that are capable of producing phonetically equivalent strings. The `editdistance` function [7] takes two strings and returns the *edit distance* between them. A *dynamic programming* algorithm is implemented for this computation, due to, as mentioned earlier, the flexibility that it offers in experimenting with different cost functions.

Match Threshold Parameter A user-settable parameter, *Match Threshold*, as a fraction between `0` and `1`, is an additional input for the phonetic matching. This parameter specifi es the user tolerance for approximate matching: `0` signifi es that only perfect matches are accepted, whereas a positive threshold specifi es the allowable error (that is, edit distance) as the fraction of the size of query string. The appropriate value for the threshold parameter is determined by the requirements of the application domain.

Intra-Cluster Substitution Cost Parameter The three cost functions in Figure 8, namely *InsCost, DelCost* and *SubsCost*, provide the costs for inserting, deleting and substituting characters in matching the input strings. With different cost functions, different flavors of edit distances may be implemented easily in the above algorithm. We support a *Clustered Edit Distance* parameterization, by extending the *Soundex* [11] algorithm to the phonetic domain, under the assumptions that clusters of like phonemes exist and a substitution of a phoneme from within a cluster is more acceptable as a match than a substitution from across clusters. Hence, near-equal phonemes are clustered, based on the similarity measure as outlined in [18], and the substitution cost within a cluster is made a tunable parameter, the *Intra-Cluster Substitution Cost*. This parameter may be varied between `0` and `1`, with `1` simulating the standard *Levenshtein* cost function and lower values modeling the *phonetic proximity* of the like-phonemes. In addition, we also allow user customization of clustering of phonemes.

4 Multiscript Matching Quality

In this section, we fi rst describe an experimental setup to measure the quality (in terms of precision and recall) of the LexEQUAL approach to multiscript matching, and then the results of a representative set of experiments executed on this setup. Subsequently, in Section 5, we investigate the run-time effi ciency of the LexEQUAL operator.

4.1 Data Set

With regard to the datasets to be used in our experiments, we had two choices: experiment with multilingual lexicons and verify the match quality by *manual relevance judgement*, or alternatively, experiment with tagged multilingual lexicons (that is, those in which the expected matches are marked beforehand) and verify the quality mechanically. We chose to take the second approach, but because no tagged lexicons of multiscript names were readily available[2], we created our own lexicon from existing monolingual ones, as described below.

[2] Bi-lingual dictionaries mark *semantically*, and not *phonetically*, similar words.

We selected proper names from three different sources so as to cover common names in English and Indic domains. The first set consists of randomly picked names from the *Bangalore Telephone Directory*, covering most frequently used Indian names. The second set consists of randomly picked names from the *San Francisco Physicians Directory*, covering most common American first and last names. The third set consisting of generic names representing Places, Objects and Chemicals, was picked from the *Oxford English Dictionary*. Together the set yielded about 800 names in English, covering three distinct name domains. Each of the names was hand converted to two Indic scripts – Tamil and Hindi. As the Indic languages are phonetic in nature, conversion is fairly straight forward, barring variations due to the mismatch of phoneme sets between English and the Indic languages. All phonetically equivalent names (but in different scripts) were manually tagged with a common *tag-number*. The tag-number is used subsequently in determining quality of a match – any match of two multilingual strings is considered to be correct if their tag-numbers are the same, and considered to be a *false-positive* otherwise. Further, the fraction of *false-dismissals* can be easily computed since the expected set of correct matches is known, based on the tag-number of a given multilingual string.

To convert English names into corresponding phonetic representations, standard linguistic resources, such as the *Oxford English Dictionary* [22] and TTP converters from [5], were used. For Hindi strings, *Dhvani* TTP converter [4] was used. For Tamil strings, due to the lack of access to any TTP converters, the strings were hand-converted, assuming phonetic nature of the Tamil language. Further those symbols specific to speech generation, such as the supra-segmentals, diacritics, tones and accents were removed. Sample phoneme strings for some multiscript strings are shown in Figure 9.

Lexicographic String	Language	Phonetic Representation (*in IPA*)
University	English	junəvɜrsɪti
நேரு	Tamil	neiru
École	French	eikøl
இந்தியா	Tamil	ɪndɪyɑ
हैड्रोजन	Hindi	haɪdrədʒən
Espanõl	Spanish	ɛspɑnjøl

Fig. 9. Phonemic Representation of Test Data

The frequency distribution of the data set with respect to string length is shown in Figure 10, for both lexicographic and (generated) phonetic representations. The set had an average lexicographic length of 7.35 and an average phonemic length of 7.16. Note that though Indic strings are typically visually much shorter as compared to English strings, their character lengths are similar owing to the fact that most Indic characters are composite glyphs and are represented by multiple Unicode characters.

We implemented a prototype of LexEQUAL on top of the *Oracle 9i (Version 9.1.0)* database system. The multilingual strings and their phonetic representations (in IPA alphabet) were both stored in Unicode format. The algorithm shown in Figure 8 was implemented, as a UDF in the PL/SQL language.

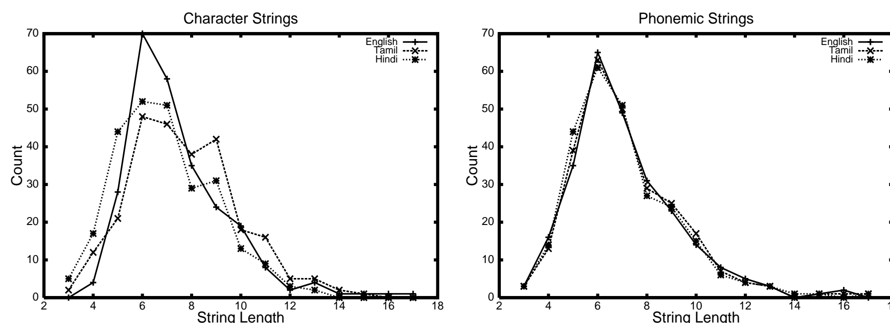

Fig. 10. Distribution of Multiscript Lexicon (for Match Quality Experiments)

4.2 Performance Metrics

We ran multiscript selection queries (as shown in Figure 3). For each query, we measured two metrics – *Recall*, defined as *the fraction of correct matches that appear in the result*, and *Precision*, defined as *the fraction of the delivered results that are correct*. The recall and the precision figures were computed using the following methodology: We matched each phonemic string in the data set with every other phonemic string, counting the number of matches (m_1) that were correctly reported (that is, the tag-numbers of multiscript strings being matched are the same), along with the total number of matches that are reported as the result (m_2). If there are n equivalent groups (with the same tag-number) with n_i of multiscript strings each (note that both n and n_i are known during the tagging process), the *precision* and *recall* metrics are calculated as follows:

$$Recall = m_1 \Big/ \sum_{i=1}^{n} ({}^{n_i}C_2) \quad \text{and} \quad Precision = m_1 \Big/ m_2$$

The expression in the denominator of *recall* metric is the ideal number of matches, as every pair of strings (*i.e.,* ${}^{n_i}C_2$) with the same tag-number must match. Further, for an ideal answer for a query, both the metrics should be `1`. Any deviation indicates the inherent fuzziness in the querying, due to the differences in the phoneme set of the languages and the losses in the transformation to phonemic strings. Further, the two query input parameters – *user match threshold* and *intracluster substitution cost* (explained in Section 3.3) were varied, to measure their effect on the quality of the matches.

4.3 Experimental Results

We conducted our multiscript matching experiments on the lexicon described above. The plots of the *recall* and *precision* metrics against *user match threshold*, for various *intracluster substitution costs*, between `0` and `1`, are provided in Figure 11.

The curves indicate that the recall metric improves with increasing user match threshold, and asymptotically reaches perfect recall, after a value of `0.5`. An interesting point to note is that the recall gets better with reducing intracluster substitution costs, validating the assumption of the *Soundex* algorithm [11].

In contrast to the recall metric, and as expected, the precision metric drops with increasing threshold – while the drop is negligible for threshold less than `0.2`, it is rapid

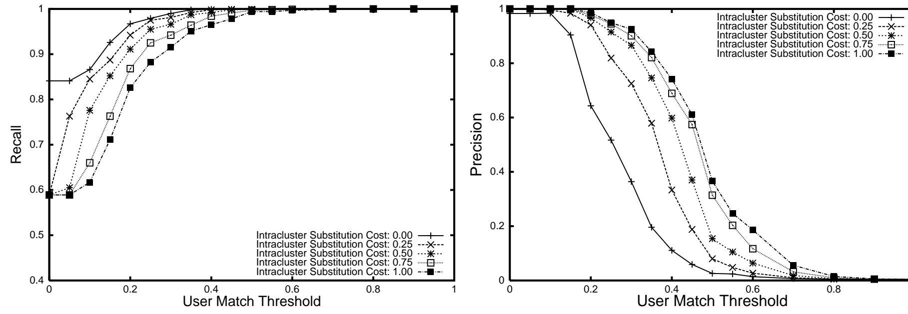

Fig. 11. Recall and Precision Graphs

in the range `0.2` to `0.5`. It is interesting to note that with an intracluster substitution cost of `0`, the precision drops very rapidly at a user match threshold of `0.1` itself. That is, the *Soundex* method, which is good in recall, is very ineffective with respect to precision, as it introduces a large number of false-positives even at low thresholds.

Selection of Ideal Parameters for Phonetic Matching Figure 12 shows the *precision-recall* curves, with respect to each of the query parameters, namely, *intracluster substitution cost* and *user match threshold*. For the sake of clarity, we show only the plots corresponding to the costs `0`, `0.5` and `1`, and plots corresponding to thresholds `0.2`, `0.3` and `0.4`. The top-right corner of the precision-recall space corresponds to a perfect match and the closest points on the precision-recall graphs to the top-right corner correspond to the query parameters that result in the best match quality. As can be seen from Figure 12, the best possible matching is achieved by a substitution cost between `0.25` and `0.5`, and for thresholds between `0.25` and `0.35`, corresponding to the knee regions of the respective curves. With such parameters, the *recall* is ≈95%, and *precision* is ≈85%. That is, ≈5% of the real matches would be *false-dismissals*, and about ≈15% of the results are *false-positives*, which must be discarded by post-processing, using non-phonetic methods.

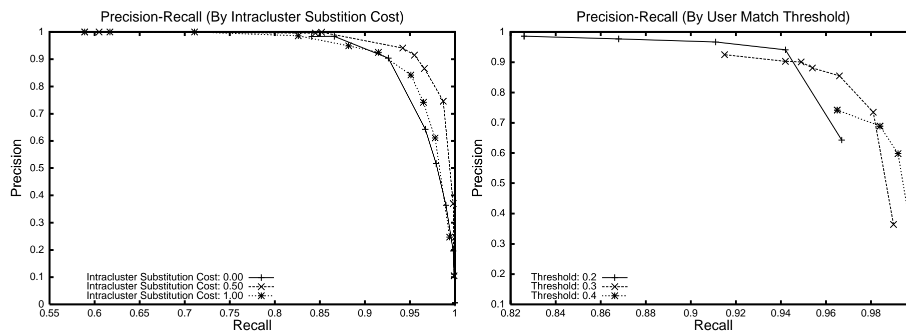

Fig. 12. Precision-Recall Graphs

We also would like to emphasize that quality of approximate matching depends on phoneme sets of languages, the accuracy of the phonetic transformations, and more importantly, on the data sets themselves. Hence the matching needs to be tuned as outlined in this section, for specific application domains. In our future work, we plan to investigate techniques for automatically generating the appropriate parameter settings based on dataset characteristics.

5 Multiscript Matching Efficiency

In this section, we analyze the query processing efficiency using the LexEQUAL operator. Since the real multiscript lexicon used in the previous section was not large enough for performance experiments, we synthetically generated a large dataset from this multiscript lexicon. Specifically, we concatenated each string with all remaining strings *within a given language*. The generated set contained about 200,000 names, with an average lexicographic length of 14.71 and average phonemic length of 14.31. The Figure 13 shows the frequency distribution of the generated data set – in both character and (generated) phonetic representations with respect to string lengths.

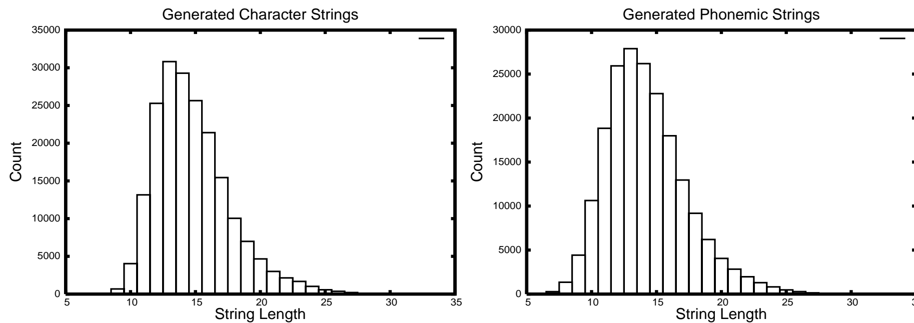

Fig. 13. Distribution of Generated Data Set (for Performance Experiments)

5.1 Baseline LexEQUAL Runs

To create a baseline for performance, we first ran the selection and equi-join queries using the LexEQUAL operator (samples shown in Figures 3 and 5), on the large generated data set. Table 1 shows the performance of the native equality operator (for exact matching of character strings) and the LexEQUAL operator (for approximate matching of phonemic strings), for these queries[3]. The performance of the standard database equality operator is shown only to highlight the inefficiency of the approximate matching operator. As can be seen clearly, the UDF is orders of magnitude slower compared with native database equality operators. Further, the optimizer chose a *nested-loop* technique for the join query, irrespective of the availability of indexes or optimizer hints, indicating that no optimization was done on the UDF call in the query.

[3] The join experiment was done on a 0.2% subset of the original table, since the full table join using UDF took about 3 days.

Query	Matching Methodology	Time
Scan	*Exact (= Operator)*	0.59 *Sec*
Scan	*Approximate (*LexEQUAL *UDF)*	1418 *Sec*
Join	*Exact (= Operator)*	0.20 *Sec*
Join	*Approximate (*LexEQUAL *UDF)*	4004 *Sec*

Table 1. Relative Performance of Approximate Matching

To address the above problems and to improve the efficiency of multiscript matching with LexEQUAL operator, we implemented two alternative optimization techniques, *Q-Grams* and *Phonetic Indexes*, described below, that cheaply provide a candidate set of answers that is checked for inclusion in the result set by the accurate but expensive LexEQUAL UDF. These two techniques exhibit different quality and performance characteristics, and may be chosen depending on application requirements.

5.2 Q-Gram Filtering

We show here that *Q-Grams*[4], which are a popular technique for approximate matching of standard text strings [6], are applicable to phonetic matching as well.

The database was first augmented with a table of *positional q-grams* of the original phonemic strings. Subsequently, the three filters described below, namely *Length* filter that depends only on the length of the strings, and *Count* and *Position* filters that use special properties of q-grams, were used to filter out a majority of the non-matches using standard database operators only. Thus, the filters weed out most non-matches cheaply, leaving the accurate, but expensive LexEQUAL UDF to be invoked (to weed out *false-positives*) on a vastly reduced candidate set.

Length Filter leverages the fact that strings that are within an edit distance of k cannot differ in length, by more than k. This filter does not depend on the q-grams.

Count Filter ensures that the number of matching q-grams between two strings σ_1 and σ_2 of lengths $|\sigma_1|$ and $|\sigma_2|$, must be at least $(max(|\sigma_1|, |\sigma_2|) - 1 - (k-1) * q)$, a necessary condition for two strings to be within an *edit-distance* of k.

Position Filter ensures that a positional q-gram of one string does not get matched to a positional q-gram of the second that differs from it by more than k positions.

A sample SQL query using q-grams is shown in Figure 14, assuming that the query string is transformed into a record in table Q, and the auxiliary q-gram table of Q is

[4] Let σ be a string of size n in a given alphabet Σ and $\sigma[i,j], 1 \leq i \leq j \leq n$, denote a substring starting at position i and ending at position j of σ. A *Q-gram* of σ is a substring of σ of length q. A *Positional Q-gram* of a string σ is a pair $(i, \sigma_{extended}[i, i+q-1])$ where $\sigma_{extended}$ is the augmented string of σ, which is appended with (q-1) start symbols(say, ◁) and (q-1) end symbols (say, ▷), where the start and end symbols are not in the original alphabet. For example, a string `LexEQUAL` will have the following *positional q-grams*: $\{$(1,◁◁L), (2, ◁Le), (3, Lex), (4, exE), (5, xEQ), (6, EQU), (7, QUA), (8, UAL), (9, AL▷), (10, L▷▷)$\}$.

```
SELECT N.ID, N.Name
FROM Names N, AuxNames AN, Query Q, AuxQuery AQ
WHERE N.ID = AN.ID
    AND Q.ID = AQ.ID
    AND AN.Qgram = AQ.Qgram
    AND |len(N.PName) − len(Q.str)| ≤ e ∗ length(Q.str)
    AND |AN.Pos − AQ.Pos| ≤ (e ∗ length(Q.str))
GROUP BY N.ID, N.PName
HAVING count(*) ≥ (len(N.PName) − 1 − ((e ∗ len(Q.str) − 1) ∗ q))
    AND LexEQUAL(N.PName, Q.str, e)
```

Fig. 14. SQL using *Q-Gram* Filters

created in AQ. The *Length Filter* is implemented in the fourth condition of the SQL statement, the *Position Filter* by the fifth condition, and the *Count Filter* by the `GROUP BY/HAVING` clause. As can be noted in the above SQL expression, the *UDF* function, LexEQUAL, is called at the end, *after* all three filters have been utilized.

Query	Matching Methodology	Time
Scan	LexEQUAL *UDF + q-gram filters*	13.5 *Sec*
Join	LexEQUAL *UDF + q-gram filters*	856 *Sec*

Table 2. Q-Gram Filter Performance

The performance of the selection and equi-join queries, after including the Q-gram optimization, are given in Table 2. Comparing with figures in Table 1, the use of this optimization improves the *selection* query performance by an order of magnitude and the *join* query performance by five-fold. The improvement in join performance is not as dramatic as in the case of scans, due to the additional joins that are required on the large q-gram tables. Also, note that the performance improvements are not as high as those reported in [6], perhaps due to our use of a standard commercial database system and the implementation of LexEQUAL using slow dynamic programming algorithm in an interpreted PL/SQL language environment.

5.3 Phonetic Indexing

We now outline a *phonetic indexing* technique that may be used for accessing the *near-equal* phonemic strings, using a standard database index. We exploit the following two facts to build a compact database index: First, the substitutions of *like* phonemes keeps the recall high (refer to Figure 11), and second, phonemic strings may be transformed into smaller numeric strings for indexing as a database number. However, the down-side of this method is that it suffers from a drop in recall (that is, *false-dismissals* are introduced).

To implement the above strategy, we need to transform the phoneme strings to a number, such that phoneme strings that are *close* to each other map to the same number. For this, we used a modified version of the *Soundex* algorithm [11], customized to the phoneme space: We first grouped the phonemes into equivalent clusters along the lines outlined in [18], and assigned a unique number to each of the clusters. Each phoneme string was transformed to a unique numeric string, by concatenating the cluster identifiers of each phoneme in the string. The numeric string thus obtained was converted into an integer – *Grouped Phoneme String Identifier* – which is stored along with the phoneme string. A standard database B-Tree index was built on the grouped phoneme string identifier attribute, thus creating a compact index structure using only integer datatype.

For a LexEQUAL query using phonetic index, we first transform the operand multiscript string to its phonetic representation, and subsequently to its grouped phoneme string identifier. The index on the grouped phoneme string identifier of the lexicon is used to retrieve all the candidate phoneme strings, which are then tested for a match invoking the LexEQUAL UDF with the user specified match tolerance. The invocation of the LexEQUAL operator in a query maps into an internal query that uses the phonetic index, as shown in Figure 15 for a sample join query. Note that any two strings that match in the above scheme are *close phonetically*, as the differences between individual phonemes are from only within the pre-defined cluster of phonemes. Any changes across the groups will result in a non-match. Also, it should be noted that those strings that are within the classical definition of *edit-distance*, but with substitutions across groups, will not be reported, resulting in *false-dismissals*. While some of such false-dismissals may be corrected by a more robust design of phoneme clusters and cost functions, not all *false-dismissals* can be corrected in this method.

```
SELECT N.ID, N.Name
FROM Names N, Query Q
WHERE N.GroupedPhonStringID = Q.GroupedPhonStringID
    AND LexEQUAL(N.PName, Q.PName, e)
```

Fig. 15. SQL using Phonetic Indexes

We created an index on the grouped phoneme string identifier attribute and re-ran the same selection and equi-join queries on the large synthetic multiscript dataset. The LexEQUAL operator is modified to use this index, as shown in the SQL expression in Figure 15, and the associated scan and join performance is given in Table 3.

Query	Matching Methodology	Time
Scan	LexEQUAL *UDF + phonetic index*	0.71 *Sec*
Join	LexEQUAL *UDF + phonetic index*	15.2 *Sec*

Table 3. Phonemic Index Performance

While the performance of the queries with phonetic index is an order of magnitude better than that achieved with q-gram technique, the phonetic index introduces a small, but significant 4 - 5% *false-dismissals*, with respect to the classical edit-distance metric. A more robust grouping of like phonemes may reduce this drop in quality, but may not nullify it. Hence, the phonetic index approach may be suitable for applications which can tolerate false-dismissals, but require a very fast response times (such as, web search engines).

6 Conclusions and Future Research

In this paper we specified a multilingual text processing requirement – *Multiscript Matching* – that has a wide range of applications from *e-Commerce* applications to search engines to multilingual data warehouses. We provided a survey of the support provided by SQL standards and current database systems. In a nutshell, multiscript processing is not supported in any of the database systems.

We proposed a strategy to solve the multiscript matching problem, specifically for proper name attributes, by transforming matching in the *lexicographic space* to the equivalent *phonetic space*, using standard linguistic resources. Due to the inherent fuzzy nature of the phonetic space, we employ approximate matching techniques for matching the transformed phonemic strings. Currently, we have implemented the multiscript matching operator as a UDF, for our initial pilot implementation. We confirmed the feasibility of our strategy by measuring the quality metrics, namely *Recall* and *Precision*, in matching a real, tagged multilingual data set. The results from our initial experiments on a representative multiscript nouns data set, showed good recall (≈95%) and precision (≈85%), indicating the potential of such an approach for practical query processing. We also showed how the parameters may be tuned for optimal matching for a given dataset. Further, we showed that the poor performance associated with the UDF implementation of approximate matching may be improved significantly, by employing one of the two alternate methods: the *Q-Gram* technique, and a *Phonemic Indexing* technique. These two techniques exhibit different quality and performance characteristics, and may be chosen depending on the requirements of an application. However, both the techniques, as we have demonstrated, are capable of improving the multiscript matching performance by orders of magnitude.

Thus, we show that the LexEQUAL operator outlined in this paper is effective in multiscript matching, and can be made efficient as well. Such an operator may prove to be a valuable first step in achieving full multilingual functionality in database systems.

In our future work, we plan to investigate techniques for automatically generating the optimal matching parameters, based on a given dataset, its domain and a training set. Also, we plan to explore extending the approximate indexing techniques outlined in [1, 21] for creating a metric index for phonemes. We are working on an *inside-the-engine* implementation of LexEQUAL on an open-source database system, with the expectation of further improving the runtime efficiency.

Acknowledgements This work was partially supported by a Swarnajayanti Fellowship from the Department of Science and Technology, Government of India.

References

1. R. Baeza-Yates and G. Navarro. Faster Approximate String Matching. *Algorithmica*, Vol 23(2):127-158, 1999.
2. E. Chavez, G. Navarro, R. Baeza-Yates and J. Marroquin. Searching in Metric Space. *ACM Computing Surveys*, Vol 33(3):273-321, 2001.
3. M. Davis. Unicode collation algorithm. *Unicode Consortium Technical Report*, 2001.
4. Dhvani - A Text-to-Speech System for Indian Languages. *http://dhvani.sourceforge.net/*.
5. The Foreign Word – The Language Site, Alicante, Spain. *http://www.ForeignWord.com*.
6. L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan and D. Srivastava. Approximate String Joins in a Database (almost) for Free. *Proc. of 27th VLDB Conf.*, September 2001.
7. D. Gusfield. Algorithms on Strings, Trees and Sequences. *Cambridge University Press*, 2001.
8. International Organization for Standardization. ISO/IEC 9075-1-5:1999, Information Technology – Database Languages – SQL (parts 1 through 5). 1999.
9. The International Phonetic Association. Univ. of Glasgow, Glasgow, UK. *http://www.arts.gla.ac.uk/IPA/ipa.html*.
10. D. Jurafskey and J. Martin. Speech and Language Processing. *Pearson Education*, 2000.
11. D. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. *Addison-Wesley*, 1993.
12. A. Kumaran and J. Haritsa. On Database Support for Multilingual Environments. *Proc. of 9th IEEE RIDE Workshop*, March 2003.
13. A. Kumaran and J. Haritsa. On the Costs of Multilingualism in Database Systems. *Proc. of 29th VLDB Conference*, September 2003.
14. A. Kumaran and J. Haritsa. Supporting Multilexical Matching in Database Systems. *DSL/SERC Technical Report TR-2004-01*, 2004.
15. B. Lambert, K. Chang and S. Lin. Descriptive analysis of the drug name lexicon. *Drug Information Journal*, Vol 35:163-172, 2001.
16. M. Liberman and K. Church. Text Analysis and Word Pronunciation in TTS Synthesis. *Advances in Speech Processing*, 1992.
17. J. Melton and A. Simon. SQL 1999: Understanding Relational Language Components. *Morgan Kaufmann*, 2001.
18. P. Mareuil, C. Corredor-Ardoy and M. Adda-Decker. Multilingual Automatic Phoneme Clustering. *Proc. of 14th Intl. Congress of Phonetic Sciences*, August 1999.
19. G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, Vol 33(1):31-88, 2001.
20. G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio. Indexing Text with Approximate q-grams. *Proc. of 11th Combinatorial Pattern Matching Conf.*, June 2000.
21. G. Navarro, R. Baeza-Yates, E. Sutinen and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, Vol 24(4):19-27, 2001.
22. The Oxford English Dictionary. *Oxford University Press*, 1999.
23. U. Pfeifer, T. Poersch and N. Fuhr. Searching Proper Names in Databases. *Proc. Conf. Hypertext-Information Retrieval-Multimedia*, April 1995.
24. L. Rabiner and B. Juang. *Fundamentals of Speech Processing*. Prentice Hall, 1993.
25. The Unicode Consortium. The Unicode Standard. *Addison-Wesley*, 2000.
26. The Unisyn Project. The Center for Speech Technology Research, Univ. of Edinburgh, United Kingdom. *http://www.cstr.ed.ac.uk/projects/unisyn/*.
27. J. Zobel and P. Dart. Finding Approximate Matches in Large Lexicons. *Software – Practice and Experience*, Vol 25(3):331-345, March, 1995.
28. J. Zobel and P. Dart. Phonetic String Matching: Lessons from Information Retrieval. *Proc. of 19th ACM SIGIR Conf.*, August 1996.