SUPPORTING MULTI-LEXICAL MATCHING

IN DATABASE SYSTEMS

A. Kumaran Jayant R. Haritsa

Technical Report
TR-2004-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

`http://dsl.serc.iisc.ernet.in`

Supporting Multilexical Matching in Database Systems *

A. Kumaran Jayant R. Haritsa
Department of Computer Science and Automation
Indian Institute of Science, Bangalore, INDIA

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 languages in different scripts. As a first step towards addressing this lacuna, we propose here a new operator called LexEQUAL*, which supports multi-script matching of proper names. The operator is implemented by first transforming matches in multi-script 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 phonemic match quality and thereby determine the match performance in the multi-script space. We evaluate the performance of the* LexEQUAL *operator on a large multi-script name 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 rapidly accelerating 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 the current database systems do support the storage and management of multilingual data [15], they are not capable of *querying* or matching text data across languages that are in *different scripts*. For example, it is not possible to compare the name "Al Qaeda" in English and the same name written in other scripts, like Arabic, Hindi or Korean, automatically, thereby hampering the work of security agencies.

We take a first step here towards addressing the above lacuna by proposing a new database operator called LexEQUAL, which supports matching of *proper names* across language scripts, hereafter referred to as "multi-script matching". To illustrate this operator, consider a hypothetical e-Business, *Books.com*

*Indian Institute of Science Technical Report, TR-2004-01 DSL/SERC, January 2004.

that sells books in different languages, with a sample product catalog as shown in Figure 1. Without loss of generality, we assume that the data is stored in Unicode [29] character set with each attribute value tagged explicitly with the language or in an equivalent format, such as *Cuniform* [15].

Author	Author_FN	Title	Price	Language
Descartes	René	Les Méditations Metaphysiques	€ 9.00	French
நாராயண்	R.K.	சுவாமியும் கினேகிதர்களும்	INR 250	Tamil
ನಾರಾಯಣ್	R.K.	ಮಾಲ್ಗುಡಿ ಡೇಸ್	INR 155	Kannada
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

Figure 1. Multilingual *Books.com*

In such an environment, we propose a new database operator, LexEQUAL that can match `Nehru` across the Indic languages which are spread across over a dozen scripts, with the following SQL query syntax:

```
select * from Books
where Author LexEQUAL 'Nehru'
      inlanguages { Hindi, Bengali, Tamil, ...}
```

Multi-script matching of proper names gains importance in light of the fact that *a fifth of normal text corpora is generic or proper names* [17]. This matching is fraught with a variety of linguistic pitfalls as explained in detail later in the paper – for example, the textual string "Rama" in English maps to two different names in Indic languages: `Rāma` or `Rama`, and it is not clear whether one or the other, or both, are to be considered as matches. Since the phoneme sets of two languages are seldom identical, a string in one language may equate to multiple strings in the second. Therefore, multi-script comparisons are *inherently fuzzy* unlike the standard uni-script lexicographic database comparisons, making it only possible to produce a likely, but not perfect, set of answers with respect to the user's intentions. That is, the metrics of *precision* and *recall* typically associated with information retrieval, also come into play in the multi-script database context.

Our approach to implementing the LexEQUAL operator is based on transforming each text string in the database, to its equivalent *phonetic* string representation. This phonetic string can be obtained using common linguistic resources (such as, dictionaries [22] or Text-to-Speech (TTS) engines [7]) and can be represented in the canonical IPA format [11]. Note that the phonetic strings may be either computed on-the-fly to save space, or stored persistently to save time, providing a classic space-performance tradeoff.

With the above framework, a match between two multi-script strings is tested by matching the corresponding phonetic strings. That is, the matching in *character space* is transformed into *a match in phoneme space*. This phonemic matching approach has its roots in the classical Soundex algorithm of Knuth [13], and has been previously used successfully in monolingual environments by the information retrieval community [32]. Our phoneme space matches are implemented using standard *approximate string matching* techniques to cater to the inherent fuzziness of multilingual matching mentioned above, and reflect the natural clustering that exists among phonemes. Further, the algorithm incorporates tunable parameters that impact the phonemic match quality and thereby determine the match performance in the name space.

We have evaluated the matching performance of the LexEQUAL operator on a multi-script telephone subscriber name dataset. Our experiments demonstrate that it is possible to simultaneously achieve good recall and precision by appropriate parameter settings. Specifically, a recall of over 95 percent and precision of over 85 percent were obtained for this dataset. In our future work we plan to investigate techniques for automatically generating the appropriate parameter settings based on dataset characteristics.

Apart from output quality, an equally important issue is the run-time of the LexEQUAL operator. To assess this quantitatively, we first implemented the LexEQUAL operator on a commercial database system through User-Defined Functions (UDF). This straightforward implementation turned out to be extremely slow – however, we were able to largely address this inefficiency by utilizing a combination of Q-Gram filters [9] and phoneme indexing [31] techniques that limit the number of strings passed to the expensive UDF function. We present experimental numbers to quantitatively demonstrate the effect of these improvements.

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 Related Research

To our knowledge, the problem of matching multi-script strings has not been addressed previously in the database research literature. Our use of a phonetic matching scheme for multi-script strings is inspired by the successful use of this technique in the *mono-script* context by the information retrieval and pharmaceutical communities. Specifically, phonetic retrieval is discussed in [23] and [32], where the authors present their experience in phonetic matching of text strings, and provide measures on correctness of matches with a suite of techniques. Phonetic searches have also been employed in pharmaceutical systems such as [16], where the goal is to find "look-alike sound-alike" drug names.

Apart from being multi-script, another novel feature of our work is that we not only consider the output quality of the LexEQUAL operator 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.

The approximate matching techniques that we use in the phonemic space have been selected from the large body of literature available on approximate matching, most of which has arisen from the computer science theory community (see [20] for a comprehensive survey of these techniques). Specifically, we use edit-distance in the phonemic space as the evaluation metric for a match, with the cost matrix for the various edit operations (insertion, substitution, and deletion) reflecting the natural clustering that exists in the phonemic domain.

To improve the efficiency of LexEQUAL, we resort to Q-Gram filters [9], 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 use phonetic indexes to speed up the match process – such indexes have been previously considered in [31] 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.

1.2 Organization of this Paper

The rest of the paper is organized as follows: The multi-script query processing problem and the associated linguistic issues are discussed in Section 2. Our implementation of the LexEQUAL operator is presented in. Section 3. The experimental setup and results for the LexEQUAL match quality are highlighted in Section 4. Techniques for improving the run-time efficiency of LexEQUAL are outlined and evaluated in Section 5. Finally, we summarize our conclusions and outline future research avenues in Section 6.

2 Multi-script Query Processing

In this section, we outline the scope of the multi-script matching problem that is addressed in this paper, and highlight some of the linguistic issues that arise in the matching process.

Primarily, we consider the problem of matching text attributes across multiple languages arising from different scripts. For example, the European Union may require to support such queries over a federated database that has relational columns in both Latin and Greek scripts. We restrict our matching to attributes that contain *proper names* (such as, *Authors, Corporations, Telephone Subscriber Names, etc.*) which are assumed not to have any semantic value to the user, other than their vocalization. We also assume that the attribute is tagged with its language, the reasons for which are given below. Though we restrict multi-script matching problem (in this paper) to only text attributes storing proper nouns, they nevertheless represent a significant fraction of user queries, as *proper and generic nouns represent a significant part of the user query strings and form about a fifth of normal corpora in text databases* [17].

The multilingual matching we have outlined here is applicable to many user domains, including telephone white pages enquiry, bibliography searches, and web search engines. We also expect such technology to be useful for integrating data in multi-national corporations (such as *Amazon.com, Siemens etc.*), news organizations (such as *Reuters, BBC etc.*), and inter-governmental organizations (such as *UN, EU, Interpol etc.*). Finally, it is also likely to be useful for data integration in large multilingual data warehouses.

2.1 Multi-script Selection

We now motivate the need for multi-script *selection* and *join* in database systems. Consider a query to retrieve all the works of an author from *Books.com*, irrespective of the language of publication. Figure 2 shows such a query for author `Nehru`, along with a sample output from the database system.

```
Select Author, Title, Price
From   Books
Where  Author = `नेहरु`
   or  Author = `Nehru` ...
```

Author	Title	Price
Nehru	Discovery of India	$ 9.95
नेहरु	भारत एक खोज	INR 175

Figure 2. Example for Multilexical Selection

The above query suffers from several problems: First, the user needs to know and specify the search string `Nehru` in all the languages in which `Nehru`'s works might have been published. Second, even if the user has this complete set of languages, the user needs to have access to lexical resources, such as fonts and multilingual editors, in each of such languages to input the query and specify all possible variations of the search string. In addition, the representation of proper names is significantly error-prone in databases, due to lack of dictionary support during data-entry[1].

To address the above problem, we propose the following query specification instead:

```
select * from Books
where Author LexEQUAL 'Nehru'
      inlanguages { English, Hindi, Arabic }
```

[1]The error rate for attributes storing names is empirically shown to be ≈ 3% (≈1.5% mis-spelling and ≈1.5-2% mis-typing) [12].

where LexEQUAL is a new *multi-script matching* operator that retrieve the strings that match phonetically to the query string in all of the user specified languages.

2.2 Multi-script Join

The new multi-script matching operator may also be used to *join* multilingual attributes, based on their phonetic representations. For example, the addresses of authors of books that were sold may be obtained by joining the *Author* attributes from the `Books` and `Authors` tables, as shown in the following SQL expression and in Figure 3.

```
select Authors.Author, Authors.Address, Books.Title
from Authors, Books
where Authors.Author LexEQUAL Books.Author
```

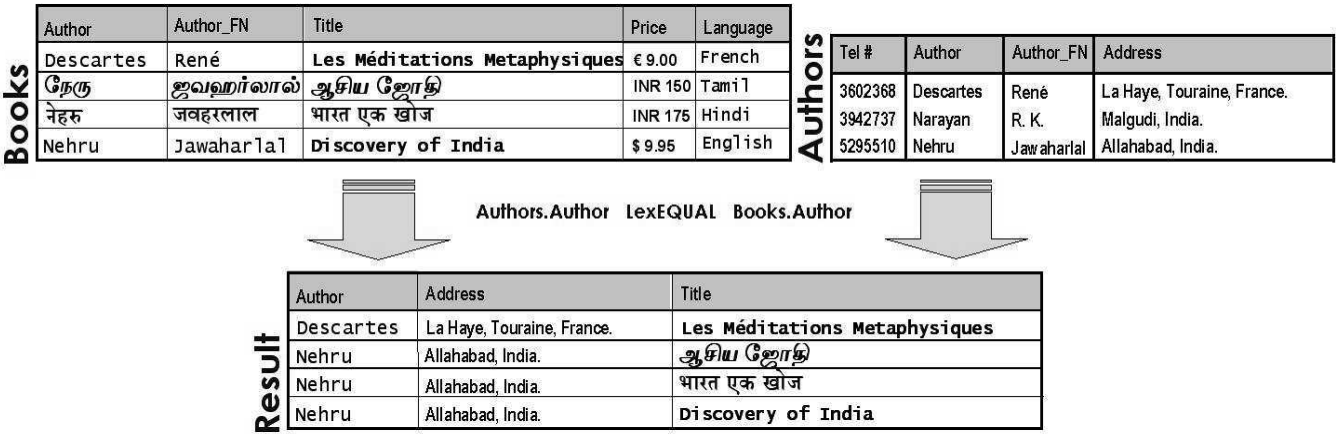

Figure 3. Multilexical (*Phonetic*) Join

A real-life *e-Governance* application that requires a join based on the phonetic equivalence of multi-script data is outlined in [14].

2.3 Linguistic Issues

The multi-script matching of proper names is fraught with variety of linguistic pitfalls: while issues such as, equating strings that vary only in combining diacritical marks[2] and user-accent differences[3] may be handled easily, we outline two important issues that are open and being addressed in our research.

2.3.1 Language-dependent Vocalizations

Given a text string, such as `Jesus`, the phonemic equivalent is different in English and in Spanish (where it is pronounced like "Hesus"). So, it is not clear when a match is being looked for, which vocalization(s) should be used. One solution is to take the vocalization that is appropriate to the language in which the

[2]By choosing the appropriate comparison levels pre-defined in Unicode, databases may match strings that vary only in diacritical marks (such as `Muller` and `Müller`). See Appendix A.5 and [5] for details.

[3]Accents do not play an important role for our multi-script matching, since we start with the written form of the names, stored in the database attributes. Such strings may be transformed using the same text-to-phoneme function.

source data is present (i.e. either English or Spanish) and hence we require language of an attribute to be identified.

Automatic language identification is not a straightforward issue – while many languages, including most of the Indic languages, are uniquely identified by their associated Unicode character-blocks, this is *not true* for the Western European languages which all share a common Unicode character-block. It is also not true for the Chinese-Japanese-Korean set of languages, which share character-blocks. There is quite some literature in the IR community on automatic language identification but the proposed techniques assume a sufficiently large corpus of text is available to make this determination – in the database world, where we deal with limited information at the attribute level, it becomes much harder to make such assignments.

2.3.2 Context-dependent Vocalizations

In some languages, the vocalization of a set of characters is dependent on the surrounding context. For example, consider the English string `live`. It may have different vocalizations depending on context (such as those in, "long *live* the king" and "*live* telecast"). While it is easy in running text to make the appropriate association, it again becomes difficult in the database context, where information is processed at the attribute value level.

3 LexEQUAL: Multilingual Matching Operator

In this section, we first show the strategy that we propose for matching multilingual strings and then detail our algorithm along with the description of algorithmic parameters for such matching.

3.1 Multilexical Matching Strategy

Our view of storage and semantics of textual information in databases, is shown in Figure 4. 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 (traditional mapping of a name to a character string, as shown by the dashed line) may also be stored as a phoneme string (proposed mapping of a name to phoneme string, as shown by the dotted line). Further, the transformation may be done as and when needed from the stored multilingual lexicographic string, using standard linguistic resources, such as *text-to-phoneme* converters.

In a multilingual environment, when a name is queried for, the primary intention of the user is in retrieving all names that match *aurally*, irrespective of the language. We propose a framework to capture this intention, in matching multilingual attributes, by matching their equivalent *phonetic* strings. Further, such phoneme strings represent a normalized form of proper names across languages, thus provide a means of comparison. Further, when the text data is stored in multiple scripts, this may be the *only* means of comparing them. In phoneme domain, the similarity may have to be tested using approximate matching techniques, due to the inherent fuzzy nature of the representation and due to the fact that phoneme sets of two different languages are seldom identical.

3.2 LexEQUAL Matching Algorithm

We propose complementing and enhancing the standard lexicographic equality operator of database systems with an matching operator that may be used for approximate matching of user-specified multi-

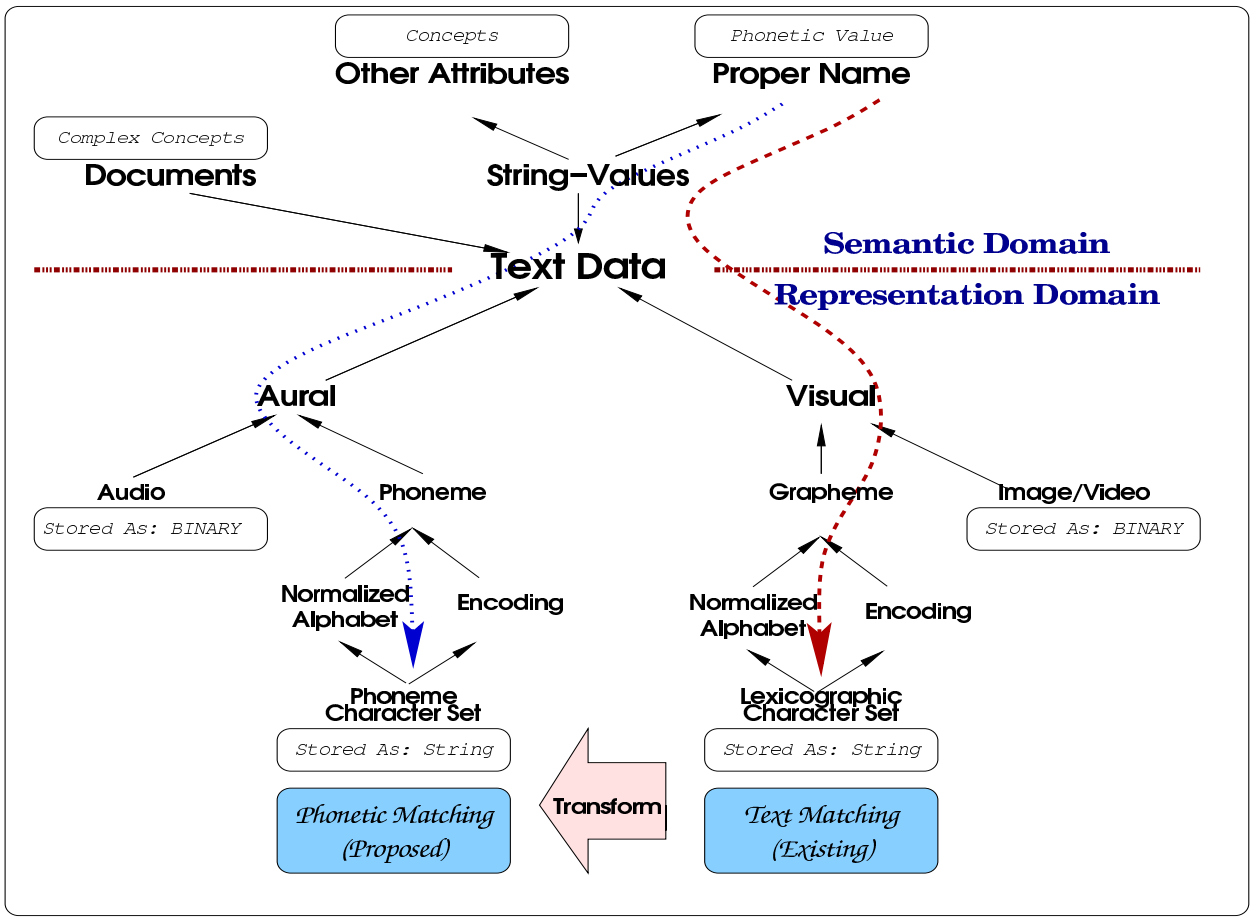

Figure 4. Ontology for Text Data

lingual names attributes. Such attributes are compared, after transforming them to equivalent phonetic strings in a common phonetic alphabet. Our algorithm for the LexEQUAL operator is shown in Figure 5.

The operator – LexEQUAL, accepts two multilingual text strings and an *user match threshold* parameter as input. The strings are transformed to their equivalent phonetic strings and the edit distance between them is computed. If the edit distance is less than the user-specified user match threshold, then a match is flagged. If the multilingual strings are in the same language, then comparison is specified as *lexicographic*, using standard database equality operator.

The `transform` function takes a multilingual string in a given language and returns its phonetic representation, in IPA alphabet (Appendix A.3 and [11]), which encodes the superset of all phonemes of all the natural languages. Such transformation may be easily implemented by integrating standard TTS systems that are capable of producing phonetically equivalent strings. The `editdistance` function [8] takes two strings and returns the *edit distance* between them. A *dynamic programming* algorithm is implemented, due to the flexibility that it offers in experimenting with different cost functions. Further, the LexEQUAL may be parameterized for fine tuning the quality of match for a given dataset.

3.2.1 *User Match Threshold* Parameter

A user-tunable parameter *User Match Threshold*, as a fraction between `0` and `1`, is accepted as an input for the phonetic matching. This parameter specifies the user tolerance for approximate matching: `0` signifies that only perfect matches are accepted, whereas a positive (but, ≤ 1) threshold specifies the allowable error (that is, edit distance) as the fraction of the size of query string. The correct value for the threshold parameter may be determined by the requirements of the application domain. For example, a *Directory Enquiry* application may accept more errors in matching than a *Credit Rating Verification* application.

3.2.2 Clustered Edit-Distance Parameterization

The three functions in Figure 5, 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. The standard *Levenshtein Edit Distance* [8] metric is simulated with all cost functions returning uniform value of `1`.

In our strategy, we allow a *Clustered Edit Distance* parameterization, by extending the *Soundex* [13] algorithm to the phonemic domain. In phonetic matching, a substitution between `a` and `ā` may be more acceptable than substitution between, say, `a` and `k`. Hence we created clusters of near-equal phonemes, based on the similarity measure as outlined in [19]. The substitution cost within a cluster is specified as a user-definable parameter – called *Intra-Cluster Substitution Cost*, and may be varied between `0` and `1`. Specifically, an intracluster substitution cost of `1` simulates the standard *Levenshtein* cost function and a cost of `<1` allows clustered phonemes to be exchanged with a smaller penalty than *Levenshtein* cost function.

Finally, we also allow user-defined clustering of phonemes, based on the languages of interest to the users and stringency of the application domain.

3.3 Our Current Implementation

We have implemented a basic architecture for querying multilingual data, as highlighted in Figure 6.

LexEQUAL (S_l, S_r, e)

Input: Strings S_l, S_r
User Match Threshold, e
Languages with IPA transformations, $\mathcal{S}_\mathcal{L}$ (*as global resource*)

Output: `TRUE`, `FALSE` or `NORESOURCE`

```
   // get the languages of the input Strings
1.  Ll ← Language of Sl;Lr ← Language of Sr;
   // if same language strings, use lexicographic comparison
2.  if Ll = Lr then return (Sl=Sr ? TRUE : FALSE);
   // else, transform and use approximate matching techniques
3.  if Ll ∈ SL and Lr ∈ SL then
4.    Tl←transform(Sl,Ll); Tr←transform(Sr,Lr);
5.    Smaller ← (| Tl | ≤ | Tr | ? |Tl| : |Tr|);
      // match if edit distance ≤ (threshold * length(SmallerString))
6.    if editdistance(Tl,Tr) ≤ (e * Smaller) then
        return TRUE else return FALSE;
7.  else return NORESOURCE;
```

editdistance(S_L, S_R)

Input: String S_L, String S_R

Output: Distance k

1. $L_l \leftarrow |\ S_L\ |$; $L_r \leftarrow |\ S_R\ |$;
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. $DistMatrix[i, j] \leftarrow$ **Min**
$$\left\{ \begin{array}{c} DistMatrix[i-1, j] + InsCost(S_{L_i}) \\ DistMatrix[i-1, j-1] + SubCost(S_{R_j}, S_{L_i}) \\ DistMatrix[i, j-1] + DelCost(S_{R_j}) \end{array} \right\}$$
8. **return** $DistMatrix[L_l, L_r]$;

Figure 5. The LexEQUAL Algorithm

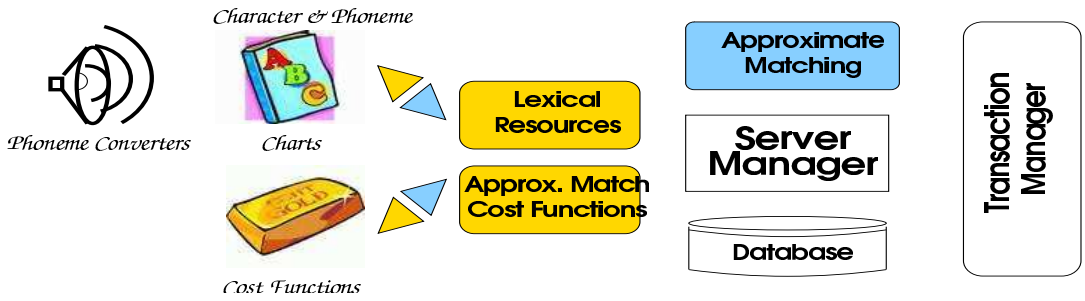

Figure 6. Architecture

Lexical resources (such as, Unicode code tables for required languages and IPA) were installed. We integrated components of a basic TTS engine that converts a given language string to its equivalent phonemes in *IPA* alphabet. Ideally, the implementation should install or invoke needed transformation modules dynamically, to optimize the memory usage of the server. Further, since the quality of the match depends on quality of transformation, the system should allow easy modification to or exchanges of transformation functions, in a modular fashion. The multi-script comparison operator – LexEQUAL was implemented as an *user-defined function (UDF)* that can be called in SQL statements. Our approach required minimal changes to existing database architecture or usage semantics. Further, it should be noted that in our implementation scheme, only the comparison semantics would be altered, but the output would be the multilingual records, in the stored language characters sets only.

4 Experimental Study

In this section, we outline an experimental setup to measure the effectiveness of phonemes matching of names. In this section, we outline our methodology for verifying the correctness of the matching and provide a methodology for fine-tuning the quality. in Section 5 we provide the performance of such queries.

4.1 Experimental Data

For our study of correctness of LexEQUAL operator, we needed a large multi-script lexicon with *phonetically equivalent* names marked explicitly. Since none were available, we had the following two choices: experiment with a given multilingual lexicon and verify correctness by *manual relevance judgements*, or alternatively, create a manually tagged multilingual lexicon and verify correctness mechanically. Bilingual dictionaries mark semantically equivalent, and not necessarily phonetically equivalent words. Though directories in local scripts are available, since equivalent strings in each were not tagged explicitly, their use in approximate matching produced huge outputs, posing a problem for manual relevance judgements. Hence we took the second approach of creating a multilingual names lexicon containing phonetically equivalent names, hand-tagging of equivalent names in different languages, and automating the verification of correctness of the matches, using these tags. Further, we generated a large (about 200,000 multi-script names) tagged lexicon, by using the originally hand-tagged multi-script lexicon, for our performance experiments.

4.1.1 Creation of Tagged Multi-script Lexicon

To create the multi-script lexicon, we followed the following approach: First, sets of names were selected from three different sources so as to cover names in English and Indic domains, fairly evenly. 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 *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 *Oxford English Dictionary*. Together the set yielded about 800 names in three different languages, namely English and two Indic languages – Tamil and Hindi. All phonetically equivalent names (the same name from different sets, and presumably in different scripts) were tagged with a common tag-number. Because of such tagging, we expect to find the precision and recall figures for every experiment accurately; any match of two multilingual strings is considered to be correct if their tag-numbers are the same, and considered to be *false-positive* if their tag-numbers are different. *False-dismissals* can be computed since for every input, we know the expected set of correct matches, based on the tag-numbers. Our aim is to get the matching to be almost perfect in our subsequent experiments, as the names were deliberately picked to be diverse.

Lexicographic String	Language	Phonetic Representation (*in IPA*)
Amazon	English	æməzɑn
நாராயண்	Tamil	nɑrɑyən
University	English	junəvɜrsɪti
இந்தியா	Tamil	ɪndɪyɑ
हैडरीजेन	Hindi	hɑɪdrədʒən
Computer	English	kəmpjutər

Figure 7. Phonemic Representation of Test Data

To convert English names into corresponding phonetic representations, we used the following two standard resources: First, the *Oxford English Dictionary* [22] that provides phonetic representations, was used whenever a transformation is available, and the text to phoneme converter published by *www.ForeignWord.com* [7], for the others. The transformed string is further cleaned up, by removing symbols specific to speech generation, such as the suprasegmentals, diacritics, tones and accents. For Indic scripts, we developed a phonemic converter along the lines of the transliterations outlined in [27]. Sample phonetic strings are given in Figure 7.

The histogram of the data set used for measuring quality of matches – both lexicographic and the generated phonetic representations are shown in Figure 8. The set had about 800 names with an average lexicographic length of 7.35 and average phonetic length of 7.16 with each multilingual string tagged with a tag-number to indicate those other multilingual strings that it is expected to match with[4].

Further, for performance experiments, we generated a large data set, using the tagged multilingual lexicon as shown above. 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 phonetic length of 14.31. The histogram of the generated data set – both in character and generated phonemic representations, is given in Figure 9. Further, every multilingual string thus generated was tagged with a tag-number to mark the other multilingual strings that it is expected to match with.

[4]It is interesting to note that though visually the Indic strings are much shorter compared to English strings, their text representations and phonemic representations are much alike, owing to the fact that most Indic characters are composite glyphs.

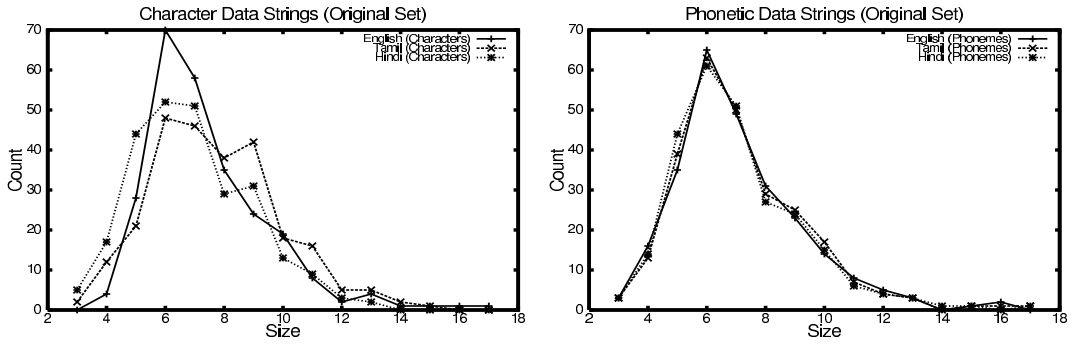

Figure 8. Profiles of Original Data Set

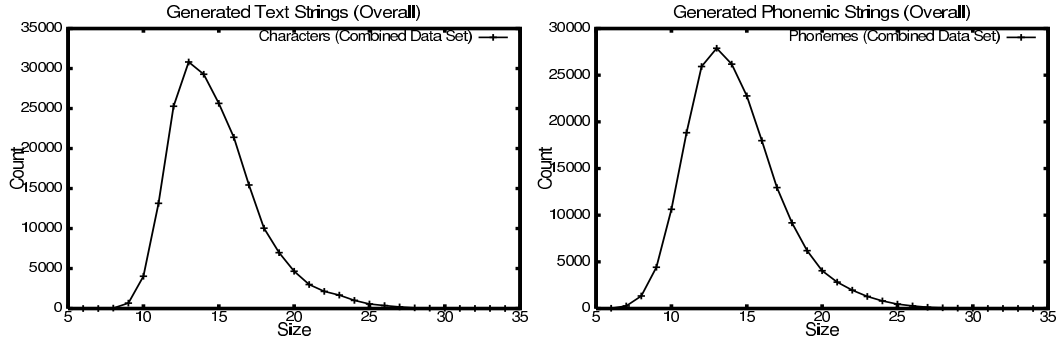

Figure 9. Profiles of Generated Data Set

4.2 Database Setup

We implemented a prototype phonetic matching system using the *Oracle 9i (Version 9.1.0)* database system. The multilingual strings were stored in Unicode, along with their phonetic representations, also in the Unicode format, in the IPA alphabet. The algorithms shown in Figure 5 were implemented, as UDF in the PL/SQL language.

4.3 Query Parameters

The following two query parameters were set for each query, to measure their effect on the quality of the matches: The first, *User Match Threshold* parameter that sets the tolerance level for matches, was varied between `0` and `1`. The second, *Intracluster Substitution Cost* (for the Clustered Edit Distance parameterization), which varies the cost of substitution of a character from within a user-specified clusters of phonemes between `0` and `1`. These parameters were explained in Section 3.2.

4.4 Metrics Measured

Along the lines of *Information Retrieval* system, we measured two specific metrics for quantifying the quality of approximate matching. They are, *Recall*, defined as *the fraction of correct matches in the result* and *Precision*, defined as *the fraction of the results that are correct.*

For every query we measured the recall and the precision figures by the following methodology: We matched each phonetic string in the data set with every other phonetic string, counting the number of matches (m_1) that were correctly reported (that is, those matches that were between strings with

the same tag-number, which are expected to match in the first place), along with the total number of matches that are reported as the result (m_2). Please note that the total number of matches reported (m_2) includes *false-positives*, that is, those phonetic strings that matched across groups. If there are n equivalent groups (with the same tag-number) of multi-script strings with n_i strings each[5], the *precision* and *recall* metrics are calculated as follows:

$Recall = m_1 / \sum_{i=1}^{n} ({}^{n_i}C_2)$, and

$Precision = m_1 / m_2$

The denominator in *recall* 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 phonetic strings.

4.5 Correctness of Phonetic Matching

We ran our experiments by matching each of the multilingual string on the tagged lexicon, to measure the quality of the phonetic matching. Our objective is to tune the parameters to achieve the best possible recall and precision, for the given data set.

4.5.1 Performance of Approximate Matching for the Given Lexicon

The plots of the *recall* metric against *user match threshold*, for various *intracluster substitution costs*, between `0` and `1`, are provided in Figure 10.

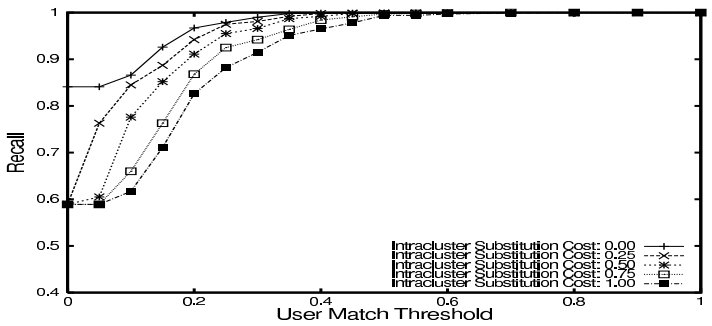

Figure 10. Recall

Overall, the recall improves with increasing user match threshold, and reaches perfect recall, after an threshold of `0.5`, asymptotically. An interesting point to note is that the recall gets better with reducing intracluster substitution costs, validating the assumption of *Soundex* algorithm [13]. Also, our results indicate that it is necessary to set the threshold to above a figure of `0.3`, to be assured of a recall of over 90%.

The plots of *precision* metric against *user match threshold*, for various *intracluster substitution cost*, varied between `0` and `1`, are provided in Figure 11.

The graphs indicate that while the precision drops with increasing threshold, the drop is negligible for threshold $\leq$`0.2`, but rapid in the range between `0.2` and `0.5`. For a precision of $\geq$80%, the threshold may need to be $\leq$`0.3`. However, it is interesting to note that with a intracluster substitution cost of `0`, the precision drops too rapidly, too early (around `0.1` itself). That is, the *Soundex* method, which

[5]Both n and n_i are known during the tagging process.

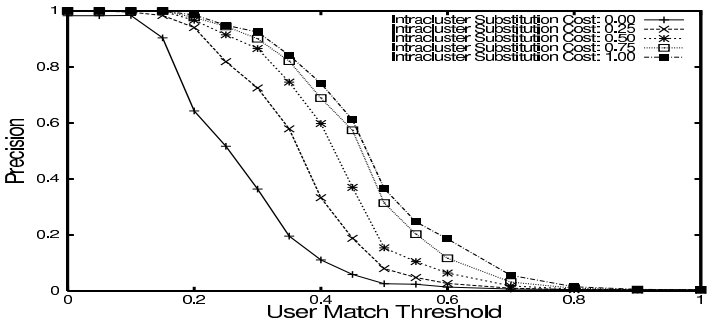

Figure 11. Precision

is good in recall, is very ineffective in precision, as it introduces too many false-positives even at low thresholds.

4.5.2 Selection of Ideal Parameters for Phonetic Matching

Figure 12 illustrates the *recall-precision* 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 threshold between `0.2` and `0.4`. While the top-right corner, corresponding to perfect precision and recall, the curves indicate that 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 of the respective curves.

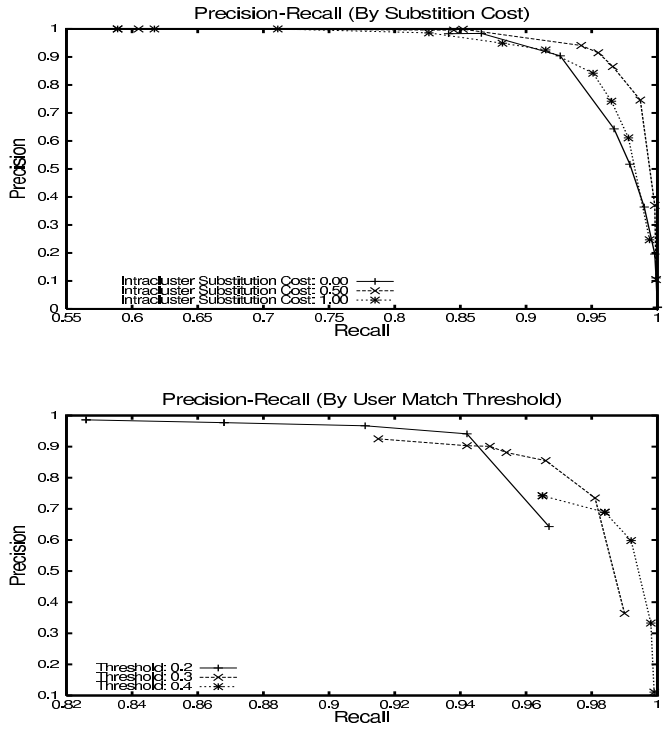

Figure 12. Precision-Recall Graphs

The analysis suggests as the best set of parameters for phonetic matching of the given data set as, *thresholds* in the range [0.25, 0.3], and *intracluster substitution cost* in the range [0.25, 0.5]. With such parameters, the *recall* is ≈95%, and a *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-phonemic methods[6].

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 following a procedure as outlined in this section, by the developers or administrators of specific applications as required by their application domains. In our future work, we plan to investigate techniques for automatically generating the appropriate parameter settings based on dataset characteristics.

5 Improving Approximate Matching Query Performance

In this section we discuss the query performance of queries using the LexEQUAL operator. All experiments shown in this section are run on the large generated data set (of about 200,000 multi-script names) to provide a reasonable run times for comparison.

5.1 Baseline LexEQUAL Runs

While the dynamic programming algorithm used for LexEQUAL is flexible for experimentation, it has large time complexity and the UDF implementation incurs high call overheads. In addition, since the UDF cannot be costed properly, the optimizer may resort to inefficient plans.

To create a baseline for performance, we first ran the simple `select` and `join` queries using LexEQUAL operator on the large generated data set. The Table 1 shows the performance of the native equality operator (for exact matching of character strings) and the LexEQUAL operator (for approximate matching of phonetic strings), for basic **scan** and **join** queries[7]. The performance of the standard database equality operator was shown only to highlight the inefficiency of approximate matching operator. As can be seen clearly, the UDF is orders of magnitude slower compared with native database equality operators. In the join query, as expected, the optimizer chose *nested-loop* join irrespective of the size of join, availability of indexes or optimizer hints, indicating that little optimization was done on the query.

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. Performance of Approximate Matching

Next, to improve the efficiency of the approximate matching LexEQUAL operator, we outline two techniques – *Q-Grams* and *Approximate Phonetic Index*, that provide a candidate set of answers cheaply, which is further operated on by accurate but inefficient LexEQUAL UDF, to weed out *false-positives*.

[6]In our experiments, we were handicapped by our choice of one of the Indic languages, namely Tamil, that has the minimal phoneme set, contributing to loses in transformation.

[7]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.

5.2 Q-Gram Technique

In this section, we show that the *Q-Grams* used in approximate matching of standard text strings [9] may be extended to phonetic matching as well, and we sketch briefly the filters and their implementation in SQL.

We first augment the database with a table of q-grams of the original phonetic strings. Once created, the following three filters, namely *Length*, *Count* and *Position* filters that use distinct properties of q-grams were used to filter out the unmatchable strings using traditional database operators. Thus the filters weed out all mismatches cheaply, leaving the expensive approximate matching LexEQUAL operator to be called only on a vastly reduced candidate set, to weed out *false-positives*, accurately.

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

```
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  /* Length Filter   */  |len(N.PName) − len(Q.str)| ≤ e ∗ length(Q.str)

  AND  /* Position Filter */  |AN.Pos − AQ.Pos| ≤ (e ∗ length(Q.str))

GROUP BY N.ID, N.PName
HAVING /* Count Filter    */  count(*) ≥ (len(N.PName) − 1 − ((e ∗ len(Q.str) − 1) ∗ q))

   AND  LexEQUAL(N.PName, Q.str, e)
```

Figure 13. SQL using *Q-Gram* Filters

A sample scan query is shown in Figure 13, assuming that the query string is transformed into a record in table Q, and the auxiliary q-gram table of Q is created in AQ. The *Length Filter* is implemented in the fourth condition of the SQL statement, *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 the three filters have filtered out the unmatchable strings. Thus, the expensive *UDF* is called only for those pairs of strings, not weeded out by the filters.

5.2.1 Performance of Q-Gram Filters

We re-ran the same **scan** and **join** queries on the large generated multi-script data set, employing the q-gram filters in a SQL expression as shown in Figure 13. The performance of the queries are given in the Table 2.

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

Table 2. Performance with *Q-Gram* Filters

Evidently, the use of q-gram filters have improved the *scan* query performance by an order of magnitude and the *join* query performance by five-fold, over plain UDF-only queries. The improvement in performance in join 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, we would like to note that the performance improvements were not as high as reported in [9], perhaps due to our use of standard commercial database system and the implementation of LexEQUAL using slow dynamic programming algorithm, in an interpreted language environment, namely PL/SQL.

5.3 Phonetic Indexing Technique

In this section we propose a *phonetic indexing* technique that may be used for accessing the *near-equal* phonetic strings, using a standard database index.

The recall and precision graphs in Figures 10 and 11 indicate that for small values of intracluster substitution costs, the *recall* remains high, but the precision drops too rapidly due to the introduction of large number of *false-positives*. However, we show in this section that such grouping may be used for building an approximate index, that may retrieve cheaply a candidate set (with *false-positives)* of answers, which may then be processed using slower, but accurate LexEQUAL UDF function.

5.3.1 Phonetic Index, using Database Integer Indexes

In this technique, the basic strategy is to transform the phoneme strings to a number, such that phoneme strings that are *close* to each other map to the same number. Such transformation modifies the *Soundex* algorithm [13] to the phoneme space. The transformed numbers are indexed (as integers) using standard database indexes. By searching such an index, the set of candidate matches (with *false-positives*) may be obtained efficiently, to be further processed using the expensive, but accurate LexEQUAL UDF.

We first grouped the phonemes into equivalent clusters along the lines of clustering outlined in [19], and assigned a unique number to each of the clusters. Each phoneme string was transformed to a numeric string, by concatenating the cluster identifiers of each phoneme in the string. Please note that such a transformation is unique for a given string. The numeric string thus obtained is converted into an integer – *Grouped Phoneme String Identifier* and stored along with the phoneme string. A standard database B-Tree index is built on phoneme group identifier attribute, thus creating a compact index structure using only integer datatype.

```
SELECT N.ID, N.Name
FROM  Names N, Query Q
WHERE /* index scan */  N.GrPhStringId = Q.GrPhStringId
  AND LexEQUAL(N.PName, Q.PName, e)
```

Figure 14. SQL using Phonetic Indexes

For an approximate scan or join query, we transform the query string to its phonetic representation, and subsequently to its grouped phonetic string identifier. The index on the grouped phoneme string

identifier of the lexicon is used first to retrieve all the candidate phoneme strings, which are further tested for a match invoking the LexEQUAL UDF with the user specified tolerance for match. The invocation of LexEQUAL operator in a query maps into an internal query as shown in Figure 14 that uses the phonetic index, for a sample join query.

Please 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.

5.3.2 Approximate Phonetic Index Performance

The *phonetic group identifier* attribute is created on the phoneme table, and is indexed as described earlier. We ran the same table scan and join queries on the large generated multilingual data set, after creating the index on the grouped phoneme string identifier attribute that is obtained by transforming each of the phonetic strings of stored names. The LexEQUAL operator is modified to use this index, using the SQL expression given in Figure 14. The performance of scan and join using the *phonetic index* 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. Performance using *Indexes*

Clearly, the performance of queries using *approximate phonetic index* is improved by orders of magnitude, even beyond the performance improvements shown by q-gram techniques. However, we also observed a small, but significant 4 - 5% *false-negatives*, with respect to the classical edit-distance metric. With more robust grouping of phonemes, a closer match may be achieved.

6 Conclusions and Future Research

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

We proposed a strategy to solve the multi-script matching problem, specifically for proper name attributes, by transforming matching in the *lexicographic space* to equivalent *phonemic space*. Such transformation may be done using standard linguistic resources such as dictionaries and text-to-speech converters. Due to the inherent fuzzy nature of the phonemic space, we employ approximate matching techniques for matching the transformed phonemic strings. Since none of the existing databases natively support approximate matching, we implemented the multi-script matching operator as a *user-defined function* in a commercial database. Overall, we outlined our implementation that changes minimally the basic database architecture.

We confirmed the correctness of our strategy by measuring the quality metrics, namely *Recall* and *Precision*, in matching a real, tagged multilingual data set, to obtain the best trade off between recall and precision. The results from our initial experiments on a given data set showed good ≈95% recall and ≈85%precision, 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 characteristics.

Further, we showed that the poor performance associated with the UDF implementation of approximate matching may be improved significantly, by employing the same techniques used in approximate matching, namely *Q-Gram* techniques. In addition, we proposed a solution that captures the phonetic closeness – *phonetic index* that may be used for building an index to aid performance improvements. We demonstrated that such techniques improved the matching query performance by orders of magnitude.

Thus, we show that the LexEQUAL operator outlined in this paper to be effective in multi-script matching, and can be made efficient as well using *q-gram* and *phonetic indexing* techniques. *Multi-script Matching* may prove to be a valuable tool and hence we recommend its implementation as a SQL function to leverage the full capabilities of the database engine.

6.1 Future Research

Multilingual (*Semantic*) Selection: Joining on multilingual attributes need not be restricted to lexical domains only, but may be extended to semantics as well. In *Books.com* a query to retrieve all books related to *History* may look like the one in Figure 15. The values for the *Category* attribute in the resulting set of records are neither *equal*, nor *equivalent lexically*, but they are all equivalent *semantically* to `History`.

```
Select Author, Title, Category, Price
From   Books
Where  Category SemEqual `History`
```

Author	Title	Category	Price
Durant	History of Civilization	History	$ 149.95
नेहरु	भारत एक खोज	इतिहास	Rs 175
நேரு	ஆசிய ஜோதி	சரித்திரம்	Rs 150
Toynbee	A Study of History	History	$ 29.95

Figure 15. Multilexical (*Semantic*) Selection

We are currently working on implementation of an operator, `SemEQUAL` that allows such matching of attributes based on their semantic values, using standard linguistic resources, such as WordNet [6] that define semantic units of a language and the rich inter-relationships between them.

Approximate Indexes: We are exploring the use of approximate indexes as outlined in [2], for further performance improvements. Further, specific issues in phonetic indexes must be addressed, such as the collation order of IPA phonemic set not corresponding to any of the languages. Hence, processing range queries (such as, names between *abacus* and *kite*) may not make sense, phonetically.

Multilingual Benchmark Suites: Database system are the backbone for most *e-Commerce* and *e-Governance* applications, handling large volumes of multilingual text. However, no well accepted and trusted benchmarks, similar to the standard TPC benchmarks [28], exist for comparing different database systems with respect to multilingual functionalities and performance. We propose to develop such suites, using techniques outlined in this paper and our paper on multilingual performance [15].

References

[1] A. Amir *et al.* Advances in Phonetic Word Spotting. *Proc. of CIKM, Atlanta, Georgia*, 2001.

[2] R. Baeza-Yates *et al.* Proximity Matching Using Fixed-Queries Trees. *Proc. of 5th Symp. on Combinatorial Pattern Matching, Asilomar, California*, 1994.

[3] R. Baeza-Yates and G. Navarro. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 2000.

[4] W. A. Burkhard and R. M. Keller. Some Approaches to Best-Match File Searching. *Comm. of the ACM*, April, 1973.

[5] M. Davis. Unicode collation algorithm. *Unicode Consortium Technical Report*, 2001.

[6] C. Fellbaum and G. A. Miller. WordNet: An Electronic Lexical Database (Language, Speech and Communication). *MIT Press, Cambridge, Massachusetts*, 1998.

[7] The Foreign Word – The Language Site, Alicante, Spain. *http://www.ForeignWord.com.*

[8] D. Gusfield. Algorithms on Strings, Trees and Sequences. *Cambridge University Press, Cambridge, UK*, 2001.

[9] L. Gravano *et al.* Approximate String Joins in a Database (almost) for Free. *Proc. of the 27th VLDB Conference, Rome, Italy*, 2001.

[10] International Organization for Standardization. ISO/IEC 9075-1-5:1999, Information Technology – Database Languages – SQL (parts 1 through 5). 1999.

[11] The International Phonetic Association. University of Glasgow, Glasgow, UK. *http://www.arts.gla.ac.uk/IPA/ipa.html.*

[12] D. Jurafskey *et al.* Speech and Language Processing. *Pearson Education, New Delhi, India*, 2000.

[13] D. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. *Addison-Wesley, Reading, Massachusetts*, 1993.

[14] A. Kumaran and J. R. Haritsa. On Database Support for Multilingual Environments. *Proc. of the IEEE RIDE Workshop on Multilingual Information Management, Hyderabad, India*, 2003.

[15] A. Kumaran and J. R. Haritsa. On the Costs of Multilingualism in Database Systems. *Proc. of the 29th VLDB Conference, Berlin, Germany*, 2003.

[16] B. L. Lambert *et al.*. Descriptive analysis of the drug name lexicon. *Drug Information Journal*, 2001.

[17] M. Liberman and K. Church. Text Analysis and Word Pronunciation in TTS Synthesis. *Advances in Speech Processing*, 1992.

[18] J. Melton *et al.* SQL 1999: Understanding Relational Language Components. *Morgan Kaufmann, San Francisco, California*, 2001.

[19] P. Mareuil *et al.* Multilingual Automatic Phoneme Clustering. *Proc. of International Congress of Phonetic Sciences (ICPhS 99), San Francisco, California*, 1999.

[20] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 2001.

[21] G. Navarro *et al.* Indexing Text with Approximate q-grams *University of Chile - Technical Report*, 1994.

[22] The Oxford English Dictionary. *Oxford University Press, Oxford, UK*, 1999.

[23] U. Pfeifer *et al.* Searching Proper Names in Databases. *University of Dortmund - Technical Report*, 1994.

[24] L. Rabiner *et al. Fundamentals of Speech Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[25] D. Shasha *et al.* New Techniques for Best-Match Retrieval. *ACM Trans. on Information Systems*, April, 1990.

[26] S. Sarawagi *et al.* Interactive De-duplication using Active Learning. *Proc. of ACM SIGKDD Conference, Edmonton, Canada*, 2002.

[27] Technology Development for Indian Languages. Government of India. *http://tdil.mit.gov.in.*

[28] The Transaction Processing Council. San Francisco, California. *http://www.tpc.org.*

[29] The Unicode Consortium. The Unicode Standard. *Addison-Wesley, Reading, Massachusetts*, 2000.

[30] The Unisyn Project. The Center for Speech Technology Research, University of Edinburgh, UK. *http://www.cstr.ed.ac.uk/projects/unisyn/.*

[31] J. Zobel *et al.* Finding Approximate Matches in Large Lexicons. *Software – Practice and Experience*, 1995.

[32] J. Zobel *et al.* Phonetic String Matching: Lessons from Information Retrieval. *Proc. of 19th ACM SIGIR Conference, Zurich, Switzerland*, 1996.

Appendix A: Background Information

This appendix provides the needed background on character encoding, phonemes and encoding and on definition and techniques used in approximate matching.

A.1 Lexicography and Characters

A *Character* is the smallest component of written language that has a representational value. A *Script* is a set of characters, such as *Latin*, usually pertaining to a group of languages. A language *Repertoire* is a set of characters, possibly from multiple scripts, that form the individual alphabet of that language. In addition, a language defines the rules for composition of its alphabet and their ordering uniquely, called *Lexicography*, that define the word ordering in a language.

In database systems, text data strings are stored in some unique *Character Encoding* that assigns a unique value to each of the characters in a language. There are several well-known encodings, such as ISO-8859 and Unicode [29], that are used to represent a character set. It should be noted here that while *scripts* correspond to character blocks in any of the encoding schemes, *language* is defined on a set of characters, possibly from several scripts, with specific rules on composition, collations, vocalization etc. All database systems use characters as the basic unit for storage and the language collations as the basis for sorting and index building, if defined, or lexicographic ordering if collation is undefined, or sorting is done across languages.

A.2 Phonology and Phonemes

Phonology is the study of sound structure related to speech, conforming to the grammar of a *language*. Each human language usually has between *20* to *40* abstract linguistic units, called *Phonemes*, that provide an alphabet of sounds that describe the articulation of the words in that language uniquely. A *Phone* is the physical sound produced conforming to a phoneme. Since the phone is produced by the vocal tracks of individuals, there are infinite variations of phones (called *Allophones*) that are possible based on speaker's individual, cultural and environmental factors. However, they are identified with a specific phoneme using common aural signatures.

Phonemes are grouped together in *syllables*, which are in turn grouped together in *words* of a language. Phonemes are to speech, what characters are to written text. Not all possible orders of phonemes are allowed, much as not all possible sequences of characters are allowed. However, there is no a straightforward one-to-one mapping between characters of a language to phonemes, as the vocalization of the characters depend on context of the character within the word, or even words around it. Such rules of the mapping of a group of characters to a group of phonemes are extensively researched in Linguistics and Speech Processing communities. While such transformation rules are outside the scope of this paper, they are available in standard implementations of *Text-to-Speech (TTS)* systems of a language.

A.3 International Phonetic Alphabet

International Phonetic Association (*IPA*) [11] is one of the popular standards for describing phonemes of any given language[8]. The phonetic alphabet of IPA is capable of representing the full range of vocalizations primitives, irrespective of languages. Popular linguistic resources, such as *Oxford English Dictionary* [22], define and publish the phonemic equivalent of all words in IPA alphabets and standard TTS systems can generate a phonetically equivalent IPA string for a given character string of that language. More recently, efforts such as Unisyn [30] are underway to specify abstract phonemes that may provide an accent-free phonemic representation, which may be instantiated to a user's geographic, cultural requirements.

Further, the IPA alphabets are allocated a specific character block in the Unicode[29] encoding scheme.

[8]Standards such as *Arpabet* [12] also exist, though they originally designed for American English, using ASCII alphabets.

Hence, phonetic representation of character strings in *any* language may be stored and manipulated as Unicode strings in IPA character set. The support for storing Unicode data in database systems and the query performance profiles are detailed in [15].

A.4 Approximate String Matching

Approximate matching techniques are used for matching strings that are *close to each other* in a common alphabet, but not exactly equal. Common use for approximate matching techniques are in Bioinformatics, for genomic comparison and in Information Retrieval, for retrieval with expected typographic errors. Several frameworks exist to capture the notion of *closeness* of strings. The popular among them is the *Edit Distance* metric, which is used in the *Approximate String Matching*, as given in the following definitions [8].

Edit Distance The *edit distance* between two strings in a common alphabet Σ, is the minimum number of edit operations (i.e., insertions, deletions and substitutions) that are needed to transform one string to the other.

Approximate Matching Two strings are considered to *match approximately*, if the *edit distance* between them is less than a user specified threshold (possibly, as a function of strings themselves).

A.4.1 Q-Grams for Approximate Matching [8]

Let σ be a string of size n in a given alphabet Σ. $\sigma[i, j]$, $1 \leq i \leq j \leq n$, denotes a substring starting at position i and ending at position j of σ. A substring, $\sigma[i, i+q-1]$ of length q, is called a *Q-Gram* of σ. The *q-grams* of σ consists of all q-length substrings of σ, and is obtained by sliding a window of size q over the string. Further, the pair $(i, \sigma[i, i+q-1])$ is called the *positional q-gram*, where i is the starting position of the q-gram in σ. Usually, the q-gram matching techniques use augmented string σ_{aug}, where $(q-1)$ start symbols (say, $\triangleleft$) are pre-pended to σ and $(q-1)$ end symbols (say, $\triangleright$) are appended to σ, where $\triangleleft$ and $\triangleright$ are not part of the original alphabet, Σ. Note that for a given string σ, there are $(|\sigma|+q-1)$ q-grams.

A.5 Existing Support for Multilexical Matching

In this section, we outline briefly the support for multi-script matching, provided by the standards and the current database systems.

Unicode Support Unicode, the multilexical character encoding standard, specifies the semantics of comparison of two multilingual strings in three different levels [5]: using base characters, case or diacritical marks. Such schemes are applicable only between strings in those languages that share same script. Comparison of multilingual strings across scripts is *binary*.

SQL Standard Support The SQL:1999 standard [10] [18] allows the specification of *Collation Sequences* pertaining to a specific language, to correctly sort and index the text data. Comparison within a collation has normal semantics and comparison across collations is *binary*.

Database Systems Support The following four features of databases systems are considered:

Multilexical Comparison All systems have pre-defined collation sequences for every language supported. While the comparison within a collation has normal semantics, comparison across collations is binary.

Multilexical Indexing Since comparison across collations are binary, any indexes that are also built on strings from different collations are built with only binary sort order.

Approximate Matching Approximate matching is not supported by any of the databases. However, all database systems allow *User-defined Functions* that may be used to add functionality to the server.

Phonetic Matching Most database systems allow matching of *English* strings using pseudo-phonetic *Soundex()* originally defined in [13]. However, such algorithms do not scale well beyond English.

Regular Expression Matching Most database systems allow matching of regular expressions using `LIKE` operator, but such matching does not scale beyond a single script.

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