On Pushing Multilingual Query Operators into Relational Engines

A. Kumaran* Pavan K. Chowdary Jayant R. Haritsa
Database Systems Laboratory, SERC/CSA
Indian Institute of Science, Bangalore 560012, INDIA

Abstract

To effectively support today's global economy, database systems need to manage data in multiple languages simultaneously. While current database systems do support the storage and management of multilingual data, they are not capable of querying across different natural languages. To address this lacuna, we have recently proposed two cross-lingual functionalities, LexEQUAL[13] *and* SemEQUAL[14]*, for matching multilingual names and concepts, respectively.*

In this paper, we investigate the native implementation of these multilingual functionalities as first-class operators on relational engines. Specifically, we propose a new multilingual storage datatype, and an associated algebra of the multilingual operators on this datatype. These components have been successfully implemented in the PostgreSQL database system, including integration of the algebra with the query optimizer and inclusion of a metric index in the access layer. Our experiments demonstrate that the performance of the native implementation is up to two orders-of-magnitude faster than the corresponding outside-the-server implementation. Further, these multilingual additions do not adversely impact the existing functionality and performance. To the best of our knowledge, our prototype represents the first practical implementation of a cross-lingual database query engine.

1 Introduction

The Internet is steadily turning multilingual[1] and it is therefore imperative that the key applications of the Internet, such as *e-Commerce* and *e-Governance* portals, work seamlessly across multiple natural languages. A critical requirement to achieve this goal is that the underlying data sources – relational database management systems – should manage multilingual data effectively, efficiently and seamlessly. While current database systems certainly do support the storage and management of multilingual data, they are not capable of querying *across different natural languages*, that is, supporting "cross-lingual queries". To address this lacuna, we have recently proposed two new multilingual operators – LexEQUAL [13] and SemEQUAL [14] – which support (approximate) phoneme-based matching of names, and ontology-based matching of concepts, respectively.

The above works focussed primarily on defining the operator functionalities and their *outside-the-server* implementations. In this paper, we take the next logical step of investigating how to push them as first-class operators into relational database engines. Towards achieving this goal, we propose a new multilingual datatype, called UniText, and an operator algebra, called Mural, that defines uniform query semantics for the multilingual operators on the UniText datatype. Mural offers an intuitive framework for declaratively expressing complex queries with the multilingual operators. Further, it is relationally complete, implying that the multilingual datatype and operators may be *organically* added to the existing system, with little impact on existing functionality. We also specify the other components needed for leveraging the query optimizer, specifically the operator cost models and selectivity estimators.

All the above components have been successfully implemented in the PostgreSQL [26] open-source relational database engine. In addition, to speed up the query processing of the inherently fuzzy matches in multilingual predicates, a metric index, called M-Tree [3], has been incorporated using the GiST feature of PostgreSQL. Our experiments on representative multilingual datasets demonstrate that such an implementation can deliver up to *two* orders-of-magnitude improvement over a comparable outside-the-server approach. Further, these multilingual additions do not adversely impact the current functionality and performance.

To the best of our knowledge, the prototype implementation of multilingual features on PostgreSQL described here is the first practical attempt towards the ultimate goal of realizing *natural-language-neutral* database engines.

*Currently with Microsoft Research India (kumarana@microsoft.com)

[1] Two-thirds of current Internet users are non-native English speakers [21] and the majority of web pages will be multilingual by 2010 [27].

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)
8-7695-2570-9/06 $20.00 © 2006 IEEE

Author	Author_FN	Title	Price	Category	Language
Durant	Will/Ariel	History of Civilization	$ 149.95	History	English
Descartes	René	Les Méditations Metaphysiques	€ 49,00	Philosophie	French
नेहरू	जवाहरलाल	भारत एक खोज	INR 175	इतिहास	Hindi
Gilderhus	Mark	History and Historians	$ 19.95	Historiography	English
Nero	Bicci	The Coronation of the Virgin	€ 99,00	Art Fini	Italian
Nehru	Jawaharlal	Letters to My Daughter	£ 15.00	Autobiography	English
無門	慧開	無門関	¥ 7500	禅	Japanese
Σαρρη	Κατερινα	Παιχνίδια στο Πιάνο	€ 12,00	Μουσική	Greek
Lebrun	François	L'Histoire De La France	€ 75,00	Histoire	French
நேரு	ஜவஹர்லால்	ஆசிய ஜோதி	INR 250	சரித்திரம்	Tamil
Franklin	Benjamin	Un Américain Autobiographie	$ 19.95	Autobiographie	French
بهنسي ، د	حفيف	العمارة عبر التاريخ	SAR 95	معماري	Arabic

Figure 1. Multilingual Books Catalog Table

Organization

The remainder of this paper is organized as follows: Section 2 gives an overview of into multilingual query processing. The new multilingual data type, operator algebra and cost models are proposed in Section 3. The core implementation of the operators in PostgreSQL is described in Section 4, while Section 5 presents the performance of this implementation. Related work is reviewed in Section 6, and our conclusions are summarized in Section 7.

2 Background on Multilingual Operators

Consider a hypothetical e-Commerce application, *Books.com*, that sells books across the globe; the Book table storing data in multiple languages may be as shown in Figure 1. This table may be considered as a logical view assembled from data sourced off several databases (each aligned with the local language needs), but queried in a unified manner by multilingual users.

2.1 Multilingual Name Matching

In this environment, suppose a user wants to search for the works of an author in all (or a specified set of) languages. An example of LexEQUAL, the operator proposed in [13] to support this requirement, is shown in Figure 2. This operator takes an input name in one language (`'Nehru'` in `English` in the example), and returns all *phonemically close* names in the user-specified set of languages (`English`, `Tamil` and `Hindi`, in the example). Though restricted to proper names, such matching represents a significant part of the user query strings in text databases and search engines, as proper and generic names constitute *a fifth of normal corpora* [12].

```
SELECT Author,Title,Language FROM Book
WHERE Author LexEQUAL 'Nehru'
IN { English, Hindi, Tamil }
```

Author	Title	Language
Nehru	Letters to My Daughter	English
நேரு	ஆசிய ஜோதி	Tamil
नेहरू	भारत एक खोज	Hindi

Figure 2. Multilingual Name Query / Result

More formally, the LexEQUAL operator, hereafter referred to as Ψ for notational convenience, is defined as follows:

Definition 2.1: $(s_i \Psi s_j) \iff (\text{editdistance}(p_i, p_j) \leq t)$, where p_x is the phonetic string corresponding to the multilingual name string s_x, and t is the allowable threshold for mismatch.

Converters to transform multilingual strings to their respective phonetic strings in a canonical IPA [25] alphabet are employed for implementing the Ψ operator, as outlined in Figure 3. The operator accepts two multilingual text strings, and a match threshold value, as inputs. The strings are first transformed to their equivalent phonemic strings using the transform function, and if the edit distance between the phonemic strings is less than the threshold value, a true is returned, false otherwise. In our current implementation, the editdistance function computes the standard *Levenshtein* edit distance between the two strings. The mismatch threshold, t, is specified as a user-settable parameter.

$\Psi\ (S_l, S_r, t)$
Input: *Multilingual Strings* S_l, S_r; *Threshold* t
Output: true, false
1. $P_l \leftarrow$ transform(S_l); $P_r \leftarrow$ transform (S_r);
2. **if** editdistance(P_l, P_r) $\leq t$ **then**
return true **else return** false;

Figure 3. The Ψ Operator Algorithm

The Ψ operator is functionally analogous to the standard database equality operator, but in the *phonetic* domain. Very importantly, the Ψ operator may also be used for *joining* multilingual table attributes (as shown later in Example 3.3).

2.2 Multilingual Concept Matching

We now turn our attention to a user who wishes to retrieve all `History` books in a set of languages of his/her choice – note that unlike proper names, `History` is a *concept*. An example of SemEQUAL, the operator proposed in [14] to support this requirement, is shown in Figure 4. This operator takes the query in one language (`'History'` in `English`) and returns all records in user specified output languages (namely, `English`, `French` and `Tamil`), with categories that match any of the subclasses of `'History'`. Note that all records in the result set have a *Category* value that is equivalent to or is *subsumed* by `History`[2]. The interlinked multilingual taxonomical hierarchies required to produce such outputs are rapidly becoming available as a result of the synergistic WordNet projects in progress around the globe [28, 23, 24].

SELECT *Author,Title,Category* FROM *Book*
WHERE *Category* SemEQUAL *'History'*
IN {*English, French, Tamil* }

Author	Title	Category
Durant	History of civilization	History
Lebrun	L'Histoire de la France	History
[illegible]	[illegible]	[illegible]
Nehru	Letters to My Daughter	Autobiography
Gilderhus	History and Historians	Historiography
Franklin	Un Américain Autobiographie	Autobiography

Figure 4. Multilingual Concept Query / Result

Denoting an inter-linked set of taxonomic hierarchies of the categorical values in multiple languages by $\mathcal{H}_{\mathcal{ML}}$, the SemEQUAL operator (hereafter referred to as Φ for notational convenience) is formally defined as follows:

[2] `Historiography` (the study of history writing and written histories) and `Autobiography` are considered as specialized branches of `History` itself. The third record has as category the value *Charitram* in Tamil, meaning `History`.

Definition 2.2: Given two multilingual noun strings w_i and w_j, and the interlinked multilingual taxonomical hierarchy $\mathcal{H}_{\mathcal{ML}}$, $(w_i \Phi w_j) \iff (w_i \cap \mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(w_j) \neq \phi)$, where $\mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(x)$ computes the transitive closure of x in $\mathcal{H}_{\mathcal{ML}}$.

The skeleton of the algorithm to semantically match a pair of multilingual strings is outlined in Figure 5. Here, the Φ function takes two multilingual strings S_{Data} and S_{Query} as input. It returns true if the string S_{Data} is a member of the transitive closure of S_{Query} in the multilingual taxonomic hierarchy $\mathcal{H}_{\mathcal{ML}}$.

$\Phi\ (S_{Data}, S_{Query}, \mathcal{H}_{\mathcal{ML}})$
Input: *Multilingual Strings* S_{Data}, S_{Query}
Taxonomic Hierarchy $\mathcal{H}_{\mathcal{ML}}$
Output: true or false
1. $\mathcal{TC}_{\mathcal{Q}} \leftarrow$ TransitiveClosure($\mathcal{S}_{Query}, \mathcal{H}_{\mathcal{ML}}$);
2. **if** ($\{\mathcal{S}_{Data}\} \cap \mathcal{TC}_{\mathcal{Q}}$) **return** true **else return** false;

Figure 5. The Φ Operator Algorithm

Variations of the above multilingual operators are described in [13, 14] – we do not consider them here since they are not directly relevant to the focus of this paper.

3 Multilingual Relational Algebra

In this section, we propose MURAL (MUltilingual Relational ALgebra), a domain-specific query algebra to match multilingual text data across languages. We first present a new datatype, UniText, and then explain how the Ψ and Φ operators evaluate on this datatype. The Mural algebra is shown to be relationally complete in [15], facilitating integration with current database engines.

3.1 UniText Data Type

All the basic types of relational systems are preserved in the Mural algebra, and a new datatype – UniText – is added to store multilingual text strings. UniText enhances the normal Text type by additionally storing an identifier of the language of the string. That is, UniText is a 2-tuple, where the first component is the text string in a standardized encoding (referred to as Text), and the second is an identifier for the language of the string (referred to as LangID), assuming that the text string is in a single language. The explicit identifier is necessary as several languages share a script, and a string may have different pronunciations or meanings, depending on its language. In addition to the language identifier, UniText can be made to optionally store additional information, such as the materialized phoneme strings corresponding to the multilingual strings, to improve the runtime performance of queries on this datatype.

Example 3.1: <"*A Sample String*", English>, < "*Une Corde Témoin*", French> and <"உவமான சரம்", Tamil> are of proposed UniText datatype, where the first part of each of the UniText values is a Unicode string, and the second part is its language identifier.

It is apparent that the standard database operations need to be redefined to function correctly on the UniText datatype. For this purpose, we introduce two simple operators: 1) *Composing Operator* (denoted as $\succ$) that can compose a UniText datatype out of a given Unicode string and its language identifier; and 2) An inverse *Decomposing Operator* (denoted as $\prec$) that decomposes a UniText data to a Unicode String and its language identifier.

3.2 Multilingual Operators on UniText

In the context of the UniText data type, the multilingual homophonic Ψ operator is defined as follows:

$\Psi : Set< U_1 > \times Set< U_2 > \rightarrow Set< U_1, U_2, dist >$

The input is two sets of UniText strings, and the output is the Cartesian product of the two sets, with each result tuple tagged with the *edit-distance* between the phonemic strings corresponding to the two multilingual strings. This operation preserves both the input strings, which are available for subsequent operations. Either of the input attributes may be removed by subsequent projection operations and this is left to the user. The materialization of the phoneme strings is left unspecified, as it does not affect the functionality of the operator.

Example 3.2: To select the (last) names of authors from the Book table that are phonetically close (within a threshold distance of 2) to `Nehru`, the query expression is as follows:

$$\Pi_{Book.Author}(\sigma_{dist<2}(\Psi(Book.Author, \{\text{“}\texttt{Nehru}\text{”}\})))$$

Example 3.3: Assuming that we have a Publisher table with PName as a multilingual attribute storing the publisher's name, the following query selects authors who have (last) names similar to that of a publisher (within a threshold distance of 2):

$$\Pi_{Book.Author, Publisher.PName}(\sigma_{dist<2} (\Psi(Book.Author, Publisher.PName)))$$

Turning our attention to the multilingual homosemic Φ operator, it is defined as follows in the context of the UniText data type:

$\Phi : Set< U_1 > \times Set< U_2 > \rightarrow Set< U_1, U_2, match >$

The input is two sets of UniText strings, and the output is the Cartesian product of the two sets, with each result tuple tagged with a boolean value $match$, which is set to true, if $u_1 \in \mathcal{T}_{\mathcal{H}}(u_2)$, where $u_1 \in U_1$ and $u_2 \in U_2$. This operation preserves both the input strings, which are available for subsequent operations. Removal of either of the input attributes is by projection operations, which is left to the user.

Example 3.4: The query to retrieve all the book titles that are categorized under `History` in the Book table, is specified as follows:

$$\Pi_{Book.Title}(\sigma_{match=\text{true}} (\Phi(Book.Category, \{\text{“}\texttt{History}\text{”}\})))$$

3.2.1 Composition of Operators

An overview of the composition of the Ψ and Φ operators with each other, and with the traditional binary relational algebra operators, namely $\times, -$ and $\cup$, and the aggregation operator Δ, is shown in Table 1 (details in [15]). These composition rules are incorporated in the optimizer to facilitate generation and evaluation of alternative execution plans for a given query.

Oper	Commutes	Associates	Distributes Over
Ψ	Yes	Yes	$\times, \cup, -, \Xi, \Phi, \Delta$
Φ	No	Yes	$\times, \cup, \Xi, \Delta$

Table 1. Interaction between Operators

As an aside, we note that in addition to the above operators, all text comparison operations (specifically, $=, \neq, <, >$) may be applied to the UniText datatype; in such cases, the operator functions solely on the Text component of the UniText. This was implemented in order to make UniText support normal Text operations as well. Further, we also support a UniText comparison operator, Ξ, to support direct equality comparisons of both components of the UniText datatype. These issues are discussed in more detail in [15].

3.3 UniText Operator Cost Models

We now move on to presenting the cost models for the Ψ and Φ operators. There are two variations possible for each of these operators: **scan** type, which is of the form <*Attr*> Oper <*Const*>, and **join** type, which is of the form <*Attr*> Oper <*Attr*>. Based on the notation defined in Table 2, the cost models for these various combinations are quantified (in *big-O* notation) in Table 3.

The cost models were arrived at based on our implementation of the Ψ and Φ operators, as follows: All *edit-distance* computations were implemented using the diagonal transition [16] algorithm. The modeling of Ψ operation costs with indexes corresponds to indexes being created on the materialized phoneme strings. Since the phoneme

Symbol	Represents
LHS/RHS Operand parameters are marked by L/R subscripts.	
R_L, R_R	No. of Records
l_L, l_R	Avg. length of Records
P_L, P_R	No. of Pages
A_L, A_R	No. of Pages for Approximate Index
k	Ψ Error Tolerance, as a fraction, $\in (0, 1)$
σ	Ψ Size of the Alphabet (= $\|\Sigma\|$)
$R_{\mathcal{H}}$	No. of Records storing $\mathcal{H}$ (=$\|\mathcal{H}_{\mathcal{MC}}\|$)
$P_{\mathcal{H}}$	No. of Pages storing $\mathcal{H}$
$E_{\mathcal{H}}$	No. of Pages storing Index of $\mathcal{H}$
f, h	Average *fan-out* and *height* of $\mathcal{H}$

Table 2. Symbols used in Analysis

	Remarks	Complexity	Disk I/O
scan-type Operations			
Ψ	No Index	$R_L l_L k/\sqrt{\sigma}$	P_L
Ψ	Approx. Index	$R_L l_L k^2/\sqrt{\sigma}$	A_L
Φ	No Index	R_L+$R_{\mathcal{H}}(h$+1)	$P_{\mathcal{H}}(h$+1)
Φ	Index on $\mathcal{H}$	R_L+$lnE_{\mathcal{H}}(h$+1)	$lnE_{\mathcal{H}}(h$+1)
join-type Operations			
Ψ	No Index	$R_L R_R l_L k/\sqrt{\sigma}$	$3(P_L$+$P_R)$
Ψ	Approx. Index	$R_L R_R l_L k^2/\sqrt{\sigma}$	A_L+A_R
Φ	No Index	R_L+R_R+ $R_R R_{\mathcal{H}}(h$+1)	$3(P_L$+$P_R)$ +$P_{\mathcal{H}}$
Φ	Index on $\mathcal{H}$	R_L+R_R+ $R_R log E_{\mathcal{H}}(h$+1)	$3(P_L$+$P_R)$ +$E_{\mathcal{H}}$

Table 3. Cost Models for Operators

strings are materialized at string insertion time, the overheads of materialization are not included in the cost computation. For approximate indexes (such as the M-Tree), the fraction of the database scanned was approximated by a linear function on the error threshold, based on the empirical evaluations presented in [15].

3.4 UniText Operator Output Cardinalities

We now turn our attention to outlining the heuristics used to estimate the cardinalities of the operator result relations.

3.4.1 Ψ Output Cardinality Estimation

Estimation of result cardinality in the metric matching domain is a problem that has received attention in recent times (e.g. [2]). In our implementation, we leveraged the natively supported end-biased histograms [8] in the PostgreSQL database system, which are known to be practical and effective summary structures for a wide range of estimation problems. The cardinality estimation for the approximate matching proceeds as follows [9]: The ten most-frequent values of the phonemic string attribute are stored, along with their frequencies, explicitly in the histogram associated with that attribute. The selectivities based on the approximate matching (with the user specified threshold for the specific query) from among these most frequent values are used as the first approximation for the selectivity of the entire query. Further, a fraction corresponding to the threshold factor (based on the empirical study of approximate matching presented in [15]) is used to inflate the selectivity estimation to model the fuzzy matching of non-frequent values of the attribute. The resulting selectivity is used as the selectivity of the Ψ operator.

We hasten to add that, very recently in [11], a methodology that uses specialized histograms inside the engine has been proposed to estimate matching cardinalities in the metric space. However, for our experimental evaluation, we have used only the natively supported histograms in the Postgres system, leaving the implementation of the specialized structures to future work.

3.4.2 Φ Output Cardinality Estimation

The output size of the Φ operator is estimated using the notation in Table 2, as follows: Given that the average height of the multilingual taxonomical hierarchy $\mathcal{H}$ is h, the selectivity of a **scan** predicate is given by $(h+1)/|\mathcal{H}|$, and the selectivity of a **join** predicate is given by $R_L(h+1)/|\mathcal{H}|$. In the case where closures are pre-computed and stored, the estimation accuracy may be improved further by using the exact values as $|\mathcal{T}_{\mathcal{H}}(v)|\ /|\mathcal{H}|$ and $R_L|\mathcal{T}_{\mathcal{H}}(v)|\ /|\mathcal{H}|$, respectively, where $|\mathcal{T}_{\mathcal{H}}(v)|$ is the size of the closure of v in $\mathcal{H}$.

4 Core Implementation in PostgreSQL

In this section, we discuss strategies for incorporating the multilingual functionalities as first-class operators in the PostgreSQL open-source database system, with a core implementation of all the elements of the Mural algebra in the database kernel. Subsequently, in Section 5, we analyse the performance of this implementation.

4.1 System Setup

Our implementation platform was PostgreSQL Version 7.4.3, operating on a stand-alone standard Pentium-IV workstation (2.3GHz) with 1 GB main memory and 80 GB hard disk, running the RedHat Linux (Version 2.4) operating system. The operator implementation was done using C, and hence is highly portable to diverse systems and platforms.

Our choice of PostgreSQL was influenced by its object-relational architecture, featuring strong support for extensible datatypes, functions, operators, and index methods. In addition to the multilingual operators, we implemented M-tree [3], a specialized index structure for metric spaces, using the GiST (Generalized Search Tree) index generator available in PostgreSQL. The advantage of GiST is that it can be used for quickly developing custom access methods with specific semantics for newly added datatypes.

4.2 Ψ Operator Implementation

Although PostgreSQL has a new operation addition facility, this is restricted to binary operators, and therefore cannot be directly used to implement Ψ, which as discussed previously, is a *tertiary* operator. Therefore, we used the workaround of implementing Ψ as a binary operator, making the third input, the error threshold parameter, a user-settable value in a system table. The value of the parameter for matching may be globally set by the administrators, based on the requirements of a domain or set by a user for the current session. An appropriately modified version of the Ψ operator presented in Figure 3 (one that takes the two strings as operator inputs and the threshold from the system table) was implemented.

The open-source Dhvani [22] text-to-phoneme engine that is tuned for two Indic languages – Hindi and Kannada – was integrated with PostgreSQL, after appropriate modifications to output the phonemic strings in IPA [25] alphabet. From an efficiency point of view, the phonemic strings corresponding to the multilingual strings were materialized to avoid repeated conversions (as in the case of join processing). In the optimizer module, the cost of the operator was set to the formulae given in Table 3, and the selectivities were estimated using the methodology outlined in Section 3.4.

4.2.1 M-Tree Index

The Ψ operator is *inherently fuzzy* since it involves approximate matching of phonemic strings. Speeding up of such fuzzy matching is not feasible with the standard and ubiquitious B-tree indices. While a large number of structures have been proposed in the literature, we chose to implement the M-Tree [3] index structure, since it is a height-balanced structure and proposed specifically for database solutions. The M-Tree index was implemented in PostgreSQL using its GiST feature [6] that provides a framework for managing a balanced index structure that can be extended to support index semantics. Further, we specifically chose the *random-split* alternative for splitting nodes while expanding the M-tree, since it offers the best index modification time and has insignificant incremental disk I/O compared to other alternatives that are more computationally intensive.

Note that though the Slim-Tree [19] is considered more efficient for certain data profiles, it could not be considered by us as it requires explicit re-insertion of specific elements on designated nodes of the index tree; such a feature is yet to be supported in the GiST implementation.

The version of PostgreSQL system that is used for our implementation does not support concurrency for GiST trees, due to the lack of write-ahead logging of the index structures. Hence, a crash could render the index structure inconsistent. Our experiments therefore focussed only on the query performance of the M-Tree index and not on its durability aspects. Further, since the forthcoming Version 8.1 is expected to have native support for GiST concurrency and recovery, we expect this to soon become a non-issue.

4.3 Φ Operator Implementation

The Φ operator was also added to PostgreSQL as a binary join operator, using the operator addition facility in the PostgreSQL system. The homosemic matching functionality, as given in Figure 5, was implemented in C. Computing transitive closures (Step 1 of the Φ algorithm) is known to be a hard problem [1, 7] in relational database systems. To make the computation more efficient, we read the WordNet taxonomical hierarchies from the database tables and pinned them in the main memory. This strategy was made viable by the fact that the size of the English noun taxonomic hierarchy (the most developed one currently) is only about 4 MB; further, the WordNet hierarchies of different languages have been found to have similar structural and size characteristics [14], and hence it appears reasonable to assume that they would have similar storage requirements. Even in highly multilingual environments, we do not anticipate more than a few tens of languages to be concurrently used in practice – therefore, pinning all the WordNets of these languages in the main memory may be well within the capacities of current database and web servers.

Every time a closure for a RHS attribute value is computed, it is materialized as a hash table in the main memory, which is helpful in two ways: Firstly, the second step of checking set-membership of a set of LHS attribute values, becomes much faster as the same hash table is used for all LHS values. Secondly, the hash table is checked for possible reuse for several RHS values; when a closure computation is needed, the materialized hash table is verified to check if the closure is already available for the same RHS value. Thus, a class of operators that need to process several LHS operand values for a given set of RHS operand values may amortize the cost of computing and materializing the closures. For example, nested-loops join queries using the Φ operator may be made more efficient by making the RHS operand the outer table, thus using the same closure for all inner table values. Further optimization may be achieved by

sorting the RHS values and computing the closure only for unique values.

4.3.1 Speeding-up Closure Processing

Very recently, the Hopi-index has been proposed for speeding up processing of path expressions in XML document collections [17]. This index is based on computing, storing and querying using a 2-hop cover for a given network. An interesting open question is whether these techniques can also be used for speeding up the closure processing in relational environments. At the current time, we have not pursued this technique in our implementation, since it requires storage of the index in the database tables, and runtime access to this index table (using other SQL statements). In a native implementation, such SQL processing is bound to add to the overheads, especially when the closures need to be computed inside another SQL statement (say, a join). However, it may still be possible to customize the Hopi index for WordNet hierarchies, and we plan to explore the viability of such techniques in our future work.

5 Performance Experiments

In this section, we outline our performance experiments with the above implementation of multilingual operators on PostgreSQL. We start off with validating the operator cost models and then move on to profiling the Ψ and Φ operators on both *scan* and *join* queries.

5.1 Data Setup

To benchmark the Ψ and Φ operators in the PostgreSQL database environment, we used the same datasets that were used in [13, 14]. Specifically, for evaluating the Ψ operator, a pre-tagged multilingual names data set with approximately $200,000$ records was used in the experiments. On the other hand, for evaluating the Φ operator, the entire English WordNet hierarchy with about $110,000$ word forms, $80,000$ synsets and $140,000$ relationships between them, was stored in the database. Specificaly, the WordNet hierarchy required about 4 MB of storage. Since the WordNets in other languages are in different stages of development, for performance experiments we *simulated* linked WordNets by replicating English WordNet in Unicode, and creating an equivalence link between corresponding synsets. This methodology appears acceptable for assessing performance with multiple WordNet hierarchies, as the structural and size characteristics of many WordNets were shown to be similar in [14]. Queries that compute closures of varying sizes were employed for profiling the Φ operator. All experiments were run on these datasets on the previously-mentioned workstation, quiesced of all other activities.

The UniText datatype and operators were added to the database system, without affecting the existing datatypes and features of the system. After making all modifications to the database system, we confirmed that the existing performance of the system is not affected adversely by the new modifications. This was done by comparing the performance of the standard regression test suites on our modified PostgreSQL system, against its performance on the original PostgreSQL server, in the same test machine – we found no statistically significant degradation in performance.

5.2 Optimizer Prediction Accuracy

In order to ascertain the quality of our cost models and the accuracy of the optimizer in predicting the query costs, we experimented with a range of multilingual queries that join two tables using a multilingual operator. Further, the output of the queries were collapsed using `count (*)`, to nullify the effect of I/O from the query processing time. First, a set of tables of varying characteristics (in terms of attribute count and attribute size) were created and populated with different data sets (with varying record counts and number of database blocks). Then the selected queries were run over a range of selectivities (by appropriately setting the *threshold* parameters) on all these tables. In addition, between different runs of the same query, duplicate records were introduced in the tables and the histograms rebuilt, in order to vary the input to the optimizer, for subsequent queries. For each query, we recorded the optimizer predicted cost and the actual runtime of the query, and Figure 6 plots the correlation between these statistics.

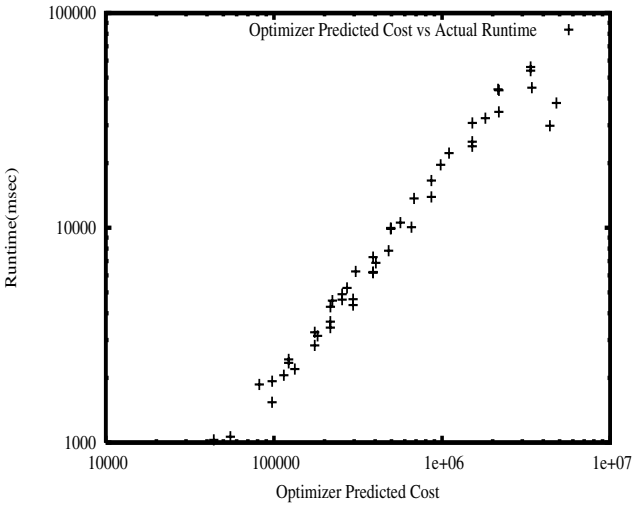

Figure 6. Cost Prediction Quality

As can be seen in Figure 6, there is a reasonably good correlation between the predicted optimizer costs and the

actual runtimes of the queries. The computed correlation coefficient on the plot is well over 0.9, indicating reasonably accurate cost models. Although there is some error in estimating large queries, we observed that this error is in the same range as in the case of estimation with the standard *monolingual* operators.

5.2.1 A Motivating Optimization Example

We illustrate the power of the optimization strategies to distinguish between efficient and inefficient executions with the new operators, by the following example:

Example 5.1: Assume a relational schema that has Author (A) table with AuthorID and AName, Publisher (P) table with PublisherID and PName, and Book (B) table with BookID and foreign keys to Author and Publisher. Now consider the query – *Find the books whose author's name sounds like that of a publisher's name (match threshold of 3)*. For this query, both of the following expressions (also shown pictorially in Figure 7) capture the query semantics:

Plan 1: $\Pi_{B.BookID}$
$(\sigma_{(Threshold \leq 3)}(\Psi_{A.AName,P.PName}$
(P, $(A \bowtie_{BookID} B))))$

Plan 2: $\Pi_{B.BookID}(B \bowtie_{BookID}$
$(\sigma_{(Threshold \leq 3)}(\Psi_{A.AName,P.PName}(P, A))))$

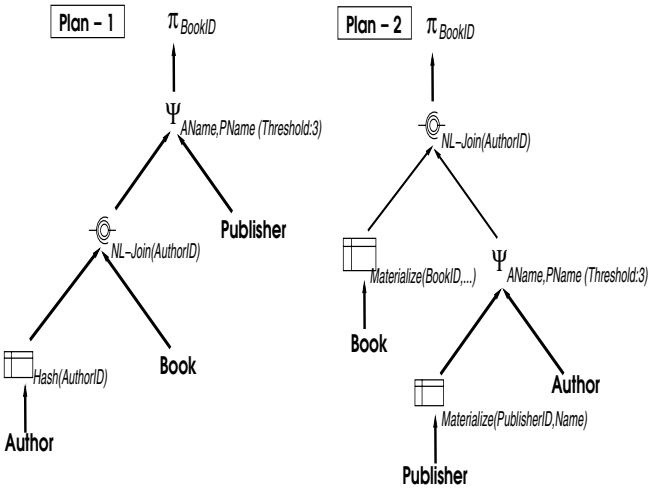

Figure 7. Query Plan for Example 5.1

We created the tables Author, Book and Publisher, along the lines of our examples in the previous sections, and forced the optimizer to evaluate and run two different execution plans for the above query by enabling or disabling different optimizer options. For each plan, we recorded the optimizer predicted cost and measured the runtime of the execution. The optimizer predicted cost and the runtime for Plan 1 are 2,439,370 and 82.15 seconds, respectively. The corresponding figures for Plan 2 are 7,513,852 and 2338.31 seconds, respectively. Clearly, Plan 1 is superior (in terms of runtime, a post-facto observation) and is chosen (due to its lower predicted cost by the optimizer) for execution. Further, we were able to force different query execution plans by modifying the characteristics of the underlying tables, confirming the use of our cost models and optimization strategies by the optimizer.

5.3 Performance of Ψ Implementation

After implementing the Ψ operator natively, the **scan** and **join** queries were run on the multilingual data sets and the average of their performance on a large number of runs, for a constant threshold value of 3, is given in Table 4. In addition, the performance of the same queries with the metric M-Tree index on the materialized phoneme strings are also presented in the same table.

To place the core performance numbers in context, we present in Table 4 the corresponding performance statistics when the multilingual operators are implemented *outside-the-server* using standard database features (PL/SQL procedures, SQL scripts and recursive SQL constructs). The index used here is the Metric-Distance-Index (MDI) which can be implemented using the standard B-tree index, as described in [15]. It should be noted that the performance experiments were run after the phoneme strings corresponding to the multilingual strings had been materialized and stored explicitly in the table.

Implementation Type	Query Type	Scan (*Sec.*)	Join (*Sec.*)
Core Implementation	*No Index*	5.20	1.97
	M-Tree Index	4.24	1.92
Outside-Server Implementation	*No Index*	3618	453
	MDI Index	498	169

Table 4. Performance of Ψ Implementation

From Table 4, two main observations can be made: First, the native performance of the operator is about *two orders of magnitude* better than even the index-based outside-the-server performance. The primary reason for the improvement is that the overheads due to the UDF invocations and execution in a separate process space are largely eliminated. In addition to such obvious performance improvements, the

optimizer can also select effective query execution plans in the native implementation. We can therefore conclude that there are very substantial benefits possible by pushing multilingual operators into the core of relational engines.

Second, we found that the M-Tree metric index only marginally improved the performance of the approximate matching queries, although it had done extremely well for the situations presented in [3]. Our analysis indicates that the lack of performance improvement in our environment arises from poor pruning efficiency, primarily caused by the following factors: (a) the high dimensionality of the string matching due to the long string lengths, and (b) the coarse edit-distance measure, as opposed to the continuous Euclidean distance measure. In our future work, we plan to experiment with alternate index structures and techniques to improve the performance of such queries.

5.4 Performance of Φ Implementation

Turning our attention to the homosemic functionality, the native implementation of the Φ operator was done as indicated in Section 4.3, and a set of queries computing closures of various cardinalities in the WordNet noun hierarchy were run to profile its performance. The performance of the queries with a B+Tree index on the parent attribute of the taxonomy table was also evaluated. Again, to put the core performance into context, we additionally evaluated an outside-the-server UDF-based implementation in the PL/SQL programming environment.

The performance statistics for both the core and outside-the-server implementations of the Φ operator are shown in Figure 8, as a function of the closure cardinality with regard to the WordNet taxonomic hierarchy. Note that the graph is shown on a *log-log* scale.

We see in this figure that in the absence of indices, the core implementation is about an order of magnitude faster than the outside-the-server implementation. With an index, the performance improvement jumps to over two orders of magnitude. In addition, even in absolute terms, such performance with a few tens of milliseconds response time for a typical closure size of around $2,000$ [14], appears sufficient for practical deployments.

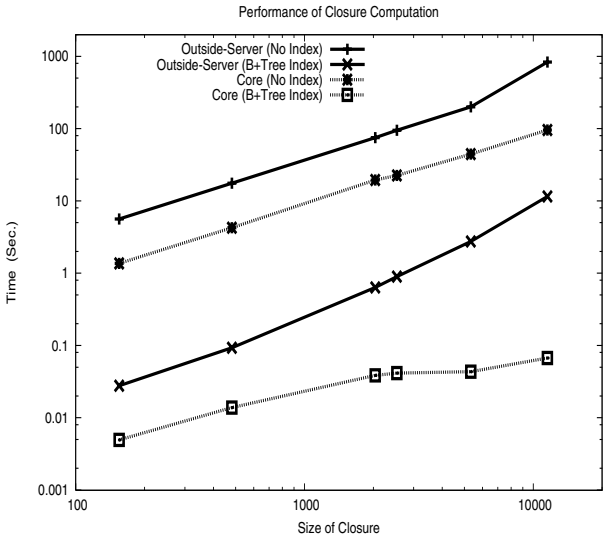

Figure 8. Performance of Φ Implementation

6 Related Research

To the best of our knowledge, the work described here represents the first practical implementation of multilingual functionalities as first-class operators in the database kernel. Our earlier efforts in [13, 14] had focussed primarily on defining useful multilingual functionalities, but had not considered issues such as operator algebras, cost models, and indices, that are required for an integrated core implementation.

A substantial body of literature is available from the information research community in the areas of knowledge-based and natural-language based retrieval. While the techniques employed are diverse, they are not directly applicable to attribute level data in the OLTP type environments which we consider. Phonetic matching of English strings was discussed in [20], but their focus is on the linguistic issues and quality of the resulting match. Also, their main-memory implementation does not raise issues related to on-disk database processing. Algebras specific to a domain are available, such as PiQA[18] in Bioinformatics and TAX[10] in XML. Our approach parallels such efforts in the multilingual text domain.

Recently, a major database vendor has proposed and demonstrated matching functionality using ontologies [4]. While their implementation is not yet available in the released versions, we expect that our multilingual operators can easily leverage on such implementations. Further, we are heartened by this parallel effort in ontological query processing that is in convergence with our research methodology for semantic matching in the multilingual domain.

7 Conclusions

We have investigated, in this paper, implementation strategies for multilingual name matching and multilingual concept matching inside a relational database engine. Specifically, we presented the Mural query algebra which defines a multilingual datatype and operators, for intuitively expressing complex queries. Mural is relationally complete and can therefore be added to existing relational systems at little cost. We also defined all components that are required

for a full integration of the multilingual query algebra with the database engine, specifically the cost models, composition rules and cardinality estimation models for the new operators.

An approach was outlined for a *core* implementation of the multilingual functionality as first-class operators inside the open-source PostgreSQL database kernel. In order to optimize the performance of the homophonic Ψ operator, a metric M-Tree index was added using the GiST feature of PostgreSQL system. The homosemic Φ operator was implemented by pinning the WordNet taxonomic hierarchies in the main memory since the computation of transitive closures is expensive in relational systems. As part of our ongoing work, we plan to implement the recently-proposed Hopi index[17] to further enhance the efficiency of this operator.

Our native implementation of the above functionalities was shown to improve the multilingual performance of the PostgreSQL database system by nearly two orders of magnitude over a comparable *outside-the-server* implementation. Equally importantly, the Mural algebra was shown to be effective in choosing efficient plans by leveraging the relational query optimizer effectively. However, our experiments also indicated that the M-tree index is only marginally effective for approximate string matchings, and we plan to experiment with alternate structures in our future research.

In closing, our work attempts to take the first practical step towards the ultimate goal of realizing *natural-language-neutral* database engines. Overall, our results show that the multilingual functionality may be added in a practical and efficient manner to current relational engines. We hope that our experience would lead to native support of such operators in the next generation of commercial database engines.

Acknowledgements

This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science and Technology, Govt. of India. We thank Mitesh Jat and Rupesh Bajaj for their assistance in the experimental study.

References

[1] R. Agrawal and H. V. Jagadish. Direct algorithms for computing Transitive Closure of DB Relations. *Proc. of 13th VLDB Conf.*, 1987.

[2] S. Chaudhuri, V. Ganti and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. *Proc. of 20th IEEE ICDE Conf.*, 2004.

[3] P. Ciaccia, M. Patella and P. Zezula. M-Tree: An Efficient Access Method for Similarity Search in Metric Space. *Proc. of 23rd VLDB Conf.*, 1997.

[4] S. Das, E. I. Chong, G. Eadon and J. Srinivasan. Supporting Ontology-based Semantic Matching in RDBMS. *Proc. of 30th VLDB Conf.*, 2004.

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

[6] J. M. Hellerstein, J. F. Naughton and A. Pfeffer. Generalized Search Trees for Database Systems. *Proc. of 21st VLDB Conf.*, 1995.

[7] Y. Ioannidis. On the Computation of TC of Relational Operators. *Proc. of 12th VLDB Conf.*, 1986.

[8] Y. Ioannidis. Universality of Serial Histograms. *Proc. of 19th VLDB Conf.*, 1993.

[9] Y. Ioannidis and V. Poosala. Histogram-based Solutions to Diverse Database Estimation Problems. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[10] H. V. Jagadish, L. Lakshmanan, D. Srivastava and K. Thompson. TAX: A Tree Algebra for XML. *Proc. of DBPL Conf.*, 2001.

[11] L. Jin and C. Li. Selectivity Estimation for Fuzzy String Predicates in Large Data Sets. *Proc. of 31st VLDB Conf.*, 2005.

[12] D. Jurafskey and J. Martin. Speech and Language Processing. *Pearson Education*, 2000.

[13] A. Kumaran and J. R. Haritsa. LexEQUAL: Supporting Multiscript Matching in Database Systems. *Proc. of 9th EDBT Conf.*, 2004.

[14] A. Kumaran and J. R. Haritsa. SemEQUAL: Multilingual Semantic Matching in Relational Systems. *Proc. of 10th DASFAA Conf.*, 2005.

[15] A. Kumaran, P. K. Chowdary and J. R. Haritsa. On Pushing Multilingual Query Operators into Relational Engines. *Tech. Report TR-2005-01, DSL/SERC, Indian Inst. of Science*, 2005.

[16] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1), 2001.

[17] R. Schenkel, A. Theobald and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collection. *Proc. of 9th EDBT Conf.*, 2004.

[18] S. Tata and J. M. Patel. PiQA: An Algebra for Querying Protein Data Sets. *Proc. of 15th SSDBM Conf.*, 2003.

[19] C. Traina Jr., A. Traina, B. Seeger and C. Faloutsos. Slim-trees: High Performance Metric Trees Minimizing Overlap Between Nodes. *Proc. of 7th EDBT Conf.*, 2000.

[20] J. Zobel and P. Dart. Phonetic String Matching: Lessons from Information Retrieval. *Proc. of 19th ACM SIGIR Conf.*, 1996.

[21] The Computer Scope Ltd. *http://www.NUA.ie/Surveys*.

[22] Dhvani - A Text-to-Speech System for Indian Languages. *http://dhvani.sourceforge.net*.

[23] Euro-WordNet. *www.illc.uva.nl/EuroWordNet*.

[24] Indo-WordNet. *www.cfilt.iitb.ac.in*.

[25] IPA. *http://www.arts.gla.ac.uk/IPA/ipa.html*.

[26] PostgreSQL Database System. *http://www.postgresql.com*.

[27] The WebFountain. *http://www.almaden.ibm.com/WebFountain*.

[28] The WordNet. *http://www.cogsci.princeton.edu/~wn*.