On Pushing Multilingual Query Operators inside Relational Engines

A. Kumaran Pavan Kumar Chowdary Jayant R. Haritsa

Technical Report
TR-2005-01

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

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

On Pushing Multilingual Query Operators inside 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 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 them across different natural languages. To address this lacuna, we have recently proposed two new cross-lingual functionalities, LexEQUAL[26] and SemEQUAL[27], which support phoneme-based matching of names, and ontology-based matching of concepts, respectively.

In this paper, we investigate the implementation of these multilingual functionalities as first-class operators on relational engines, using PostgreSQL open-source database system. We first propose a new multilingual datatype, operators with their cost models and Mural, a multilingual query algebra. To verify the efficacy of our approach, these components have all been successfully implemented in the PostgreSQL. Further, to speed up multilingual processing, a metric index has been incorporated using the GiST feature of PostgreSQL. An *outside-the-server* implementation using existing features of the database system has also been done, to establish a baseline performance measure. Our experiments over representative multilingual datasets demonstrate orders-of-magnitude performance gains for the core implementation, in addition to being able to leverage on the well-developed relational query optimizer. To the best of our knowledge, our prototype system is the first practical attempt towards the ultimate goal of realizing *natural-language-neutral* database engines.

1 Introduction

The internet – the primary digital arena for information, interaction, entertainment and commerce, is expanding rapidly[1], in addition to turning multilingual steadily[2]. It is imperative that the key applications of the Internet, such as *e-Commerce* and *e-Governance* portals, must work across multiple natural languages, seamlessly. A critical requirement to achieve this goal is that the underlying data source – relational database management systems – should manage multilingual data effectively, efficiently and seamlessly. Current database systems do support the storage and management of multilingual data, but are not capable of querying across different natural languages. To address this lacuna, we have recently proposed two new multilingual functionalities – LexEQUAL [26] and SemEQUAL [27] – which support phoneme-based matching of names, and ontology-based matching of concepts, respectively.

While the above focussed primarily on the *outside-the-server* implementation of the multilingual functionalities, in this paper, we investigate their *core* implementation, as first-class operators on relational engines. To push the operators into the core engine, we propose a new multilingual datatype – Uniform (UNIcode FORMat), a set of multilingual operators similar to LexEQUAL and SemEQUAL and a query algebra – MURAL (MUltilingual Relational ALgebra) that defines uniform query semantics and offers an intuitive framework for declaratively expressing complex queries. Further, the algebra and the operator cost models, selectivity estimates and composition rules, are critical input for relational query optimizer, for the selection of efficient query execution plans.

To verify the efficacy of our approach, all the above components have been successfully implemented in the PostgreSQL [7] open-source relational database engine. In addition, to speed up the query processing, a metric index has been incorporated using the GiST feature of PostgreSQL. An *outside-the-server* implementation using existing features of the database system has also been done, to establish a baseline performance for comparison with core implementation. Our experiments on representative multilingual datasets demonstrate orders-of-magnitude performance gains for the core implementation over the outside-

[1] Internet user population is growing at a rate of 12 to 15% yearly[4].

[2] Two-thirds of the current Internet users are non-native English speakers [1] and it is predicted that the majority of web-published data will be multilingual by 2010 [10].

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
नेहरु	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	Arti Fini	Italian
Nehru	Jawaharlal	Letters to My Daughter	£ 15.00	Autobiography	English
無門	慧開	無門關	¥ 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
بهتسي ء د	عفيف	العمارة عبر التاريخ	SAR 95	معْماريّ	Arabic

Figure 1: **Sample *Books.com* Catalog**

the-sever approach. To the best of our knowledge, our prototype implementation on PostgreSQL system is the first practical attempt towards the ultimate goal of realizing *natural-language-neutral* database engines.

1.1 A Motivating Example

Consider a hypothetical e-Commerce application – *Books.com* that sells books across the globe, with a sample product catalog in multiple languages as shown in Figure 1. The product catalog shown may be considered as a logical view assembled from data sourced off several databases (each aligned with the local language needs), but searchable in a unified manner for multilingual users.

1.1.1 Multilingual Homophonic Matching

In this environment, suppose a user wants to search for the works of an author in all (or a specified set of) languages. The SQL:1999 compliant query requiring specification of the authors name in several languages is undesirable, due to the requirement of expertise and lexical resources in several languages. We propose a simple homophonic operator[3] – designated as Ψ in Figure 2 – that takes input name in one language (`English`, in the example shown), but returns all *phonemically close* names in the user-specified set of languages (`English`, `Hindi` and `Tamil`, in the example shown). We refer *matching on multilexical text strings, based on their phonemic equivalence* as Homophonic matching.

SELECT *Author,Title,Language* FROM *Books*
WHERE *Author* Ψ-Operator `'Nehru'`
IN { *English, Hindi, Tamil* }

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

Figure 2: **A Homophonic Query and Result Set**

1.1.2 Multilingual Homosemic Matching

Consider the query to retrieve all `History` books in a set of languages of users choice. A SQL:1999 compliant query, having the selection condition as Category = `"History"` would return only those books that have *Category* as `History`, in English. A multilingual user may be served better if all the History books in all the languages (or in a set of languages specified by her) are returned. A simple homosemic operator[4] – designated as Φ in Figure 3 – and the corresponding result set may be desirable.

SELECT *Author,Title,Category* FROM *Books*
WHERE *Category* Φ-Operator *'History'*
IN {*English, French, Tamil* }

Author	Title	Category
Durant	History of Civilization	History
Lebrun	L'Histoire De La France	Histoire
நேரு	ஆசிய ஜோதி	சரித்திரம்
Nehru	Letters to My Daughter	Autobiography
Gilderhus	History and Historians	Historiography
Franklin	Un Américain Autobiographie	Autobiographie

Figure 3: **A Homosemic Query and Result Set**

Note that in addition to all books with *Category* having a value equivalent to `History`, the categories that are *subsumed* by `History`[5] are also retrieved. We refer *matching text strings based on their meanings, irrespective of the languages*, as Homosemic Matching.

1.2 Contribution of this Paper

In this paper, our contributions are as follows:

- **Formal definition and analysis** of the multilingual functionality proposed earlier in [26, 27].
- **Proposal of operator cost models** and **selectivities** and Mural query algebra, for a core implementation of the multilingual functionality in the relational systems.
- **Core implementation** of the MURAL query algebra and demonstration of improvement in performance to the tune of 2 to 3 orders of magnitude.

[3] Along the lines of LexEQUAL functionality proposed in [26].

[4] Along the lines of SemEQUAL functionality proposed in [27].

[5] `Historiography` (the study of of history writing and written histories) and `Autobiography` are considered as specialized branches of `History` itself. The third record has as *category* the value $Ch\underline{a}ritr\underline{a}m$ in Tamil, meaning `History`.

1.3 Organization of The Paper

The remainder of this paper is organized as follows: Section 2 defines and analyses the new multilingual functionality. Section 3 proposes Mural algebra, along with operator cost models necessary for a core implementation. Section 4 outlines a *core* implementation of the functionalities and Section 5 presents the performance of this implementation. Section 6 outlines how the functionalities may be added to current systems using existing features. Finally, Section 7 concludes the paper, highlighting related research.

2 Multilingual Functionalities

In this section, we define the phonetic and semantic matching functionalities to match *multilingual text attributes*, formalizing the functionality given in [26, 27].

2.1 Homophonic Functionality Definition

Let L_i be a natural language with an alphabet Σ_i. Let s_i in language L_i be a string composed of characters from Σ_i, and let $\mathcal{S}_\mathcal{I}$ be set of all such s_i. Let $\mathcal{S} = \cup_\mathcal{I}\mathcal{S}_\mathcal{I}$, for a given set of languages. We also assume that the phoneme strings are encoded in the *International Phonetic Association* (*IPA*) [6] alphabet, namely, Σ_{IPA}. Every natural language string can be transformed to a phonetic string in the IPA alphabet (in line with the phonetic conventions of the language). A transformation, $\mathcal{T}_\mathcal{I}$, between a given language string s_i and a corresponding phonemic string p_i, is represented by $\mathcal{T}_\mathcal{I} : \mathcal{S}_\mathcal{I} \rightarrow \mathcal{S}_{\mathcal{IPA}}$. The union of such transformation functions $\mathcal{T}$ $(= \cup_i\mathcal{T}_\mathcal{I})$ in a set of desired languages, represented by $\mathcal{T} : \mathcal{S} \rightarrow \mathcal{S}_{\mathcal{IPA}}$, is assumed to be given as an input to the query processing engine.

Definition 2.1**:** Two strings $s_i \in S_I$ and $s_j \in S_J$ are *phonetically equal*, iff their phonemic representations p_i and p_j are the same.

Example 2.1**:** Given that the strings {"*Nero*", "*Nehru*" in English, "நேரு" in Tamil and "नेहरु" in Hindi} have corresponding phonemic representations {"nerou", "næhru", "næru" and "næhru"} respectively, only the English "*Nehru*" and the Hindi "नेहरु" are phonetically same. ◇

However, since the phoneme set used by different languages are seldom equal, it is almost impossible to match two multilingual strings phonetically. Hence in the phonetic domain, we define *phonemic closeness*, a weaker notion of equality as follows:

Definition 2.2**:** Two strings s_i and s_j are *phonetically close* if and only if $\{\text{editdistance}(p_i,p_j) \leq t\}$, where t is a error tolerance for match.

The error tolerance parameter t is a fraction of the input string lengths and usually defined as a symmetric function of the two input strings. Further, this parameter must be calibrated based on the characteristics of the data domain and may be set based on the requirements of application or the user. The homophonic matching Ψ operator, based on Definition 2.2, is defined as follows:

Definition 2.3**:** $\{s_i \Psi_t s_j\} \Longleftrightarrow \{\text{editdistance}(p_i,p_j) \leq t\}$. ◇

2.1.1 Ψ Operator Algorithm

Figure 4 outlines the implementation of the Ψ operator, assuming the necessary linguistic resources are available. The operator accepts two multilingual text strings and a match threshold value as input. 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. The editdistance function [19] computes the standard *Levenshtien* edit distance between them. The error threshold is specified as a user-set fraction of the length of the smaller of the two input strings.

Ψ (S_l, S_r, e)
Input: *Multilingual Strings* S_l, S_r; *Threshold* e
Output: true, false
1. $T_l \leftarrow \mathcal{T}_l(S_l)$; $T_r \leftarrow \mathcal{T}_r(S_r)$;
2. $Smaller \leftarrow (\mid T_l \mid \leq \mid T_r \mid ? \mid T_l \mid : \mid T_r \mid)$;
3. **if** editdistance$(T_l, T_r) \leq (e * Smaller)$ **then**
return true **else return** false;

Figure 4: **The Ψ Operator Algorithm**

2.1.2 Ψ Operator Properties

The Ψ operator defined as above is functionally analogous to database equality operator, but in the *phonetic* domain. The following are the properties of this operator.

Property 2.1**:** The Ψ operator is commutative.

Property 2.2**:** The Ψ operator commutes with projection, provided the attributes used in Ψ is preserved by the projection operator.

Property 2.3**:** The Ψ operator commutes with selection, sort and join operators.

Property 2.4: The Ψ operator commutes with aggregate operators, provided the aggregation preserves the phonetic attribute.

The first property follows immediately based on the definition, and assuming normal semantics for threshold measure. The second through fourth properties follow, since the values in the result set are not altered by the operator.

2.2 Homosemic Functionality Definition

The definitions in this section assume only that the values of an attribute are from a specified domain, $\mathcal{D}$[6], with a set of distinct semantic values. Within each domain $\mathcal{D}$, the semantic values are assumed to be arranged in a taxonomic hierarchy, $\mathcal{H}$ that defines `is-a` relationships among the atomic semantic values of the domain. Such network may be a collection of directed acyclic graphs. Given an atom x and a domain $\mathcal{D}$, the transitive closure of x in $\mathcal{D}$ is unique, and is denoted by $\mathcal{T}_\mathcal{D}(x)$. Similarly, the transitive closure of a set $X(= \{x_i | x_i \in \mathcal{D}\})$ is denoted by $\mathcal{T}_\mathcal{H}(X)$, and is defined as $\cup_i\mathcal{T}_\mathcal{D}(x_i)$, where $x_i \in X$. Assuming the above notation, we provide the following definitions for *semantic matching*:

[6]The domains may correspond to different areas of discourse: *Astronomy, Bio-Informatics, Linguistics*, etc.

Definition 2.4**:** Given a taxonomy $\mathcal{H}$ in domain $\mathcal{D}$ and two nodes x and y in $\mathcal{D}$, we define x *is-a* y, iff $x \in \mathcal{T}_{\mathcal{H}}(y)$.

Definition 2.5**:** Given $\mathcal{H}$ in domain $\mathcal{D}$ and two sets of nodes X and Y in $\mathcal{D}$, we define X *is-a* Y, iff $X \subseteq \mathcal{T}_{\mathcal{H}}(Y)$.

Since linguistic domain ontologies have low resolution power (that is, words with multiple meanings), we provide a weaker version of the semantic equality, as follows:

Definition 2.6**:** Given a taxonomic hierarchy $\mathcal{H}$ in domain $\mathcal{D}$ and two sets of nodes X and Y in a domain $\mathcal{D}$, we say X *is-possibly-a* Y, iff $X \cap \mathcal{T}_{\mathcal{H}}(Y) \neq \phi$.

Definition 2.6 is used for defining homosemic operator in the following section.

2.2.1 WordNet-based Taxonomic Hierarchies

WordNet [11] is a standard linguistic resource for English, that provides mappings between words and their meanings (called *synsets*). WordNet defines, among other things, the classification relationships between its synsets arranged in a taxonomical hierarchy. Complete WordNet is available for English and efforts are underway to develop WordNets in different languages, paralleling the English WordNet. Inter-linking of synsets between WordNets of different languages is available for some languages currently [3], and is planned for others [5]. Figure 5 shows a simplified hierarchy in English and German (in solid lines) and the interlinking of noun forms between English and German using ILI links (in dotted lines).

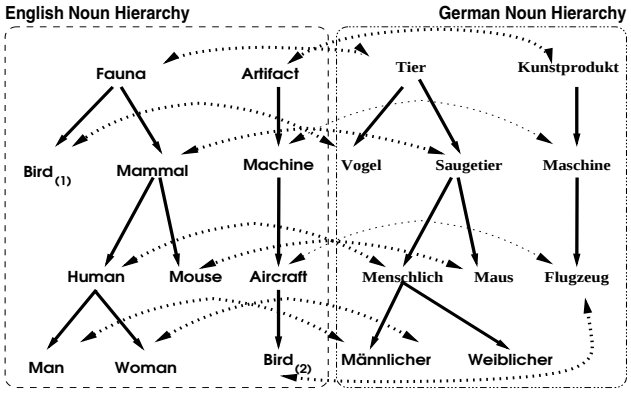

Figure 5: **Sample Interlinked WordNet Hierarchy**

Let $\mathcal{W}_{\mathcal{I}}$ be the WordNet of language L_i. Let $\mathcal{S}_{\mathcal{I}} = \cup_i s_i$. By definition, $\mathcal{W}_{\mathcal{I}}$ contains semantic primitives s_i, of L_i. The noun taxonomic hierarchy defines a set of DAGs, $\mathcal{H}_{\mathcal{I}}$ between the elements of $\mathcal{S}_{\mathcal{I}}$. A WordNet defines a mapping $\mathcal{M}_{\mathcal{I}}$, between a wordform ($w_i$) and its meanings ($\mathcal{M}_{\mathcal{I}}(w_i)$), as $\mathcal{M}_{\mathcal{I}}$:$w_i \rightarrow S_w$, where S_w is a set of s_i, $s_i \in \mathcal{S}_{\mathcal{I}}$. Consider the union of all semantic primitives of a set of languages, $\mathcal{S}_{\mathcal{ML}}$ (= $\cup_i \mathcal{S}_{\mathcal{I}}$) and the union of interrelationships between them $\mathcal{H}$ (= $\cup_i H_I$). Clearly, $\mathcal{H}$ is a set of DAGs, among the elements of $\mathcal{S}_{\mathcal{ML}}$. Augmenting $\mathcal{H}$ with the ILI links, a taxonomic network, $\mathcal{H}_{\mathcal{ML}}$, is created. This $\mathcal{H}_{\mathcal{ML}}$ is used for homosemic definition, as follows:

Definition 2.7: Given the multilingual taxonomic hierarchy $\mathcal{H}_{\mathcal{ML}}$, $\{w_i \Phi_{\mathcal{H}_{\mathcal{ML}}} w_j\}$ is true, if $\{S_I$ *is-possibly-a* $S_J\}$ under $\mathcal{H}_{\mathcal{ML}}$, where $S_I = \mathcal{M}_{\mathcal{I}}(w_i)$ and $S_J = \mathcal{M}_{\mathcal{J}}(w_j)$.

Note that this definition 2.6 guarantees not to produce *false-dismissals*, though it may introduce *false-positives* by matching on unintended word-senses.

Example 2.2: Both the predicates (Bird $\Phi_{\mathcal{H}_{\mathcal{ML}}}$ Fauna) and (Bird $\Phi_{\mathcal{H}_{\mathcal{ML}}}$ Artifact) evaluate to true, as the set of synsets of Bird namely, $\{Bird_1, Bird_2\}$ has a *non-empty* intersection with the closure of Fauna and Artifact. ◇

Example 2.3: Consider the predicate ($\text{Bird}_{English}$ $\Phi_{\mathcal{H}_{\mathcal{ML}}}$ $\text{Kunstprodukt}_{German}$): The answer is evaluated as, *Is* $\{\text{Bird}_1, \text{Bird}_2\} \cap \{$Kunstprodukt, Maschine, Flugzeug, Bird$\} \neq \phi$, which evaluates to true. ◇

2.2.2 The Φ Operator Algorithm

The skeleton of the Φ algorithm to match a pair of multilingual strings is outlined in Figure 6.

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

Figure 6: **The Φ Operator Algorithm**

The Φ function takes two multilingual strings S_{Data} and S_{Query} as input. It returns true, if LHS string maps to a semantic atom, that is some member of transitive closure of the RHS operand in the taxonomic network $\mathcal{H}_{\mathcal{ML}}$.

2.2.3 The Φ Operator Properties

The following are the properties of the Φ operator.

Property 2.5**:** The Φ operator is not commutative.

Property 2.6**:** The Φ operator commutes with projection, provided the attributes used in Φ is preserved by the projection operator.

Property 2.7**:** The Φ operator commutes with selection, sort or join operators.

Property 2.8**:** The Φ operator commutes with aggregate operators, provided the aggregation preserves the semantic attribute.

Property 2.9**:** The definition implies that A Φ B is true, iff A is a descendant of B, in $\mathcal{H}$. Or, equivalently, A Φ B is true, iff B is an ancestor of A, in $\mathcal{H}$.

Property 2.5 follows directly from the asymmetry of the Φ operator. The properties 2.6 through 2.8 follow from the algebra of the standard relational operators. Property 2.9 follows directly from graph theory (as $\mathcal{H}$ is a set of DAGs) that guarantees the existence of a path from x to y, if the node y occurs in the closure of x in $\mathcal{H}$.

Property 2.5 reduces the plan search space by restricting flipping of operands of the Φ operator due to its asymmetry. The properties 2.6 through 2.8 provide means for enumerating different execution plans for queries using Φ operator. Property 2.9 suggests an alternative method for implementing Φ operator, which may be exploited depending on the structural characteristics of the hierarchy.

3 MURAL: **Multilingual Relational Algebra**

In this section we propose a domain-specific query algebra to match multilingual text data across languages. We first propose a new datatype Uniform and three new operators – Ξ, Ψ and Φ – that may be applied only on Uniform datatype, to extend the normal comparison semantics of the database systems.

3.1 Data Types

All the basic types of relational systems are preserved in Mural algebra, except Text datatype which is replaced by Uniform (UNIcode FORMat) datatype, specifically to hold the text strings.

3.1.1 Multilingual Text (Uniform) Datatype

The multilingual text data type is a 2-tuple, where the first is the text string in a standardized encoding (referred to as Text), and the second is an identifier for the language of the string[7] (referred to as LangID). The explicit identifier is necessary, as several languages share a script and a string may have different pronunciations or meanings, depending on its language[8].

Example 3.1: While <"*Sample String*", English>, <"*Une Corde Témoin*", French> and <"உவமான சரம்", Tamil> are of proposed Uniform datatype, their first components form normal Unicode strings. ◇

3.1.2 Decomposing and Composing Uniform Datatype

It is apparent that with Uniform datatype as above, standard database operations need to be redefined to work on the new datatype. We first introduce two simple operators on Uniform datatype – *Composing Operator* (denoted as, $\succ$) that can compose a Uniform datatype out of a given Unicode string and a language identifier and an inverse operator – *Decomposing Operator* (denoted as, $\prec$) that decomposes a Uniform data to a Unicode String and a language identifier. These two operators – $\succ$ and $\prec$ – may be implemented in a fairly straight forward manner, at little cost.

3.2 Uniform Matching (Ξ) Operator

A simple Homostr operator (Ξ) is defined as follows:

Ξ : *Set*$< U_1 >\times$*Set*$< U_2 >\rightarrow$*Set*$< U_1, U_2,$boolean$>$

This operator compares two Uniform datatypes; the result is the Cartesian product of the sets, with each tuple of the output tagged with a true or false. The match is tagged with a true if both the components of the 2-tuples match, or tagged with a false, otherwise.

Example 3.2: (<"*Jean*", English> Ξ <"*Jean*", English>) is true, and (<"*Gift*", English> Ξ <"*Gift*", German>) is false. ◇

[7] The text string is assumed to be in a single language, hence a unique language identifier is possible. Further, a unique value of Unknown is allowed, as a catch-all.

[8] Automatic language detection is possible with a large corpora, but not with attribute level datum.

3.2.1 Normal Text Operators on Uniform

All simple text comparison operations (specifically, $=, \neq, <, >$) applied to Uniform datatype, operate on the Text component of the decomposed Uniform. Specifically, the expression $a\ R\ b$, where R is one of the normal text operators applied on a pair of Uniform datatypes a and b is equivalent to $((\Pi_{a_{Text}}(\prec(a)))\ R\ (\Pi_{b_{Text}}(\prec(b))))$.

Example 3.3: The predicate (<"*Gift*", English> = <"*Gift*", German>) evaluates to true. ◇

It should be noted that while the $=, \neq, <, >$ operations are legal with Uniform strings, the results are meaningless when the strings are from different scripts; the equality always evaluates to false (as it should) and the sorting results depend on where the scripts corresponding to the languages, occur in the Unicode placement, which is largely arbitrary. However, we preserve such operational semantics, to be compatible with well-known Text datatype, which has similar behaviour.

3.3 Homophonic (Ψ) Operator

The homophonic Ψ operator is defined as follows:

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

The input is two sets of Uniform strings, and the output is the Cartesian product of the two sets, with each result tuple tagged with the *edit-distance* between their phonemic representations. This operation preserves both the input strings, which are available for subsequent operations. Removal of either of the input attributes is by projection operation, which 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 3.4**:** To select all Authors from table Authors (A) that are phonetically close to `Nehru` (threshold distance of 2), the query expression is as follows:

$\Pi_{A.Author}(\sigma_{dist<2}\ (\Psi_{Author}(A, \{\text{“}\texttt{Nehru}\text{”}\})))$ ◇

Example 3.5**:** To phonetically join two tables, Authors (A) and Books (B) (threshold distance of 2) and retrieve all phonetically close pairs of names, the query expression is as follows:

$\Pi_{A.Author, B.Author}(\sigma_{dist<2}(\Psi_{Author}(A, B)))$ ◇

3.4 Homosemic (Φ) Operator

The homosemic operator (Φ) is defined as follows:

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

The input is two sets of Uniform strings, and the output is the Cartesian product of the two sets, with each result tuple tagged with $match$, which is set to a boolean value, if $u_{1_i} \in \mathcal{T}_{\mathcal{H}}(u_{2_j})$, where $u_{1_i} \in U_1$ and $u_{2_j} \in U_2$. This operation preserves both the input strings, which are available for subsequent operations. Removal of either of the input attributes is by projection operation, which is left to the user.

Example 3.6**:** The query to retrieve all Books (B) that are categorized under `History`, may be retrieved as follows:

$\Pi_{B.BookID}(\sigma_{match \neq \phi}(\Phi_{Category}(B, \{\text{“}\texttt{History}\text{”}\})))$

◇

3.5 Composition of Operators

In this subsection, we provide an overview of the composition of the new operators, namely, Ξ, Ψ and Φ, with each other and with the traditional relational algebra operators, namely $\times, -$ and $\cup$ and the aggregation operator Δ, based on Properties 2.1 through 2.9. The highlights of the composition rules are given in Table 1.

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

Table 1: **Interaction between** Uniform **Operators**

Such composition rules are essential for the optimizer, to enumerate alternate query execution plans for a given query, by rearranging the operators. Each enumerated plan is costed, based on the cost models and selectivities (discussed in subsequent sections) of the operators, and the best plan in terms of overall cost or time for the first tuple (depending on the user setting) is chosen for execution.

3.6 Relational Completeness of MURAL

A formal system is said to be *Relationally Complete* if it is atleast as powerful as relational calculus (or equivalently, relational algebra). A test of relational completeness is that whether all queries expressible in relational algebra may be expressed in the proposed system. In this section we show that MURAL is relationally complete.

Lemma 3.1**:** There exists a mapping scheme Ω_{Sch} between a MURAL schema and a standard relational schema.

Proof: MURAL has all the datatypes of standard relational algebra, but for Text datatype, which is replaced by the Uniform datatype. Hence, for all schema objects, other than Text, Ω_{Sch} is identity. Ω_{Sch} between Uniform and text datatypes are defined as follows: Given an n-tuple relation R_m (= $\{r_1, r_2, ...r_n\}$) in MURAL specification and that r_i is of Uniform datatype, R_m can be mapped onto an equivalent relation R_r in standard relational algebra, where R_r is, $((R_m - r_i) \cup (\prec (r_i)))$ (namely, $\{r_1, r_2, ...r_{i-1}, r_{i_{Text}}, r_{i_{LangID}}, ..., r_n\}$). The resulting R_l is a relation composed of $(n+1)$-tuple of base datatypes. Similarly, an n-tuple relation S_r composed of base datatypes, may be converted into $(n-1)$-tuple relation S_m in MURAL algebra, by composing Uniform datatype from two *appropriate* base datatypes (from s_i Text datatype and s_j that stores a language identifier) as, $((S_r - s_i - s_j) \cup (\succ (s_i, s_j)))$. Thus, a relation in normal relational algebra may be transformed with no loss of information into a relation in MURAL, and *vice-versa*. ◇

Theorem 3.2 **(Relational Completeness Theorem):** There is a mapping scheme Ω that maps a relational algebra database D to a MURAL database $\Omega(D)$ such that, for every query Q on D, there is a corresponding expression $\hat{Q}$ such that $\hat{Q}(\Omega(D)) = \Omega(Q(D))$.

Proof: Note that for proving relational completeness, we need only to show only a mapping for all possible queries from *standard* database to the transformed database. Lemma 3.3 defines and ensures that a mapping exists between a MURAL schema and a schema in standard algebra. For normal datatype attributes and normal relational algebra operators, Ω is identity, trivially. We do not need to consider the new operators Ξ, Ψ, Φ that can be applied only on Uniform datatype, hence there is no need for defining Ω for this part. However, we need to show a Ω for normal Text manipulating operators applied on D has an equivalence in $\Omega(D)$. Suppose Q is an expression (in conjunctive normal form) in standard relational algebra (= $q_1 \wedge q_2 \wedge ... \wedge q_n$). Each q_i is a disjunction of the form $q_{i1} \vee q_{i2} \vee ... \vee q_{id_i}$, where q_{ij} is a predicate of the form $(a\ R\ b)$, where R is one of the standard operators on an standard attributes a and b. As discussed in Section 3.2.1, any such operation, depending on whether the text part or the ID part was used in Q, may be mapped to an expression $\hat{Q}$ as, $((\Pi_{a_{uni_{Text}}}(\prec (a_{uni})))\ R\ (\Pi_{b_{uni_{text}}}(\prec (b_{uni}))))$ or $((\Pi_{a_{uni_{ID}}}(\prec (a_{uni})))\ R\ (\Pi_{b_{uni_{ID}}}(\prec (b_{uni}))))$, where a_{uni} is $\succ (a_{uni_{Text}}, a_{uni_{ID}})$ and b_{uni} is $\succ (b_{uni_{Text}}, b_{uni_{ID}})$. Thus, the MURAL algebra is *relationally complete*. ◇

A practically significant outcome of the above result is that the existing systems, which are relationally complete, may be extended relatively easily to handle multilingual data. Only new multilingual datatype and operator functionalities need to be added.

3.7 Cost Models for Operators

In this section we discuss the MURAL operator cost models. There are two variations possible for each of the operators: **scan** type, which is of the form <*Attr*> Oper <*Const*>, and **join** type, which is of the form <*Attr*> Oper <*Attr*>. For the cost models, the notation defined in Table 2 are used and the costs of operations (in *big-O* notation) are given in Table 3.

Symbol	Represents
LHS (L) and RHS (R) Operands	
R_L, R_R	No. of Records in L, R
U_L, U_R	No. of Unique Values of L, R
l_L, l_R	Avg. length of Records in L, R
P_L, P_R	No. of Pages in L, R
E_L, E_R	No. of Pages for Exact Index in L, R
A_L, A_R	No. of Pages for Approximate Index in L, R
I_L, I_R	No. of keys per Index page
k	Ψ Error Tolerance (as a fraction in $(0, 1]$)
σ	Ψ Size of the Alphabet (= $\|\Sigma\|$)
R_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**

O	Remarks	Algorithm Complexity	Disk I/O
scan Operations			
Ξ	No Index	$R_L l_L$	P_L
Ξ	Index	$logE_L I_L l_L$	$logE_L$
Ψ	No Index	$R_L l_L k/\sqrt{\sigma}$	P_L
Ψ	Approx. Idx	$R_L l_L k^2/\sqrt{\sigma}$	A_L
Φ	No Index	R_L+$R_{\mathcal{H}}(h$+1)	$P_{\mathcal{H}}(h$+1)
Φ	Index on $\mathcal{H}$	U_L+$logE_{\mathcal{H}}(h$+1)	$logE_{\mathcal{H}}(h$+1)
join Operations			
Ξ	No Index	$R_L R_R l_L$	$3(P_L + P_R)$
Ξ	Index	$U_L logE_R I_R l_R$	$E_L + E_R$
Ψ	No Index	$U_L U_R l_L k/\sqrt{\sigma}$	$3(P_L + P_R)$
Ψ	Approx. Idx	$R_L R_R l_L k^2/\sqrt{\sigma}$	$A_L + A_R$
Φ	No Index	$R_L + R_R + U_R R_{\mathcal{H}}(h$+1)	$3(P_L$+$P_R)$+$P_{\mathcal{H}}$
Φ	Index on $\mathcal{H}$	$R_L + R_R + U_R logE_{\mathcal{H}}(h$+1)	$3(P_L$+$P_R)$+$E_{\mathcal{H}}$

Table 3: **Cost Models for Operators**

For all the **join** predicates, we assume only one invocation of Ψ or Φ operator is made for duplicate values. All *edit-distance* computations are assumed to be implemented using *diagonal transition* [28] algorithm instead of the standard *dynamic-programming* algorithm, by virtue of its better complexity. For Ψ operation costs with indexes, the indexes are assumed to be created on the materialized phonemes strings.

3.8 Estimations of Operator Output

This section outlines heuristics used to estimate the output size of operators.

Estimation of Size of Homostr **Output**
The output size of the Ξ operator is assumed to be similar to that of normal $=$ operator.

Estimation of Size of Homophone **Output**
We estimated the output of the Ψ operator, based on equivalent *q-grams* of the given data set, as follows: Using the notation given in Table 2, the number of records in the q-gram table is given by, $R_L(l_L + 2q - 2)$. Given a query string s_{query} with $|s_{query}|+2q-2$ q-grams, the selectivity may be estimated as $|s_{query}|+2q-2$ / $R_L(l_L+2q-2)$, for a specific match. For small q, the above expression may be approximated to $|s_{query}|/R_L l_L$ for the selectivity of **scan** operations and $|s_{query}|R_R l_R/R_L l_L$ for **join** operations.

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

4 Implementation in PostgreSQL System

In this section we explore adding 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. We subsequently analyse the performance of such implementation and compare it with a quick outside-the-server implementation using UDFs. Finally, we demonstrate the power of Mural algebra in selecting efficient query execution plans.

4.1 System Setup for Core Implementation

The core implementation of the functionality was done on the PostgreSQL open-source database system [7] (Version 7.4.3), on RedHat Linux (Version 2.4) operating system. The implementation was tested on a stand-alone standard Pentium IV workstation (2.3GHz) with 1 GB main Memory. The operator implementation took approximately 6 person-months and was implemented in C. In addition to the operators, we implemented a specialized index structure for metric spaces, using GiST features available in PostgreSQL system. An open-source text-to-phoneme engine – Dhvani [2], was integrated with the system, after appropriate modification to output the phonemic strings in IPA alphabet and to make it a callable routine from the query processing engine.

4.2 Ψ Operator Implementation

The Ψ operator was implemented as a binary join operator, using the facility provided by the PostgreSQL system to define new operators. However, since there is no facility to add a tertiary operator, we implemented Ψ as a binary operator, and made the third input, the error threshold parameter, a user-settable parameter to be set 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 application. The homophonic matching function in Figure 4 is modified slightly to take the two strings as operator input, and the threshold from the system table, and implemented in the system. A modified *Dhvani* text-to-phoneme converter was used to convert the multilingual strings to their phonemic representations. 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 during a join query processing). The selectivity of the fuzzy Ψ operator was set by modifying the equi-join selectivity to take care of the fact that a given string may match with multiple unique target strings. The cost of the operator was set to the formulae given in Table 3, and the selectivity estimate using the methodology outlined in Section 3.8.

4.2.1 Specialized Index Structures

As the normal B+Tree index cannot be used in accessing *near* phoneme strings, we chose a metric index structure – M-Tree – to index the materialized phoneme strings, for

speeding up the Ψ operator. The M-Tree index is a height-balanced tree and is appropriate for dynamic data environments, such as database systems. We chose the random-split alternative [16] for splitting nodes when expanding the tree, since it offers best index modification time and has insignificant incremental disk I/O compared to other alternatives that are more computationally intensive.

The M-Tree index was added to the PostgreSQL database system using the GiST indexing feature [21] available in the system. GiST (Generalized Search Tree) defines a framework for managing a balanced index structure that can be extended to support new datatypes and new queries that is natural to the datatypes. A more efficient Slim Tree [34] could not be considered, as the necessary explicit insertion of specific elements on designated nodes of the index tree, is not supported in the PostgreSQL's GiST implementation.

4.3 Φ Operator Implementation

The Φ operator was also added to PostgreSQL system as a binary join operator, using the operator addition facility in the system. The homosemic matching functionality, as given in Figure 6 was implemented in C. Due to the high cost involved in computing closures on WordNet taxonomical hierarchies from the database tables, we pinned the tables in the main memory, for efficient traversal. Further, every time a closure for a RHS attribute value is computed, it is materialized as a hash table in temporary tables in the main memory, for fast execution of second step of Φ algorithm in checking set-membership of LHS attribute, as well as for possible reuse. 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 RHS operand value may amortize the cost of computing and materializing the closures. For example, a scan or nested-loops join queries using 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. While we implemented these optimizations, we added a simple cost models as given in Table 3 and selectivities as given in Section 3.8, for use in the optimizer.

4.4 Optimizer Prediction Performance

In order to ascertain the quality of our cost models and the accuracy of the optimizer in predicting the query costs, a a series of queries using our multilingual operators on a suite of tables with varying data characteristics, were run on the system. A series of tables of varying characteristics (in terms of attribute size, tuple count, number of database blocks and selectivity) were created, and a suite of queries that used the multilingual operators were run on these tables. For each query, we recorded the optimizer predicted cost and the actual runtime of the query. It should be noted that the optimizer prediction cost is specified in terms of units of disk-page fetch in PostgreSQL system.

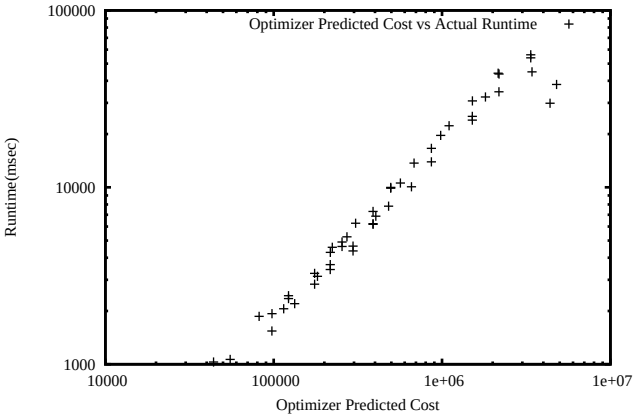

Figure 7: **Optimizer Prediction Performance**

Figure 7 plots the 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. Though there is some error in computing large queries, we observed that this error is in the same range as in the case of estimation with conventional operators.

5 Performance Experiments

In this section, we outline our performance experiments with our implementation of multilingual operators on PostgreSQL database system. We first present a baseline performance using a quick outside-the-server implementation, and demonstrate the power of core implementation and the optimization strategies.

5.1 Data Setup for Experiments

We used the same datasets that were used in our previous work, in benchmarking the homophonic and homosemic functionalities on commercial systems [26, 27]. For Ψ operator, a pre-tagged $\approx 200,000$ multilingual names data set was used for scan experiments, and a fraction of the above table used for join experiments. For homosemic experiments the entire WordNet hierarchy with $\approx 110,000$ word forms $\approx 80,000$ synsets and $\approx 140,000$ relationships between them was stored in the database, occupying about 4 MB of storage. Since different WordNets 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. Queries that compute closures of varying sizes were employed for profiling the Φ operator. All experiments were run on these dataset on a standard workstation, quiesced of all other activities[9].

[9]The quality of the results of the experiments are discussed in detail in [26, 27], where similar experiments were conducted on a commercial database system. We verifi ed that the results are identical in both the cases.

5.1.1 Metric Distance Index

While the B+ Tree indexes are largely ineffective in the edit-distance measure based metric space, a metric distance index may be implemented using standard B+Tree index in database systems. If the edit-distances of all strings in the database are known, from a given string, then they may be used for pruning the search space, based on the following lemmas that follow from the definition of *triangular inequalities* that define metric distances[10].

Lemma 5.1: Given two strings a and b at a distance of d_{ab} from each other, and a query to return strings within a distance of d_a and d_b from a and b respectively, there could be no satisfying strings, if $d_a + d_b < d_{ab}$.

Lemma 5.2: Given two strings a and b at a distance of d_{ab} from each other, a query to return strings within a distance of d_a and d_b from a and b respectively, and a candidate string s at a distance of d_{sa} ($< d_a$) from a, it may be in the result set if and only if $d_{sa} + d_{ab} \leq d_b$.

Example 5.1**:** Consider a query to find the Authors, phonetically close to `Silversmith` (threshold of 2) and `Aerosmith` (threshold of 2). This query could return no result set as per Lemma 5.2, since the distance between `Silversmith` and `Aerosmith` is 5. $\diamond$

The lemmas 5.1 and 5.2 are useful in designing the operator implementation, as they provide means of reducing the edit distance computation, based on pre-computed distances from a known string. We pursue such a strategy as follows: We chose a candidate string, *key string*, and computed the edit-distance of each of the values of the attribute from S_k to be stored along with attribute. We observed that a judicious choice of the *key string*, with a length equal to the average length of the values and with an alphabet distribution similar to that of the collection, reduces the computation significantly. A B+tree index was built on the pair, <*distance*,*string*>, called the *Metric Distance Index* (M). Given M, a **scan** type operation to retrieve the strings that are at an edit-distance less than d_q from the query string, S_q, is computed as follows: First, compute distance d_{kq} of S_q from S_k. Second, access the index M, and output all those strings with distance d_k, such that $d_{kq} + d_k \leq d_q$. The correctness of this step is guaranteed by Lemma 5.1, and it requires no explicit distance computation, but just an access of the index structure. Third, for those strings with distances d_k, such that $d_{kq} + d_s \geq d_q$, compute edit distance to verify if it is at an edit-distance $\leq d_q$.

5.1.2 Baseline Performance of Ψ Implementations

To baseline the performance of the outside-the-server implementation of the Ψ operator, we first added it to the PostgreSQL open-source database systems using user-defined function, defined in PL/SQL programming environment. The reason for the choice was to have parity with other systems that allow only such environments for adding UDFs.

[10]A similar metric index may be used for weighted edit-distances as well, though they are not discussed here due to lack of space.

Query Type	**Scan-type** (*Sec.*)	**Join-type** (*Sec.*)
Base Performance	3618	453
With Metric Index	362.9	166.9

Table 4: ***Outside-the-Server*** **Performance of Ψ Operator**

Table 4 lists the performance of the Ψ operator, without and with appropriate indexes on the materialized phonemic strings. The results show clearly that while the *no-index* implementation is expensive, the metric index structure helped in reducing the cost by nearly an order of magnitude. The main impediment to the performance is the expensive UDF invocations.

5.2 Baseline Performance of Φ Operator

Similarly, the basic Φ operator was implemented as a user defined function using PL/SQL features, and a baseline performance is measured running the operator on the dataset based on the WordNet taxonomical hierarchy. The first step of the Φ operator is the most expensive one, as closure computation on relational tables are recognized to be expensive[13, 20, 22]. The cost models and the query plans of the database systems indicate that nearly 98% of the query time was spent on the first step. We present here the time to compute a suite of queries, each with varying sizes of transitive closures. Table 5 shows the performance of our basic outside-the-server implementation of Φ operator.

Closure Cardinality	*Outside-the-Server* (without Index) (*Sec.*)	*Outside-the-Server* (with Index) (*Sec.*)
155	5.676	0.028
482	17.72	0.094
2041	75.62	0.642
2538	95.77	0.907
5340	201.8	2.778
11551	840.5	11.64

Table 5: ***Outside-the-Server*** **Performance of Φ Operator**

The basic *no-index* implementation of the Φ operator is expensive, taking upto few hundred seconds for typical queries. A basic index on the hierarchy table $\mathcal{H}_{\mathcal{ML}}$ speeds up the closure computation significantly, improving it by nearly two orders of magnitude, though there is a significant increase in the costs, when large closures are computed.

5.3 Core Performance of Ψ Operators

After implementing the Ψ operator in core as indicated in the previous sections, we ran the same experiments that were used for testing the performance of the *outside-the-server* implementation. For homophonic experiments, the GiST index (implementing the M-Tree) was also used for enhancing the performance. We provide, in Table 6,

the performance of queries scanning and joining the same $200,000$ row table that was used in outside-the-server experiments for the Ψ operator.

Query Type	**Scan-type** (*Sec.*)	**Join-type** (*Sec.*)
Base Performance	5.203	1.967

Table 6: ***Core* Performance of Ψ Operator**

As can be seen, the performance of the queries is two orders of magnitude faster than the outside-the-server performance of the same queries shown in Table 4. We measured the performance gain due to the GiST index, and Figure 8 highlights the relative performance gain due to the GiST index. We note that the performance of the Ψ operator is significantly speeded up by the GiST index, upto nearly 90%.

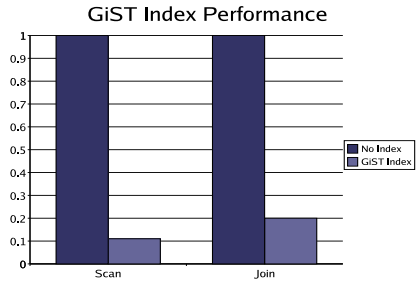

Figure 8: **Performance of GiST Index**

5.4 Core Performance of Φ Operators

The implementation of the Φ operator as a core operator was tested for queries that require closures of various sizes in WordNet taxonomic hierarchy, and the results are shown in Table 7.

Closure Cardinality	*Core* (without Index) (*Sec.*)	*Core* (with Index) (*Sec.*)
155	1.378	0.005
482	4.303	0.014
2041	19.68	0.037
2538	22.70	0.042
5340	44.86	0.044
11551	96.71	0.068

Table 7: ***Core* Performance of Φ Operator**

Compared with the outside-the-server performance of Φ operator (as shown in Table 5), we note that the performance of the core implementation is about one order of magnitude better, when computed without building an index on the hierarchy. With index, the performance is improved by at least two orders of magnitude, to a few tens of milliseconds. The performance of our PostgreSQL implementation with index structures is sufficient for practical deployments, given that the typical size of closure is around $2,000$ [27].

5.5 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 4.1**:** Consider a query *Find the books whose Author name sounds like that of the publisher (threshold of* 0.25*).* Assuming the tables Author (A) with AuthorID and Author Name, Books (B) with BookID and foreign key to its author and publisher, and Publisher (P) with PublisherID and Publisher Name, the following two expressions (also, shown pictorially in Figure 9) capture the semantics of the above query:

Plan 1: $\Pi_{A.AuthorID,P.PubID,B.BookID}$
$(\sigma_{(Threshold \leq 2.5)}(\Psi_{A.AName,P.PName}(P, A)$
$(B \bowtie_{BookID} (A \bowtie B))))$

Plan 2: $\Pi_{A.AuthorID,P.PubID,B.BookID}(\bowtie_{BookID} (A, B)$
$(\sigma_{(Threshold \leq 2.5)}(\Psi_{A.AName,P.PName}(P, A))))$

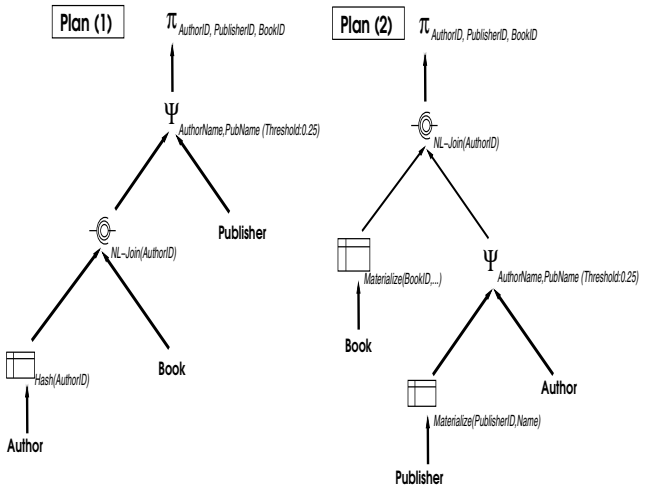

Figure 9: **Query Plan for Example** 4.1

We created 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 same query, on the same tables, 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 table, confirming the use of our cost models and optimization strategies by the optimizer. $\diamond$

6 Migration Strategies

Database systems have well established software architecture, query semantics and interface specifications. Any new features addition must consider the impact of the proposal on each of the above parameters. The new functionality addition may be classified, based on the level of integration with the system, as either *Outside-the-Server* or *Core* Implementation. While we have detailed our core implementation of the functionalities on PostgreSQL in the previous sections, in this section, we outline how these functionalities may be added to other database systems with existing database or SQL features. Clearly, such an implementation will suffer from significant overheads and the fact that they cannot leverage on the well-developed relational query optimizer.

6.1 The Homophonic Ψ Operator

In this section, we outline how the existing UDF features of the database may be leveraged for an outside-the-server implementation of Ψ operator. In the outside-the-server approach, a UDF that implements the algorithm given in Figure 4 is defined, in an environment that is native to the database system. Examples for such environments are the User-defined Function facility in IBM DB2 Universal Server, Oracle Database Server and Microsoft SQL Server. In PostgreSQL, we used the PL/SQL programming environment for defining the UDF, in order to preserve the consistency between equivalent facilities in different database systems, though a more efficient PL/C feature could have been used. Note that the interpreted nature of the environment may add additional processing overheads to the query execution. In addition, the standard B+Tree index may be used to define a metric distance index, as detailed in this paper.

6.2 The Homosemic Φ Operator

In this section, we explore the implementations of the Φ operator using existing SQL features available in database management systems. For the Φ operator the taxonomic hierarchy $\mathcal{H}_{\mathcal{ML}}$ needed for computing the semantic closures may not be pinned on to the main memory, and would need to be stored in relational tables. Given the relational storage of $\mathcal{H}_{\mathcal{ML}}$, the Φ operator may be implemented in two steps: First, the computation of the transitive closure of the RHS in the hierarchy, namely, $\mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(\text{RHS})$. Second, testing for the condition $\{\text{LHS} \cap \mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(\text{RHS})\} = \phi$. While precomputation of closures help in reducing the cost of the first step of this operator, it comes at a heavy storage penalty. Further, the availability of index structures produces different benefits for the Φ operator, depending on which attribute they are available on. Availability of index structures on $\mathcal{H}_{\mathcal{ML}}$ table is very beneficial since it speeds up the closure computation, which is the most expensive part of the Φ operator. However index on operand tables, at best, help only in eliminating the duplicate closure computation, be leveraging on their sort order, hence is not very significant.

6.2.1 Database Features for Ψ Implementation

For an basic *outside-the-server* implementing of the Φ operator, we relied on the standard SQL:1999 features. The first was implemented by a recursive SQL feature, namely the WITH clause of SQL:1999 standards and the second part by the IN predicate of standard SQL. The WITH clause is supported as it is by IBM DB2 Universal Server and by CONNECT BY clause in Oracle $9i$ database. In Microsoft SQL Server, the functionality may be implemented using scripts. In the open source PostgreSQL server, since recursive SQL is not available, we implemented the closure computation using a user-defined function in PL/SQL. All the servers support the IN clause needed for the second step of the Φ implementation, by building efficient hash-tables.

7 Conclusion

In this paper, we first highlighted the need for seamless processing of multilingual text data, with motivating example from real-life domain. We presented formally the definitions of specific multilignual operators and analysed their properties that are used to define the composition rules among them. We presented their cost models and selectivity estimates, the critical input to the relational query optimizer. Subsequently, we presented a query algebra – Mural that defines a multilingual datatype and operators, for intuitively expressing complex queries. We showed that Mural is relationally complete, and hence could be added to existing relational systems at little cost.

We outlined a *core* implementation of the functionality as first-class operators inside the database kernel. In order to optimize the performance of homophonic operator, we added a metric M-Tree index using GiST index features supported on PostgreSQL database system. We first baselined the outside-the-server performance of PostgreSQL database system, and demonstrated that the core implementation improves the performance by two orders of magnitude over the outside-the-server implementation. Further, we showed that the indexes improves the performance by another one to two orders of magnitude. Also, we demonstrated the power of the query algebra in aiding optimizer to selecte efficient execution plans. Finally, as a migration strategy, we outlined how the existing features of current database systems may be used to implement the multilingual features. Thus, our proposal of MURAL multilingual operator algebra and its *core* implementation on PostgreSQL database system represents the first step towards the ultimate objective of achieving complete multilingual functionality in database systems.

7.1 Related Research

To the best of our knowledge, ours is the first proposal that covered the entire spectrum of functionality proposal to the core implementation of multilingual phonetic and semantic matching in a database kernel.

While we had published multilingual query processing [26, 27] features earlier, no holistic approach had been taken earlier for defining and using a query algebra, which we address here. There are vast amounts of literature in the Information Retrieval [8] 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 OLTP type environments. Phonetic matching of English strings was discussed in [35, 36], but their focus had been on the linguistic issues and quality of the resulting match. Also, their main-memory implementation did not raise issues related to the on-disk database processing. Recently, a major database vendor [17] proposed and demonstrated matching functionality using ontologies. While their implementation is not available in the released versions of the software, we expect that our multilingual operators to easily leverage on such implementations, when available. Further, we are heartened by this parallel effort in ontological query processing that validates our approach. Algebras specific to a domain are available, such as PiQA[33] in Bioinformatics and TAX[23] in XML. Our approach parallels such efforts, in multilingual text domain. We also leverage on research on approximate string matching [15, 28, 29, 30], for our implementation.

References

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

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

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

[4] Global Reach. *http://www.globalreach.biz*.

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

[6] International Phonetic Association. *http://www.arts.gla.ac.uk/IPA/ipa.html*.

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

[8] ACM SIGIR. *www.acm.org/sigir*.

[9] The Unicode Consortium. *http://www.unicode.org*.

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

[11] The WordNet. http://www.cogsci.princeton.edu/w̃n.

[12] R. Agrawal et al. Direct algorithms for computing Transitive Closure of DB Relations. *Proc. of 13th VLDB Conf.*, 1987.

[13] R. Agrawal, S. Dar and H. V. Jagadish. Direct Transitive Closure Algorithms: Design and Performance Evaluation. *ACM Trans. on Database Systems*, 1990.

[14] R. Baeza-Yates and G. Navarro. Faster Approximate String Matching. *Algorithmica*, Vol 23(2):127-158, 1999.

[15] E. Chavez, G. Navarro, R. Baeza-Yates and J. Marroquin. Searching in Metric Space. *ACM Computing Surveys*, Vol 33(3):273-321, 2001.

[16] 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.

[17] S. Das *et. al.* Supporting Ontology-based Semantic Matching in RDBMS. *Proc. of 30th VLDB Conf.*, 2004.

[18] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan and D. Srivastava. Approximate String Joins in a Database (almost) for Free. *Proc. of 27th VLDB Conf.*, 2001.

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

[20] J. Han et al. Some Performance Results on Recursive Query Processing in Relational Database Systems. *Proc. of 2nd IEEE ICDE Conf.*, 1986.

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

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

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

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

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

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

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

[28] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, Vol 33(1):31-88, 2001.

[29] G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio. Indexing Text with Approximate q-grams. *Proc. of 11th Combinatorial Pattern Matching Conf.*, June 2000.

[30] G. Navarro, R. Baeza-Yates, E. Sutinen and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, Vol 24(4):19-27, 2001.

[31] P. G. Selinger *et. al.* Access Path Selection in a Relational Database Management System. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1979.

[32] P. H. Sellers. On the Theory and Computation of Evolutionary Distances. *SIAM Jour. of Applied Math.*, June 1974.

[33] S. Tata and J. M. Patel. PiQA: An Algebra for Querying Protein Data Sets. *Conf. on Scientific and Statistical Data Management*, July 2003.

[34] 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.*, March 2000.

[35] J. Zobel and P. Dart. Finding Approximate Matches in Large Lexicons. *Software – Practice and Experience*, Vol 25(3):331-345, March, 1995.

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