MLPostgres: Implementing Multilingual functionalities inside PostgreSQL Database Engine

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

P. Pavan Kumar Chowdary

Computer Science and Automation
Indian Institute of Science, Bangalore
Bangalore – 560 012 (INDIA)

June 2005

Dedicated

To

My beloved
Parents and Sister

Acknowledgments

I am grateful to my project guide, Dr, Jayant Haritsa for his excellent guidance throughout the course of this project. I take this opportunity to express my deepest gratitude to him for the moral support that he gave me and for providing me a vast pool of literature and computing resources.

I would like to thank Kumaran, who has constantly helped me throughout the project, by giving me his valuable suggestions.

I am also indebted to all my DSL labmates Naveen, Priti, Srikanta, Kumaran, Maya, Suresha for providing a lively and pleasant company. My friends made my stay at IISc quite memorable. Friends at CSA and IISc are many in number. There are several people in CSA department who deserve acknowledgment for their help in successful completion of this work. I thank all the faculty and staff member of department for their support and help.

I can not express my gratitude in words to my parents and sister for their unconditional support and encouragement.

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, two multilingual functionalities: LexEQUAL[15] and SemEQUAL[16] were recently proposed in the literature. LexEQUAL supports phoneme based matching of names and SemEQUAL supports ontology based matching of concepts.

In this report, we investigate the implementation of these multilingual functionalities as first-class operators on relational engines, using the PostgreSQL open-source database system. We first present the functionality of the above mentioned operators and their algebraic properties, selectivity estimations and cost models. Subsequently we discuss a staged implementation roadmap from *outside-the-server* to *inside-the-server* to *core* implementation of LexEQUAL and SemEQUAL in PostgreSQL. To speedup LexEQUAL processing, a metric index has been incorporated using the GiST feature of PostgreSQL. We have taken *outside-the-server* implementation using existing features of the database system as the baseline for performance measure. Our experiments over representative multilingual datasets demonstrate orders-of-magnitude performance gains for the *core* and *inside-the-server*implementation over *outside-the-server*. We also demonstrate the power of our *core* implementation in selecting efficient execution plans, which is not possible with *inside-the-server* implementation. 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.

Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

1.1 Need for multilingual data 1

1.2 Background 2

1.2.1 Multilingual Name Searches 3

1.2.2 Multilingual Concept Searches 4

1.2.3 Contribution of this Project 5

1.2.4 Organization of the Thesis 6

2 Multilingual Functionalities **7**

2.1 LexEQUAL Functionality Definition 7

2.1.1 LexEQUAL(Ψ) Operator Algorithm 8

2.1.2 LexEQUAL(Ψ) Operator Properties 10

2.2 SemEQUAL Functionality Definition 10

2.2.1 WordNet-based Taxonomic Hierarchies 11

2.2.2 The SemEQUAL(Φ) Operator Algorithm 12

2.2.3 The SemEQUAL(Φ) Operator Properties 12

2.3 Cost Models for Operators 13

2.4 Estimations of Operator Output 13

2.4.1 Estimation of Cardinality of LexEQUAL Output 15

2.4.2 Estimation of Cardinality of SemEQUAL Output 15

3 PostgreSQL **16**

3.1 PostgreSQL Query flow 16

3.2 GiST Indexes 18

3.2.1 Extensibility 19

3.2.2 Implementation 19

3.2.3 Limitations 20

4 M-Tree **21**

4.1 Metric space 21

4.2 M-Tree index 22

5 Implementation in PostgreSQL **24**
5.1 Implementation choices 24
5.1.1 System Setup for Implementation 26
5.2 Outside-the-Server Implementation 26
5.3 Inside-the-Sever Implementation 27
5.3.1 Ψ Operator Implementation 28
5.3.2 Φ Operator Implementation 28
5.4 Core Implementation 29
5.4.1 Ψ Operator Implementation 29
5.4.2 Φ Operator Implementation 30
5.4.3 Optimizer Prediction Performance 31

6 Performance Analysis **32**
6.1 Data Setup for Experiments 32
6.2 Baseline Performance of Ψ Operator 34
6.3 Baseline Performance of Φ Operator 34
6.4 Core (or Inside-the-server) Performance of Ψ Operator 35
6.5 Core (or Inside-the-server) Performance of Φ Operator 36
6.6 A Motivating Optimization Example 37

7 Conclusions **39**

8 Future Research **41**

Bibliography **41**

List of Tables

2.1 **Symbols used in Analysis** 14
2.2 **Cost Models for Operators** 14

6.1 ***Outside-the-Server* Performance of Ψ Operator** 34
6.2 ***Outside-the-Server* Performance of Φ Operator** 35
6.3 ***Core* Performance of Ψ Operator** 35
6.4 ***Core* Performance of Φ Operator** 36

List of Figures

1.1 **Multilingual *Books.com*** . . . 2
1.2 SQL:1999 **Multiscript Query** . . . 3
1.3 LexEQUAL **Query Syntax** . . . 4
1.4 **Results of** LexEQUAL **Query** . . . 4
1.5 SQL:1999 **Multilingual Semantic Matching Query** . . . 4
1.6 SemEQUAL **Query Syntax** . . . 5
1.7 **Results of** SemEQUAL **Query** . . . 5

2.1 **The LexEQUAL Algorithm** . . . 8
2.2 **Sample Interlinked WordNet Hierarchy** . . . 11
2.3 **Semantic Matching Algorithm** . . . 12

3.1 **PostgreSQL Query flow diagram** . . . 17

5.1 **Different implementation choices to add new functionality to PostgreSQL database engine** . . . 25
5.2 **SQL statement to create PL/pgSQL function** . . . 26
5.3 **SQL statement to create C-Language Function** . . . 27
5.4 **SQL statement to create operator** . . . 27
5.5 **Code to make internal function** . . . 29
5.6 **Code to make first class operator** . . . 30
5.7 **Optimizer Prediction Performance** . . . 31

6.1 **Phonemic Representation of Test Data** . . . 33
6.2 **Distribution of Generated Data Set (for Performance Experiments)** . . . 33
6.3 **Performance of M-Tree Index** . . . 36
6.4 **Query Plans for Example** 6.1 . . . 37

Chapter 1

Introduction

1.1 Need for multilingual data

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. For example, it is not possible to automatically match the English string *Al-Qaeda* and its equivalent strings in other scripts, say, Arabic, Greek or Chinese, or the English word *attack* with its equivalent word in other languages, even though such a feature could be immensely useful for news organizations or security agencies. To address this lacuna, two new multilingual functionalities – LexEQUAL [15] and SemEQUAL [16] were recently proposed in literature. LexEQUAL supports phoneme-based matching of names while SemEQUAL supports ontology based matching of concepts.

The above papers focussed primarily on the *outside-the-server* implementation of the multilingual functionalities. In this report, we investigate their implementation as first-class operators on relational engines, using PostgreSQL open-source database system. We present the operator cost models, selectivity estimates and composition rules. To push the operators into the core

[1]Internet user population is growing at a rate of 12 to 15% yearly[32].

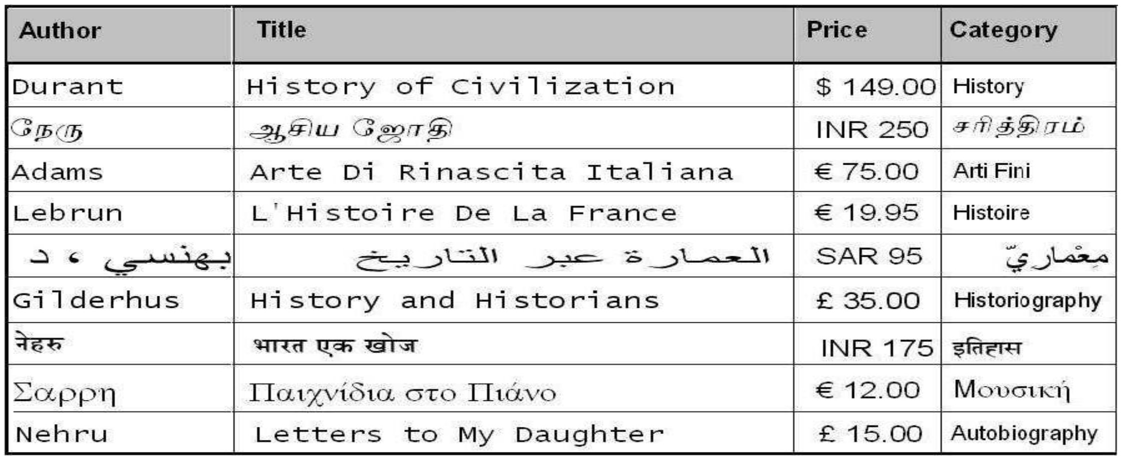
[2]Two-thirds of the current Internet users are non-native English speakers [28] and it is predicted that the majority of web-published data will be multilingual by 2010 [39].

engine, these are critical inputs for relational query optimizer, for the selection of efficient query execution plans.

Subsequently we discuss a staged implementation roadmap from *outside-the-server* to *inside-the-server* to *core* implementation of LexEQUAL and SemEQUAL in PostgreSQL. To speedup LexEQUAL processing, a metric index has been incorporated using the GiST feature of PostgreSQL. We have taken *outside-the-server* implementation using existing features of the database system as the baseline for performance measure. Our experiments over representative multilingual datasets demonstrate orders-of-magnitude performance gains for the *core* and *inside-the-server* implementations over *outside-the-server*. We also demonstrate the power of our core implementation in selecting efficient execution plans, which is not possible with inside-the-server implementation. 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.

1.2 Background

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 1.1. The product catalog shown may be considered as a logical view assembled from data from several databases (each aligned with the local language needs), but searchable in a unified manner for multilingual users.



Author	Title	Price	Category
Durant	History of Civilization	$ 149.00	History
நேரு	ஆசிய ஜோதி	INR 250	சரித்திரம்
Adams	Arte Di Rinascita Italiana	€ 75.00	Arti Fini
Lebrun	L'Histoire De La France	€ 19.95	Histoire
بهنسي ، د	العمارة عبر التاريخ	SAR 95	مِعْماريّ
Gilderhus	History and Historians	£ 35.00	Historiography
नेहरु	भारत एक खोज	INR 175	इतिहास
Σαρρη	Παιχνίδια στο Πιάνο	€ 12.00	Μουσική
Nehru	Letters to My Daughter	£ 15.00	Autobiography

Figure 1.1: **Multilingual *Books.com***

1.2.1 Multilingual Name Searches

In this environment, an SQL:1999 compliant query to retrieve all works of an author (say, `Nehru`), across multiple languages (say, in `English`, `Hindi`, `Tamil` and `Greek`) would have to be written as shown in Figure 1.2.

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

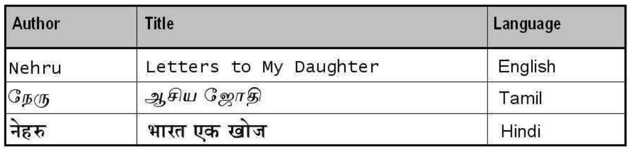
Figure 1.2: SQL:1999 **Multiscript Query**

This query suffers from a variety of limitations: Firstly, the users have to specify the search string `Nehru` in all the languages in which they are interested. This not only entails the users to have access to lexical resources, such as fonts and multilingual editors, in each of these languages to input the query, but also requires the users to be proficient enough in all these languages, to provide all close variations of the query name. Secondly, given that the storage and querying of proper names is significantly error-prone due to lack of dictionary support during data entry even in monolingual environments [13], the problem is expected to be much worse for multilingual environments. Thirdly, and very importantly, it would not permit a user to retrieve all the works of `Nehru`, *irrespective* of the language of publication. Finally, while selection queries involving multiscript *constants* are supported, queries involving multiscript *variables*, as for example, in join queries, cannot be expressed.

The LexEQUAL operator attempts to address the above limitations through the specification shown in Figure 1.3, where the user has to input the name in only one language, and then either explicitly specify only the *identities* of the target match languages, or even use `*` to signify a wildcard covering *all languages* (the `Threshold` parameter in the query helps the user *fine-tune* the quality of the matched output, as discussed later in the paper). When this LexEQUAL query is executed on the database of Figure 1.1, the result is as shown in Figure 1.4.

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

Figure 1.3: LexEQUAL **Query Syntax**



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

Figure 1.4: **Results of** LexEQUAL **Query**

1.2.2 Multilingual Concept Searches

Consider the query to retrieve all History books in Books.com, in a set of languages of users choice. An SQL:1999 compliant query to retrieve all books on history, across multiple languages (say, in English, Hindi, Tamil, Arabic) would have to written as shown in Figure 1.5. This SQL:1999 query also suffers from all the limitations which are given for writing Multilingual Name Searches in section 1.2.1. Further the current SQL:1999 compliant query, having the selection condition as Category=”History” would return only those books that have Category as History books in all the languages (or in a set of languages specified by the user) are returned.

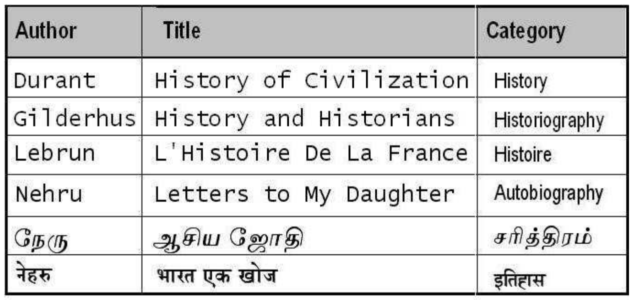
```
select Author,Title,Category from
Books where Category = ’History’
 or Category = ’इतिहास’
  or Category = ’சரித்திரம்’
   or Category = ’مِعْمارِيّ’
```

Figure 1.5: SQL:1999 **Multilingual Semantic Matching Query**

The SemEQUAL operator [16] attempts to address the above limitations through the specification shown in Figure 1.6, where the user has to input the name in only one language, and then either explicitly specify only the identities of the target match languages, or even use * to signify a wild card covering all languages as in LexEQUAL query. When this SemEQUAL query is executed on the database of Figure 1.1, the result is as shown in Figure 1.7. The output contains all books that have their category values that are semantically equivalent to History, the categories that are subsumed by History [3] are also retrieved, due to the ALL clause in the query.

```
select Author,Title,Category from
Books where Category SemEQUAL ALL
    'History' inlanguages
{English,Hindi,Tamil,Arabic}
```

Figure 1.6: SemEQUAL **Query Syntax**



Author	Title	Category
Durant	History of Civilization	History
Gilderhus	History and Historians	Historiography
Lebrun	L'Histoire De La France	Histoire
Nehru	Letters to My Daughter	Autobiography
நேரு	ஆசிய ஜோதி	சரித்திரம்
नेहरु	भारत एक खोज	इतिहास

Figure 1.7: **Results of** SemEQUAL **Query**

In [16], authors refer to such matching as **Multilingual Semantic Matching**, which matches multilingual text strings based on their meanings, irrespective of the languages of storage.

It should be specially noted here that this solution methodology is also applicable for extending the standard matching semantics of mono-lingual strings. For example, the **LexEQUAL** operator may be used for matching the English name Catherine and all its variations, such as Kathrin and Katerina. Similarly, the **SemEQUAL** operator, may be used for matching Disk Drive with Data Storage Devices and Computer Peripherals.

1.2.3 Contribution of this Project

With this project, our contributions are as follows:

- **M-Tree index implementation in PostgreSQL** to improve performance of LexEQUAL operator.

[3] Historiography(the study of history writing and written histories) and Autobiography (writing ones own life history) are considered as specialized branches of History itself.

- **Core implementation** of the LexEQUAL and SemEQUAL operators, and demonstration of improvement in performance to the tune of 2 to 3 orders of magnitude.

1.2.4 Organization of the Thesis

The rest of this Thesis is organized as follows: Chapter 2 defines and analyses the LexEQUAL and SemEQUAL. A brief introduction to PostgreSQL is presented in Chapter 3. A brief introduction to metric space and M-Tree index is presented in Chapter 4. Implementation choices and details of implementation of the functionality inside PostgreSQL is discussed in Chapter 5. The performance of different implementations of multilingual functionality are outlined in Chapter 6. Finally, we summarize our conclusions and outline future research avenues in Chapter 7.

Chapter 2

Multilingual Functionalities

In this chapter, we present the phonetic and semantic matching functionalities to match *multilingual text attributes*, formalizing the functionality of LexEQUAL and SemEQUAL given in [15, 16].

2.1 LexEQUAL 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 $\mathcal{S_I}$ be set of all such s_i. Let $\mathcal{S} = \cup_{\mathcal{I}} \mathcal{S_I}$, for a given set of languages. We also assume that the phoneme strings are encoded in the *International Phonetic Association* (*IPA*) [34] 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, $\mathcal{T_I}$, between a given language string s_i and a corresponding phonemic string p_i, is represented by $\mathcal{T_I} : \mathcal{S_I} \rightarrow \mathcal{S_{IPA}}$. The union of such transformation functions $\mathcal{T}$ $(= \cup_i \mathcal{T_I})$ in a set of desired languages, represented by $\mathcal{T} : \mathcal{S} \rightarrow \mathcal{S_{IPA}}$, 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 "नेहरू" in Hindi} have corresponding phonemic representations {"nerou", "næhru", "næru" and "næhru"} respectively, only the English "*Nehru*" and the Hindi "नेहरू" are phonetically same. ◇

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

Definition 2.2**:** Two strings s_i and s_j are *phonetically close* if and only if $\{\mathsf{editdistance}(p_i,p_j) \leq t\}$, 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 LexEQUAL matching Ψ operator, based on Definition 2.2, is defined as follows:

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

2.1.1 LexEQUAL(Ψ) Operator Algorithm

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

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

Figure 2.1: **The LexEQUAL Algorithm**

The LexEQUAL operator algorithm for comparing multiscript strings is shown in Figure 2.1. The operator accepts two multilingual text strings and a match threshold value as input. The strings are first transformed to their equivalent phonemic strings and the edit distance between them is then computed. If the edit distance is less than the threshold value, a positive match is flagged.

The `transform` function takes a multilingual string in a given language and returns its phonetic representation in IPA alphabet. Such transformation may be easily implemented by integrating standard TTP systems that are capable of producing phonetically equivalent strings. The `editdistance` function [8] takes two strings and returns the *edit distance* between them. A *dynamic programming* algorithm is implemented for this computation, due to the flexibility that it offers in experimenting with different cost functions.

Match Threshold

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

Intra-Cluster Substitution Cost

The three cost functions in Figure 2.1, namely *InsCost, DelCost* and *SubsCost*, provide the costs for inserting, deleting and substituting characters in matching the input strings. With different cost functions, different flavors of edit distances may be implemented easily in the above algorithm. we implemented a *Clustered Edit Distance* parameterization, by extending the *Soundex* [14] algorithm to the phonetic domain, under the assumptions that clusters of like phonemes exist and a substitution of a phoneme from within a cluster is more acceptable as a match than a substitution from across clusters. Hence, near-equal phonemes are clustered, based on the similarity measure as outlined in [17], and the substitution cost within a cluster is made a tunable parameter, the *Intra-Cluster Substitution Cost*. This parameter may be varied between `0` and `1`, with `1` simulating the standard *Levenshtein* cost function and lower values modeling the *phonetic proximity* of the like-phonemes.

2.1.2 LexEQUAL(Ψ) Operator Properties

The LexEQUAL(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 SemEQUAL Functionality Definition

The definitions in this section assume only that the values of an attribute are from a specified domain, $\mathcal{D}$ [1], 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, the following definitions for *semantic matching* were provided [16].

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), a weaker version of the semantic equality is defined [16], 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 $X \cap \mathcal{T}_{\mathcal{H}}(Y) \neq \phi$.

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

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

2.2.1 WordNet-based Taxonomic Hierarchies

WordNet [40] 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 [30], and is planned for others [33]. Figure 2.2 shows a simplified hierarchy in English and German (in solid lines) and the inter-linking of noun forms between English and German using ILI links (in dotted lines).

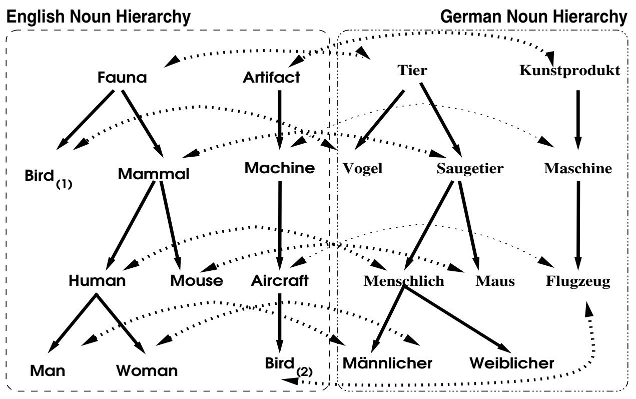


Figure 2.2: **Sample Interlinked WordNet Hierarchy**

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

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

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

2.2.2 The SemEQUAL(Φ) Operator Algorithm

The skeleton of the Φ algorithm to match a pair of multilingual strings is outlined in Figure 2.3.

```
SemEQUAL ($String_{Data}$, $String_{Query}$, $L_D$, $L_Q$, match, $\mathcal{T_L}$)
Input: $String_{Data}$ and $String_{Query}$ in languages $L_D$ and $L_Q$, match
flag, Target Languages $\mathcal{T_L}$
Output: TRUE or FALSE, [Optional] Gloss of Matched Synset
1.  ($\mathcal{W_D}$,$\mathcal{W_Q}$) ← WordNet Of ($L_D$,$L_Q$);
2.  ($\mathcal{S_D}$,$\mathcal{S_Q}$) ← Synsets of ($String_{Data}$ in $\mathcal{W_D}$, $String_{Query}$ in $\mathcal{W_Q}$);
3.  if $Match$ is EQUIVALENT then if $\mathcal{S_D} \cap \mathcal{S_Q} \neq \phi$ return true
else return false;
        else if $Match$ is GENERALIZED then
        $\mathcal{TC_Q}$ ← TransitiveClosure($\mathcal{S_Q}, \mathcal{W_Q}, \mathcal{T_L}$);
        if $\mathcal{S_D} \cap \mathcal{TC_Q} \neq \phi$ return true else ret urn false;
4.  [Optional.] return Gloss of the Matched Synset;
```

Figure 2.3: **Semantic Matching 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 $\mathcal{H}_{\mathcal{ML}}$, if it is a 'GENERALIZED' match. In the case of 'EQUIVALENT' match, it returns true, if LHS string maps to the RHS operand in the taxonomic network $\mathcal{H}_{\mathcal{ML}}$.

2.2.3 The SemEQUAL(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 $A \ \Phi \ B$ is true, iff A is a descendant of B, in $\mathcal{H}$. Or, equivalently, $A \ \Phi \ B$ 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.

2.3 Cost Models for Operators

In this section we discuss the cost models of multilingual operators. 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 2.1 are used and the costs of operations (in *big-O* notation) are given in Table 2.2.

All *edit-distance* computations are assumed to be implemented using *dynamic-programming* [18] algorithm instead of the standard *diagonal transition* algorithm, for the reason given in section 2.1.1. For Ψ operation costs with indexes, the indexes are assumed to be created on the materialized phoneme strings. For approximate index the fraction of the database scanned is assumed to be proportional to the error threshold (t).

2.4 Estimations of Operator Output

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

Symbol	**Represents**
LHS (L) and RHS (R) Operands	
R_L, R_R	No. of Records in L, R
l_L, l_R	Avg. length of Records in L, R
P_L, P_R	No. of Pages in L, R
A_L, A_R	No. of Pages for Approximate Index in L, R
k	Ψ Error Tolerance (as a fraction in $(0, 1]$)
σ	Ψ Size of the Alphabet (= $\|\Sigma\|$)
R_H	No. of Records storing $\mathcal{H}$ (=$\|\mathcal{H}_{\mathcal{ML}}\|$)
$P_{\mathcal{H}}$	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 2.1: **Symbols used in Analysis**

O	**Remarks**	**Algorithm Complexity**	**Disk I/O**
scan Operations			
Ψ	No Index	$R_L l_L l_R$	P_L
Ψ	Approx. Idx	$R_L l_L l_R k$	A_L
Φ	No Index	R_L+$R_{\mathcal{H}}(h$+$1)$	$P_{\mathcal{H}}(h$+$1)$
join Operations			
Ψ	No Index	$R_L R_R l_L l_R \sigma$	$3(P_L + P_R)$
Ψ	Approx. Idx	$R_L R_R l_L l_R k \sigma$	$A_L + A_R$
Φ	No Index	$R_L + R_R + R_R R_{\mathcal{H}}(h$+$1)$	$3(P_L$+$P_R)$+$P_{\mathcal{H}}$

Table 2.2: **Cost Models for Operators**

2.4.1 Estimation of Cardinality of LexEQUAL Output

We estimated the output of the Ψ operator, by modifying the equality (=) operator cardinality estimation, keeping in mind that each value in the left hand side can be matched with more than one unique value on the right hand side.

2.4.2 Estimation of Cardinality of SemEQUAL Output

We estimate the output size of Φ operator, using the notation in Table 2.1, 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 $R_L(h+1)/|\mathcal{H}_{\mathcal{ML}}|$.

Chapter 3

PostgreSQL

PostgreSQL [36] is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. PostgreSQL is an open-source descendant of this original Berkeley code. It supports SQL99 and offers many modern features: complex queries, foreign keys, triggers, views, transactional integrity, multiversion concurrency control. Also, PostgreSQL can be extended by the user in many ways, for example by adding new data types, functions, operators, aggregate functions, index methods, procedural languages.

And because of the liberal license, PostgreSQL can be used, modified, and distributed by everyone free of charge for any purpose, be it private, commercial, or academic.

3.1 PostgreSQL Query flow

PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

* A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called the postmaster.

* The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution, most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts.

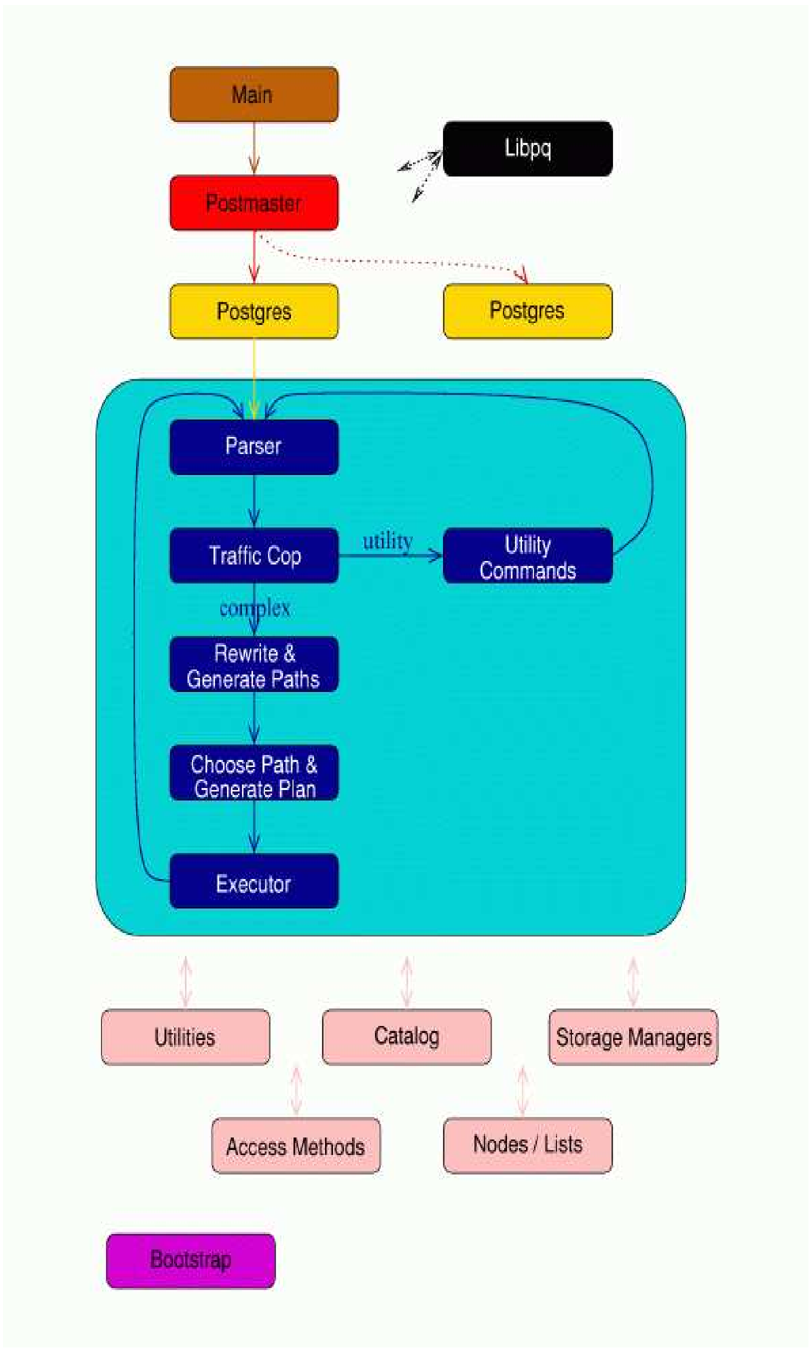


Figure 3.1: **PostgreSQL Query flow diagram**

In that case they communicate over a TCP/IP network connection. The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts (”forks”) a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original postmaster process. Thus, the postmaster is always running, waiting for client connections, whereas client and associated server processes come and go.

A query comes to the backend via data packets arriving through TCP/IP or Unix Domain sockets. It is loaded into a string, and passed to the parser, the parser stage checks the query transmitted by the application program for correct syntax and creates a query tree. The rewrite system takes the query tree created by the parser stage and looks for any rules (stored in the system catalogs) to apply to the query tree. It performs the transformations given in the rule bodies. One application of the rewrite system is in the realization of views. Whenever a query against a view (i.e. a virtual table) is made, the rewrite system rewrites the user’s query to a query that accesses the base tables given in the view definition instead. The planner/optimizer takes the (rewritten) query tree and creates a query plan that will be the input to the executor. It does so by first creating all possible paths leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each plan is estimated and the cheapest plan is chosen and handed back. The executor recursively steps through the plan tree and retrieves rows in the way represented by the plan. The executor makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the rows derived.

3.2 GiST Indexes

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B+-trees, R-trees and many other indexing schemes can be implemented in GiST.

One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

3.2.1 Extensibility

Traditionally, implementing a new index access method meant a lot of difficult work. It was necessary to understand the inner workings of the database, such as the lock manager and Write-Ahead Log. The GiST interface has a high level of abstraction, requiring the access method implementor to only implement the semantics of the data type being accessed. The GiST layer itself takes care of concurrency, logging and searching the tree structure.

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B+-trees and R-trees. That means that you can use PostgreSQL to build a B+-tree or R-tree over any data type you want. But B+-trees only support range predicates (<, =, >), and R-trees only support n-D range queries (contains, contained, equals).

So if you index, say, an image collection with a PostgreSQL B+-tree, you can only issue queries such as ”is imagex equal to imagey”, ”is imagex less than imagey” and ”is imagex greater than imagey”? Depending on how you define ”equals”, ”less than” and ”greater than” in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps ”find all images of horses” or ”find all over-exposed images”.

All it takes to get a GiST access method up and running is to implement seven user-defined methods, which define the behavior of keys in the tree. Of course these methods have to be pretty fancy to support fancy queries, but for all the standard queries (B+-trees, R-trees, etc.) they’re relatively straightforward. In short, GiST combines extensibility along with generality, code reuse, and a clean interface.

3.2.2 Implementation

There are seven methods that an index operator class for GiST must provide:

consistent

Given a predicate p on a tree page, and a user query, q, this method will return false if it is certain that both p and q cannot be true for a given data item.

union

This method consolidates information in the tree. Given a set of entries, this function generates a new predicate that is true for all the entries.

compress

Converts the data item into a format suitable for physical storage in an index page.

decompress

The reverse of the compress method. Converts the index representation of the data item into a format that can be manipulated by the database.

penalty

Returns a value indicating the ”cost” of inserting the new entry into a particular branch of the tree. items will be inserted down the path of least penalty in the tree.

picksplit

When a page split is necessary, this function decides which entries on the page are to stay on the old page, and which are to move to the new page.

same

Returns true if two entries are identical, false otherwise.

3.2.3 Limitations

The current implementation of GiST within PostgreSQL has some major limitations: GiST access is not concurrent; the GiST interface doesn't allow the development of certain data types, such as digital trees (see papers by Aoki et al); and there is not yet any support for write-ahead logging of updates in GiST indexes.

Because of the lack of write-ahead logging, a crash could render a GiST index inconsistent, forcing a REINDEX.

More information about GiST implementation in PostgreSQL can be found at:

http://www.sai.msu.su/ megera/postgres/gist/.

Chapter 4

M-Tree

Since the computation of edit distance between two phoneme strings is very expensive and existing index structures in PostgreSQL such as B+ trees used in exact matching prove inadequate for similarity searching, we added a metric(similarity) index to improve the performance of LexEQUAL operator. In this chapter we present brief description about M-Tree [5] index, which is a similarity index structure.

4.1 Metric space

A metric space comprises of a collection of objects and an associated distance function satisfying the following properties.

Symmetry:

d(a, b) = d(b,a)

Non negativity:

d(a, b) $>$ 0 if a, b are not equal and

d(a, b) = 0 if (a = b)

Triangle inequality:

d(a, b) $\leq$ d(a, c) + d(c, b)

where a, b, c are objects of the metric space.

Edit distance (Levenshtein distance) satisfies the above mentioned properties. The edit distance between two strings is defined as the total number of simple edit operations such as additions, deletions and substitutions required to transform one string to another. For example, consider the strings paris and spire . The edit distance between these two strings is 4,as the transformation of paris to spire requires one addition, one deletion and two substitutions. Edit distance computation is expensive since the algorithmic complexity is O(mn) where m, n are the length of the strings compared.

Since we used edit distance to measure the similarity between two phoneme strings, we can use metric indexes to prune the search space. Metric indexes make use of the triangle inequality to prune the search space. For example consider an element p with an associated subset of elements X such that for all x in X,

$$d(p,x) \leq k.$$

We want to find all strings within edit distance e from given query string q. That is reject all strings x such that

$$d(q,x) > e \tag{1}$$

From the triangle inequality,

$$d(q,p) \leq d(q,x) + d(x,p)$$

Hence

$$d(q,x) \geq d(q,p) - d(x,p)$$

which reduces to

$$d(q,x) \geq d(q,p) - k \tag{2}$$

From equations (1) and (2), the criterion reduces to

$$d(q,p) - k > e$$

If the inequality is satisfied, the entire subset X is eliminated from consideration. However, we need to compute the O(mn) edit distance for all the elements in the subsets that do not satisfy the above criterion.

4.2 M-Tree index

M-Tree [5] is an m-ary tree using m routing objects. We select m routing objects for the first level. Together with each routing object is stored a covering radius that is the maximum distance of any object in the subtree associated with the routing object. A new element is compared against the m routing objects and inserted into the best subtree defined as that causing the subtree covering radius to expand less and in the case of ties selecting the closest representative. Thus it can be viewed that associated with each routing object pi, is a region of the metric space

$$Reg(pi) = (uinU|d(pi, u) < ri),$$

where ri is the covering radius. Further, if a subtree becomes full it is partitioned recursively. In the internal node, pi and ri are stored together with a pointer to the associated subtree. Further to reduce distance computations M tree also stores precomputed distances between each routing object and its parent.

For a given query string and search distance, the search algorithm starts at the root node and recursively traverses all the paths for which the associated routing objects satisfy the following inequalities. This algorithm is dynamic in nature as it allows both additions and deletions.

$$d(p^p i, q) - d(p^p i, pi) \leq ri + e \tag{5}$$

$$d(pi, q) \leq ri + e \tag{6}$$

In equation (5), we can take advantage of the precomputed distance between the routing object and its parent.

Chapter 5

Implementation in PostgreSQL

In this chapter we explore different alternatives to implement new functionality in PostgreSQL. We subsequently describe the system setup for our implementation and present how we added the multilingual functionalities in the PostgreSQL open-source database system, in all different implementation alternatives. We subsequently analyse the performance of core implementation and compare it with outside-the-server and inside the server implementations. Finally, we demonstrate the power of our core implementation in selecting efficient query execution plans, which is not possible with inside-the-server implementation.

5.1 Implementation choices

We explored different alternatives to implement new functionality in PostgreSQL, and identified three ways of implementation. They are *outside-the-server*, *inside-the-server* and *core* implementation of new functionality. As we move from outside to core we move closer and closer to the database engine.

Outside-the-Server Implementation: Outside-the-server implementation is the quick way to add new functionality to PostgreSQL. In this implementation new functionality lies outside the PostgreSQL server. This implementation can be carried out by using *user defined function* facility provided by PostgreSQL. UDF functions will be written in *PL/pgSQL*. Problem with this type of implementation is that we cannot implement optimizer aware functions. Moreover these functions won't be executed inside the server address space when invoked, adding lot of overhead. So, even though implementation is easier compared to others this will add lot of overhead. This implementation can be migrated from

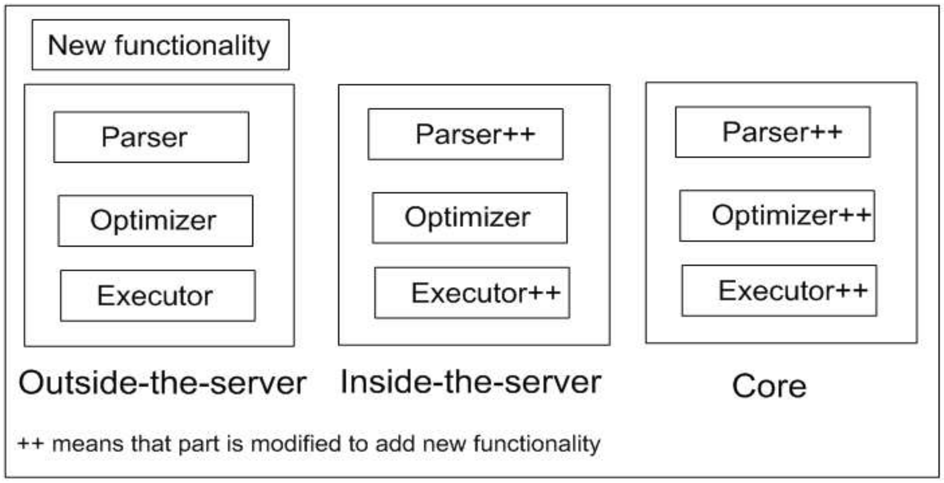


Figure 5.1: **Different implementation choices to add new functionality to PostgreSQL database engine**

one version of PostgreSQL to other versions without any difficulty. With little effort this implementation can also be migrated to other database engines.

Inside-the-Server Implementation: Inside-the-server implementation of new functionality aims at creating new functionality, which will be executed inside the server space. In this implementation new functionality lies inside the PostgreSQL engine. This implementation can be carried out by using *C-Language function* facility provided by PostgreSQL. These *C-Language functions* will be written in C. We can create a new operator from these functions, provided function has one or two arguments. Problem with this type of implementation is that still optimizer is not operator aware. One more problem with this type of implementation is operators won't get created when you install PostgreSQL. We have to create operators for each database we create adding overhead to the user. But this implementation rectifies the other major overheads caused by outside-the-server implementation using UDF functions. This implementation can also be migrated to other versions of PostgreSQL without any difficulty like outside the server implementation. But this implementation cannot be migrated to other database engines.

Core Implementation: This is the hard way to implement new functionality but is the ideal way. As the name suggests new functionality lies in the heart of PostgreSQL. In this implementation we have to make new functionality equivalent to other first class operators available in PostgreSQL (like =,$\leq$,$\geq$ etc). In this implementation we have to make optimizer operator aware and operator should execute inside the server address space.

Moreover operator should be created at the time of installing PostgreSQL, removing the overhead of creating them for each and every database separately. For making optimizer operator aware we have to make modifications to the code of PostgreSQL optimizer, so that it will recognize this new operator. This will eliminate all the overheads given for previous implementations. This will make operator equivalent to any other native database operator that comes with PostgreSQL in terms of execution.

5.1.1 System Setup for Implementation

The implementation of the multilingual functionality was done on the PostgreSQL open-source database system [36] (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. We implemented new multilingual operators for the purpose of adding multilingual functionality to PostgreSQL. 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 [29], was integrated with the system, after appropriate modification to output the phonemic strings in IPA alphabet and to made it a callable routine from the query processing engine.

5.2 Outside-the-Server Implementation

Outside-the-server implementation is carried out by using user defined function facility provided in PostgreSQL. PL/pgSQL is used to create the new functionality using this type of implementation. New functionality can be added using the command given in Figure 5.2. But the problem is functionality added this way cannot be made into an operator.

```
CREATE FUNCTION lexequal(varchar,
varchar, numeric) RETURNS boolean AS
' .........(lexequal code) '
LANGUAGE SQL;
```

Figure 5.2: **SQL statement to create PL/pgSQL function**

For outside-the-server implementation of LexEQUAL approximate matching functionality is added to the database server as a UDF. Lexical resources (e.g., script and IPA code tables) and relevant TTP converters that convert a given language string to its equivalent phonemes in IPA alphabet are integrated with the query processor. The cost matrix is made an installable resource intended to tune the quality of match for a specific domain.

SemEQUAL functionality is implemented along the same lines of LexEQUAL. WordNet resources are provided in the form of database tables.

5.3 Inside-the-Sever Implementation

Inside-the-server implementation is carried out by using C-Language function facility provided in PostgreSQL. PostgreSQL provides calling interface for these C-Language functions. Currently two calling interfaces Version-0 and Version-1 are available. We implemented these C-Language functions using version-0 calling interface provided by PostgreSQL. Version-1 calling interface is the recent one, but we selected version-0 to implement C-language functions as they are supported on both old and new versions of PostgreSQL.

The command used to create a new C-Language function in PostgreSQL is given in Figure 5.3.

```
CREATE FUNCTION lexequal(varchar,
varchar) RETURNS boolean AS ’
DIRECTORY/lexequalsource’, ’lexequal’
’ LANGUAGE C STRICT;
```

Figure 5.3: **SQL statement to create C-Language Function**

This implementation of C-Language function can be converted to an operator. Command to be used is given in Figure 5.4. Only unary and binary operators are available in PostgreSQL.

```
 CREATE OPERATOR Ψ (
leftarg = varchar,
rightarg = varchar,
procedure = lexequal,
commutator = Ψ );
```

Figure 5.4: **SQL statement to create operator**

Even though we can make an operator with this implementation we cannot claim it as first class operator as the optimizer of PostgreSQL is not aware of the new operator. More over these operator won't come with the installation of PostgreSQL, we have make these operators for every database we create, which is not the case with first class operators like =,$\leq$,$\geq$.

5.3.1 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 through a function call. The value of the parameter for matching may be globally set by the administrators, based on the requirements of a domain or application. The LexEQUAL matching function in Figure 2.1, is modified slightly to take the two strings as operator input, and the threshold from a global variable, and implemented in the system. A modified *Dhvani* text-to-phoneme converter was used to convert the multilingual strings to their phonemic representations. Currently our operator has support for English, Hindi, Kannada and slots are provided for users to integrate new TTPs for other languages. Currently language identification is done based on where the alphabet falls in Unicode code base. Hindi and Kannada TTPs are taken from Dhvani [29], a TTS developed for Simputer [37]. 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 a join query processing).

5.3.2 Φ Operator Implementation

The Φ operator was added to PostgreSQL system as a binary join operator, using the operator addition facility in the system. The SemEQUAL matching functionality, as given in Figure 2.3, was implemented in C. We implemented the generalized version of the algorithm. WordNet is taken as a resource file. Due to the high cost involved in computing closures on WordNet taxonomical hierarchies from the resource file, we pinned the WordNet 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.

5.4 Core Implementation

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 cost models and selectivity estimates. 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 our core implementation in selecting efficient query execution plans.

In order to make the new operator available along with the PostgreSQL installation, instead of manually creating it, we followed a different approach in the core implementation. We first created an PostgreSQL internal function and then created a first class operator which uses the internal function. We created internal function by first moving our code into PostgreSQL source tree and then by making the modification shown in Figure 5.5, in ”pg_proc.h” file. This modification creates an internal function called ”lexequal” in ”pg_catalog” schema, which takes two varchar arguments and returns boolean result.

```
 DATA(insert OID = 2188 ( lexequal
PGNSP PGUID 12 f f t f i 2
16 "25 25" lexequal - _null_ ));
```

Figure 5.5: **Code to make internal function**

Using the internal function lexequal we made first class operator by adding the statement shown in Figure 5.6, to ”pg_operator.h” file. This modification creates a binary commutative operator named ’Ψ’ in ”pg_catalog” schema, which is not merge or hash joinable and whose restriction and join selectivity functions are lexeqsel, lexeqjoinsel respectively. This approach does not effect he execution time of the operator, and provides the same performance that can be achieved with an inside-the-server implementation.

5.4.1 Ψ Operator Implementation

The code for implementing LexEQUAL (Ψ) operator is same as the code of inside the server implementation. To make core implementation we added selectivity and cost estimates of the operator to optimizer. We also made the user settable parameter lexthreshold equivalent to

```
DATA(insert OID = 2320 ( "Ψ" PGNSP
PGUID b f 25 25 16 2320 2321 0 0 0
0 lexequal lexeqsel lexeqjoinsel ));
```

Figure 5.6: **Code to make first class operator**

any other user settable parameters present in PostgreSQL like enable_hashjoin etc. For this we modified the files 'guc.c', 'tab-complete.c', 'postgresql.conf.sample'. The cost of the operator was set to the formulae given in Table 2.2, and the selectivity estimate using the methodology outlined in Section 2.4. To further speedup the performance we added an index and details about the index are given below.

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 [5]– 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 [5] 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 [10] 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 [24] 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.

5.4.2 Φ Operator Implementation

The code for implementing SemEQUAL (Φ) operator is same as the inside the server implementation. In this implementation to make operator first class we created operators by going through the procedure given above and added information required by the optimizer to avoid worst plans. We added simple cost models as given in Table 2.2 and cardinality estimations as given in Section 2.4, for use in the optimizer.

5.4.3 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 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.

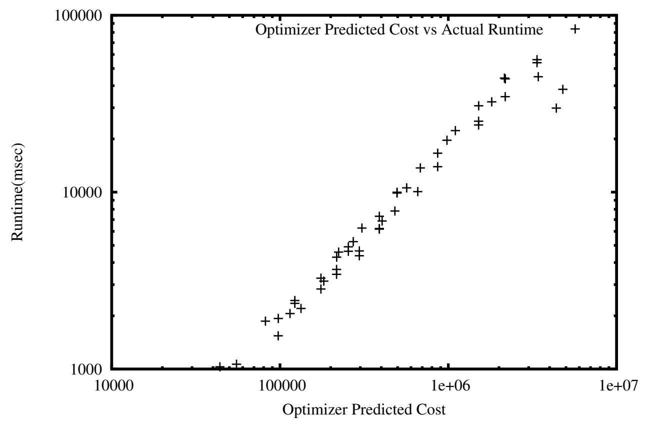


Figure 5.7: **Optimizer Prediction Performance**

Figure 5.7 plots the correlation between the predicted optimizer costs and the actual runtimes of the queries. X-axis gives optimizer prediction in the units of disk page accesses and Y-axis gives actual runtime in msecs. 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.

Chapter 6

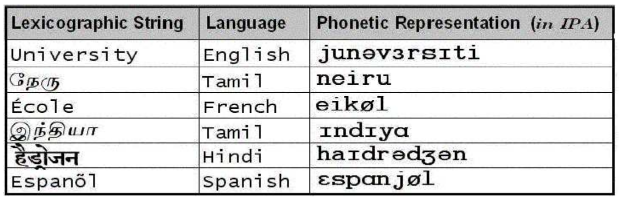
Performance Analysis

In this chapter, we outline our performance experiments with our implementation of multilingual operators on PostgreSQL database system. We first present a baseline performance using outside-the-server implementation. Since execution time of operators implemented in both inside-the-server and core approaches are same, we presented both of them combined in sections 6.4, 6.5 and show that performance will be improved by 2 to 3 orders of magnitude. Further demonstrate the power of core implementation in selecting efficient query execution plans, which is not possible with inside-the-server implementation.

6.1 Data Setup for Experiments

For Ψ operator we selected proper names from three different sources so as to cover common names in English and Indic domains. The first set consists of randomly picked names from the *Bangalore Telephone Directory*, covering most frequently used Indian names. The second set consists of randomly picked names from the *San Francisco Physicians Directory*, covering most common American first and last names. The third set consisting of generic names representing Places, Objects and Chemicals, was picked from the *Oxford English Dictionary*. Together the set yielded about 800 names in English, covering three distinct name domains. Each of the names was hand converted to two Indic scripts – Tamil and Hindi. As the Indic languages are phonetic in nature, conversion is fairly straight forward, barring variations due to the mismatch of phoneme sets between English and the Indic languages.

To convert English names into corresponding phonetic representations, standard linguistic resources, such as the *Oxford English Dictionary* [35] and TTP converters from [31], were used. For Hindi strings, *Dhvani* TTP converter [29] was used. For Tamil strings, due to the lack of access to any TTP converters, the strings were hand-converted, assuming phonetic nature of the Tamil language. Further those symbols specific to speech generation, such as the supra-segmentals, diacritics, tones and accents were removed. Sample phoneme strings for some multiscript strings are shown in Figure 6.1. This way we generated 2400 names in three different scripts.



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

Figure 6.1: **Phonemic Representation of Test Data**

Since above data size is very small, we synthetically generated a large data set from this multiscript lexicon. Specifically, we concatenated each string with all remaining strings *within a given language*. The generated set contained about 200,000 names, with an average lexico-graphic length of 14.71 and average phonemic length of 14.31. Figure 6.2 shows the frequency distribution of the generated data set – in both character and (generated) phonetic representations with respect to string lengths.

For Φ operator we wanted to use different WordNets, but since different WordNets are in different stages of development, for performance experiments we used English WordNet. Queries that compute closures of varying sizes were employed for profiling the Φ operator. All experiments were run on these datasets on a standard workstation, quiesced of all other activities.

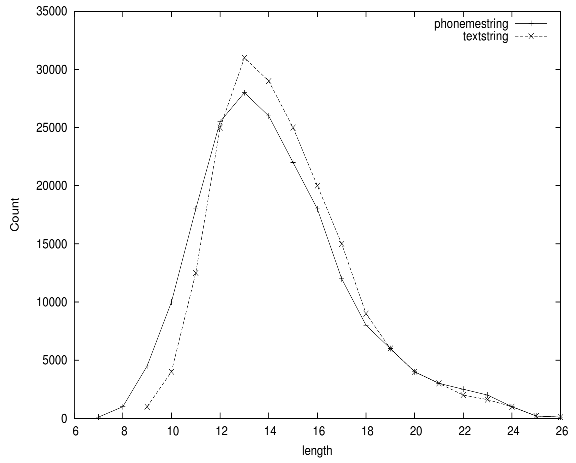


Figure 6.2: **Distribution of Generated Data Set (for Performance Experiments)**

We implemented a prototype of Ψ and Φ on top of the *PostgreSQL(Version 7.4.3)* database system. The multilingual strings and their phonetic representations (in IPA alphabet) were both stored in UTF-8 format. The algorithms shown in Figures 2.1, 2.3 were implemented, as a UDF in the PL/SQL language and as C-Language function respectively.

6.2 Baseline Performance of Ψ Operator

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.

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

Table 6.1: ***Outside-the-Server*** **Performance of Ψ Operator**

Table 6.1 lists the performance of *outside-the-server* implementation of the Ψoperator, in scanning $200,000$ row table and joining two tables with 450 rows each, without and with appropriate indexes on the materialized phonemic strings. The performance values with metric index show the best performance possible with outside-the-server implementation with ideal metric index. The results show clearly that while the *no-index* implementation is expensive, the metric index structure can help in reducing the cost by nearly an order of magnitude. The main impediment to the performance is the expensive UDF invocations.

6.3 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 computations on relational tables are recognized to be expensive[2, 9, 11]. 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 6.2 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	5.67	0.02
482	17.7	0.09
2041	75.6	0.64
2538	95.7	0.90
5340	201.8	2.77
11551	840.5	11.6

Table 6.2: ***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 B+-Tree 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.

6.4 Core (or Inside-the-server) Performance of Ψ Operator

After implementing the Ψ operator in core as indicated in the section 5.4, we ran the same experiments that were used for testing the performance of the *outside-the-server* implementation. For LexEQUAL experiments, the GiST index (implementing the M-Tree) was also used for enhancing the performance. We provide, in Table 6.3, the performance of queries scanning the same $200,000$ row table and joining same 450 row tables that were used in outside-the-server experiments for the Ψ operator.

Query Type	**Scan-type** (*Sec.*)	**Join-type** (*Sec.*)
Base Performance	5.2	1.9
M-Tree	4.2	1.9

Table 6.3: ***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 6.1. As can be seen, M-Tree didn't improved the performance as expected in this case. But it is helpful in other cases like, when duplicates are present and when difference in average lengths between the tables is high enough for it to rule out lot of cases. We measured the performance gain due to the GiST index on scan of $1000,000$ row table and $450X1000,000$ join, and Figure 6.3 highlights the relative performance gain due to the M-Tree index. We note that the performance of the Ψ operator is significantly speeded up by the M-Tree index, upto nearly 90%.

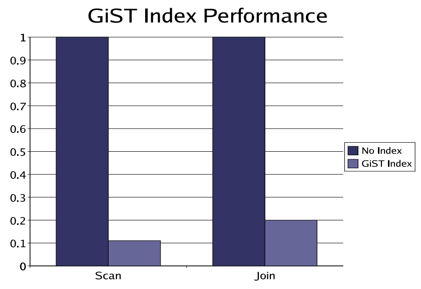


Figure 6.3: **Performance of M-Tree Index**

6.5 Core (or Inside-the-server) Performance of Φ Operator

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 6.4.

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 6.4: ***Core* Performance of Φ Operator**

Compared with the outside-the-server performance of Φ operator (as shown in Table 6.2), 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 $2,000$ [16].

6.6 A Motivating Optimization Example

We illustrate the power of the core implementation to distinguish between efficient and inefficient execution plans with the new operators, by the following example:

Example 6.1**:** 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 6.4) capture the semantics of the above query:

Plan 1: $\Pi_{A.AuthorID,P.PubID,B.BookID}$
$(\sigma_{(Threshold \leq 0.25)}(\Psi_{A.AName,P.PName}(P, A)$
$(B \bowtie_{BookID} (A \bowtie B))))$

Plan 2: $\Pi_{A.AuthorID,P.PubID,B.BookID}(\bowtie_{BookID} (A, B)$
$(\sigma_{(Threshold \leq 0.25)}(\Psi_{A.AName,P.PName}(P, A))))$

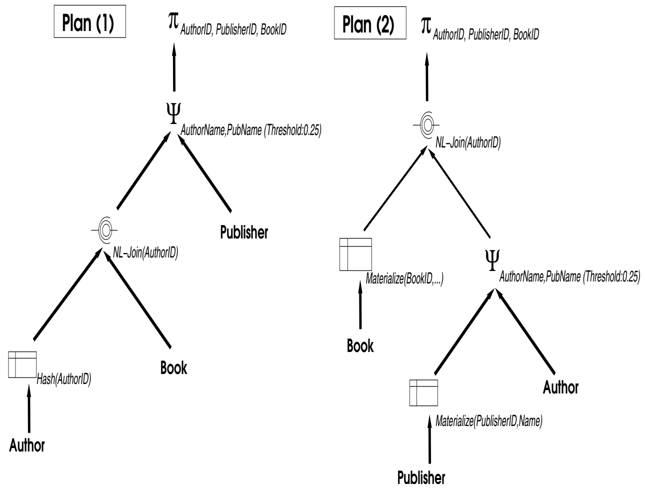

Plan (1)
π AuthorID, PublisherID, BookID
Ψ AuthorName,PubName (Threshold:0.25)
NL-Join(AuthorID)
Publisher
Hash(AuthorID)
Book
Author
Plan (2)
π AuthorID, PublisherID, BookID
NL-Join(AuthorID)
Materialize(BookID,...)
Book
Ψ AuthorName,PubName (Threshold:0.25)
Materialize(PublisherID,Name)
Author
Publisher

Figure 6.4: **Query Plans for Example** 6.1

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. $\diamond$

Inside-the-server implementation won't have the above power to distinguish between efficient and inefficient execution plans.

Chapter 7

Conclusions

With this project we have made the first step towards the ultimate objective of achieving complete multilingual functionality in database systems. In this report, 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 multilingual 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 inputs to the relational query optimizer. Subsequently, we presented strategies to add such functionality to PostgreSQL database engine. Given the enormous effort and impracticality involved in adding such functionality to stable database system implementations, we proposed a staged implementation roadmap, starting with *outside -the-server* implementation using user-defined functions (UDF) and existing SQL:1999 features, that may be implemented with no change to database server. However, the results confirm that performance may not be sufficient for practical implementations, due to the high overheads associated with outside-the-server implementation.

Hence, we explore *inside-the-server* implementation of the functionality as database operators. But in this implementation relational optimizer is unaware of new operators making it inefficient. So, we explore *core* implementations of the functionality as first-class operators inside the database kernel with selectivity estimates and cost models for new operators. In order to optimize the performance of LexEQUAL 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 select efficient execution plans. Thus our *core* implementation of multilingual functionality on PostgreSQL database system represents the first step towards the ultimate objective of realizing *natural-language-neutral* database engines.

Chapter 8

Future Research

In this project we tried to implement LexEQUAL and SemEQUAL functionality inside the PostgreSQL database engine. Even though we completed core implementation of LexEQUAL its functionality is limited to few languages. TTPs for other languages can be added to LexEQUAL to make it really multilingual. Currently language identification is done by identifying where the alphabet falls in the unicode code base. This approach works fine for Indian languages, but fails for some European languages which share same script. To rectify this, we have to add a new data type to the PostgreSQL, which stores string as well as language to which it belongs. Currently there is an inefficiency in the join processing when phoneme computation is done online. To rectify this we have to materialize the phonemes and reuse them in the join computation. We added M-Tree index to PostgreSQL to improve performance of LexEQUAL operator, but it is not improving performance in all the cases. Here there is a good research problem to implement new approximate index for this domain.

We implemented SemEQUAL by taking WordNet as a resource file, since recursive-SQL functionality is not present in PostgreSQL. But ideal way of implementing it would be to take WordNet as a table and use recursive-SQL query to compute closure. For this we have to first implement recursive-SQL functionality in PostgreSQL and then we have to make SemEQUAL operator to use this to compute results.

Bibliography

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

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

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

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

[5] 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.

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

[7] 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.

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

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

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

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

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

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

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

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

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

[17] P. Mareuil, C. Corredor-Ardoy and M. Adda-Decker Multilingual Automatic Phoneme Clustering *Proc. of 14th Intl. Congress of Phonetic Sciences,* August 1999.

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

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

[20] 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.

[21] 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.

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

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

[24] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[37] The Simputer. *http://www.simputer.org*

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

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

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