Integrating Multilingual Database Operators in PostgreSQL

A project report submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Internet Science and Engineering

by

Rupesh Bajaj

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JULY 2007

Dedicated

To

My Parents,
Bhaiya and Bhabhi

Acknowledgements

I sincerely express my gratitude to my advisor Prof. Jayant R. Haritsa for his guidance and enduring support. His constant association and critical appraisal has helped a great deal in the completion of this project in all its aspects.

I am also indebted to all my DSL labmates Tarun, Shruthi, Pooja, Sharat for providing a lively and pleasant company. My friends made my stay at IISc quite memorable. I also like to thank Sudipta Chattopadhyay, Ashutosh Bhatia, Ravikant Chaudhary, Sandeep Tandekar, Mitesh Jat, Vijay Prakesh, Mahesh Sonal, Nikesh Srivastava and Vikas Sharma for their moral support. There are several people in CSA and IISc who deserve acknowledgment for their help in successful completion of this work. I thank all the faculty and staff members of the department for their support and help. I cannot express my gratitude in words to my parents, Bhaiya and Bhabi for their unconditional support and encouragement.

Abstract

With the increasing integration of global economy and proliferation of languages other than English into information systems, capability to store and manage data in multiple languages simultaneously is of vital importance. The problem of Multilingual database tables and cross-lingual query operators has been previously dealt with and two cross-lingual operators **MLLexEqual** (*phonemic name matching*) and **MLSemEqual** (*semantic concept matching*) were introduced. We introduce **MLLike** (*phonemic regular expression matching*) operator.

In this report, we focus on efficient implementation of MLLike, MLLexEqual and MLSemEqual operators, investigating issues related to implementation of these operators inside relational engines. Specifically, we implemented these operators inside the **PostgreSQL** database system. Currently, we have implemented these operators for Hindi and English languages, but our approach can be extended to support any language. This is the first core implementation of these operators inside a database engine.

Also, we investigate **Slim tree** and **DF tree Index** as a method towards optimization of the MLLexEqual operator and address issues related to adapting it to large scale selectivity. We investigated **Hopi Index** to optimize the MLSemEqual operator. We have evaluated this indexes for MLLexEqual and MLSemEqual operators.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The recent times have seen a huge proliferation of internet and similar other communication systems with ever greater interoperability amongst themselves. At the same time, hardware and carrier costs have come down drastically, making them available to more and more people across the globe. Consequently, today's information systems are dealing with a large amount of data which are in languages other than English. Currently, multilingual data is stored in isolation with one another as different datasets and are used in isolation to one another. Cross-lingual datasets are very rare owing to unavailability of operators that can deal with them in some meaningful manner. This is despite the fact that there are several cross-lingual queries that users might wish to ask. For example, there are many e-governance and e-commerce portals or search engines where data may be available in various languages. One might wish to query certain kind of information over all the languages. Currently, this is not possible in most information systems. Consider a hypothetical e-commerce application, *Books.com*, that sells books across the globe; the **MLBooks** table storing data in multiple languages or a logical view assembled from data source of several databases, shown in Figure 1.1, can be a possible schema for viewing data about all the books that *Books.com* has in its inventory.

MLBooks

Title	Author	Category
भारत एक खोज	नेहरू	इतिहास
मेरी कहानी	नेहरू	आत्मकथा
Letters to my daughter	Nehru	History
EngineeringMathematics	R S Grewal	Mathematics
अंतिम अरण्य	निर्मल वर्मा	उपन्यास
Intoducing NLP	Sue Knight	NLP
गणित	आर्यभट	गणित
सान्ख्यकि किताब	महालिनोबिस	सान्ख्यकि

Figure 1.1: Multilingual Books.com

In such an environment, a user may wish to ask many queries, some of which may be of the type given below:

1. A user might wish to find out all the books written by author whose name starts with **'न'** in a particular set of languages (possibly all).

2. A user may want to find out all the books written by a specific author in a particular set of languages.

3. A user might wish to find out all the 'गणित' books that are available in a certain set of languages.

These requirements for cross-lingual queries have been addressed through the three operators: **MLLike**, **MLLexEqual** [2], **MLSemEqual** [3]. For the first query, the **MLLike** operator is used. This operator takes as input, a regular expression in one language, for example 'न%', and returns all the *phonemically* close matches in user specified set of languages. For the second type of query **MLLexEqual** operator is used. This operator takes as input, a name in one language ('नेहरु'in Hindi, for example), and returns all the *phonemically* close names in a user specified set of languages. The third type of query presents a different type of problem. Here, one has to find out *semantic* similarity between words. For this purpose, **MLSemEqual** operator is used. It finds out the words

belonging to different languages that are semantically related to each other. In order to do this, it uses available ontology like **WordNet** [14].

Our word is different from Kumaran's work [1] as in their work they have made the assumtion that whole of wordnet can be store in the main memory. Also their MTree-index has write-ahead logging error. Whereas in our work their is no such assumption.

Chapter 2

The Multilingual Operator

2.1 The MLLike Operator

2.1.1 MLLike Definition

The current SQL LIKE operator is used to find all the strings which satisfy a given regular expression. For example in order to find all the Author names that begin with 'न', the SQL query will be:

```
Select *
from MLBooks
where Author like ' न%'
```

This query will return only those tuples whose Author names starts with 'न' in Hindi. Despite the fact that MLBooks also contains the books whose Author name starts with 'N' in English. Our new MLLike operator can be used for such queries. MLLike operator provides the *phonemic regular expression* matching in user specified set of languages. A query using MLLike and the consequent result is given in Figure 2.1. If the user has knowledge of all the target languages, such a query can be written using the current SQL *LIKE* operator. This can be done for the query with constant regular expression. But a join between two tables is not possible without phoneme regular expression matching.

```
SELECT Title, Author, Category
FROM MLBooks
WHERE Author InLanguages {English, Hindi}
MLLike 'न%' ;
```

Title	Author	Category
भारत एक खोज	नेहरू	इतिहास
मेरी कहानी	नेहरू	आत्मकथा
Letters to my daughter	Nehru	History
अंतिम अरण्य	निर्मल वर्मा	उपन्यास

Figure 2.1: Multilingual Like Query and Result Set

2.1.2 MLLike syntax

The syntax for MLLike operator is:

```
a_expr [InLanguages {L1, L2, ......}]
MLLike
c_expr [InLanguages {L1, L2, .....}]
[With escape ('esc_char')]
```

where *a_expr* can be any column name, *c_expr* can be any column name or any regular expression. The first *InLanguages* provides the set of languages in which the left hand column value is expected. The second *InLanguages* provides the set of languages in which the right hand constant or column value is expected. The *esc_char* provides the character to be used as escape character in the regular expression.

2.2 The MLLexEqual Operator

2.2.1 MLLexEqual Definition

The **MLLexEqual** operator provides a *phoneme* matching functionality that is used in cases where one has to determine if two words are phonemically equivalent. For example, let us take the example of *Books.com*. Suppose, a user wants to find out all the books whose *Author* is phonemically equivalent to 'नेहरु' in English and Hindi. In current

databases you have to give a query as given in Figure 2.2.

```
SELECT Title, Author, Category
FROM MLBooks
where      Author = 'Nehru'  OR    Author = 'नेहरू';
```

Title	Author	Category
भारत एक खोज	नेहरू	इतिहास
मेरी कहानी	नेहरू	आत्मकथा
Letters to my daughter	Nehru	History

Figure 2.2: SQL:1999 Compliant Multilingual Names Query and Result Set

Such a query specification that requires the author's name in several languages is undesirable, due to requirement of linguistic expertise of the user and the availability of special lexical resources in several languages for the query input. For the query involving a constant, user can specify the query to the current database engine. But for the join (of say किताब and Books relations explained latter) user can't write the query in current database systems. If **MLLexEqual** operator is used for the above query, it will return all books which fuzzily match with Nehru in the given language set, available in catalog. A query using **MLLexEqual** and the consequent result is given in Figure 2.3. Note that the tuples returned by the query may not be same as above query. Quality of answer depends on the *threshold* parameter and *text-to-phoneme*[TTP] convertor.

MLLexEqual join operator is defined as follows: MLLexEqual takes an input name in one language and returns all records that have the same name in all or in a user-specified set of languages. The input query name may be specified in the most comfortable language for the user. The *threshold* parameter specified in the query determines the quality of matches.

```
SELECT Title, Author, Category
FROM MLBooks
WHERE Author InLanguages {English, Hindi}
MLLexEqual 'नेहरू' withthreshold 0.25;
```

Title	Author	Category
भारत एक खोज	नेहरू	इतिहास
मेरी कहानी	नेहरू	आत्मकथा
Letters to my daughter	Nehru	History

Figure 2.3: A Multilingual Name Query

2.2.2 MLLexEqual Syntax

The syntax for MLLexEqual operator is:

```
a_expr [InLanguages {L1, L2, .....}]
MLLexEqual
c_expr [InLanguages {L1, L2, .....}]
Withthreshold threshold_value
```

where *a_expr* can be any column name and *c_expr* can be any column name or any constant. The first *InLanguages* provides the set of languages in which the left hand column value is expected. The second *InLanguages* provides the set of languages in which the right hand constant or column value is expected. The constant *threshold_value* is any non-negative float value which provides the threshold value for fuzzy matching. The threshold_value is multiplied by the length of smaller of two strings to get the actual threshold value.

2.3 The MLSemEqual Operator

2.3.1 MLSemEqual Definition

The **MLSemEqual** is an *ontology* matching functionality that is used in cases where one has to determine if two words are ontologically equivalent. For example, let us take the

example of *Books.com*. Suppose a user wants to find out all the books whose *Category* is ontologically equivalent to 'गणित' in a set of languages. In today's databases if you give a query with **Category = 'गणित'** selection condition, only those books whose category is 'गणित' in Hindi will be returned despite the fact that the catalog also contains गणित books in Hindi, English, and other ontologically related books in these languages. **MLSemEqual** operator will provide all the books whose category is ontologically related to 'गणित' in user specified set of languages. A query using **MLSemEqual** and the consequent result is given in Figure 2.4.

```
SELECT Title, Author, Category
FROM MLBooks
WHERE Category InLanguages {English, Hindi}
MLSemEqual 'गणित' ;
```

Title	Author	Category
Engineering Mathematics गणित सान्ख्यिकि किताब	R S Grewal आर्यभट महालिनोबिस	Mathematics गणित सान्ख्यिकि

Figure 2.4: Multilingual Semantic Selection

In order to determine the ontological equivalence of word-forms across languages in **MLSemEqual** operator, **WordNet** [14] is used. The basic idea is to use the **WordNet** to find out the intra-language ontological equivalence to match words. We have given two subclasses of MLSemEqual: MLSemEqualall and MLSemEqual. MLSemEqualall gives the ontologically related words (sub-tree under the given word) whereas the MLSemEqual gives the ontologically equivalent words(Inter languages index).

2.3.2 MLSemEqualall syntax

The syntax for MLSemEqualall Operator is:

```
a_expr InLanguages {L1, L2, ......}
MLSemEqualall
```

```
c_expr [InLanguages {L1, L2, .....}]
```

where *a_expr* can be any column name and *c_expr* can be any column name or any constant. The first *InLanguages* provides the set of languages in which the left hand column value is expected. The second *InLanguages* provides the set of languages in which the right hand constant or column value is expected.

2.3.3 MLSemEqual syntax

The syntax for MLSemEqual operator is:

```
a_expr InLanguages {L1, L2, ......}
MLSemEqual
c_expr [InLanguages {L1, L2, .....}]
```

Explaination of syntax is same as above.

Chapter 3

Design and Implementation of MLPostgreSQL

3.1 Design and Implementation of MLPostgreSQL

3.1.1 Database Engine

As a platform for implementing the multilingual operators (MLLike, MLLexEqual, MLSemEqual) PostgreSQL is chosen. **PostgreSQL** is an **Object Relational Database Management System**. It is an open source database engine. It comes with all the advanced features of a contemporary database system. It has a full fledged optimizer, rewriter, stored procedures etc. It supports complex queries, foreign keys, triggers, views, transactional integrity etc. Also **PostgreSQL** [13] can be extended by users in many ways, for example adding new data types, functions, operators, aggregate functions, index methods, procedural languages etc. Its source code is released under a flexible BSD type license and is available for modification. It was a natural choice because all the issues related to multilingual operators implementation will be encountered while implementing it in PostgreSQL. This also means that all the possible avenues of implementation will be available and a complete and comprehensive implementation can be worked out. The path for the query in PostgreSQL database engine is shown in Figure 3.1. For implementation

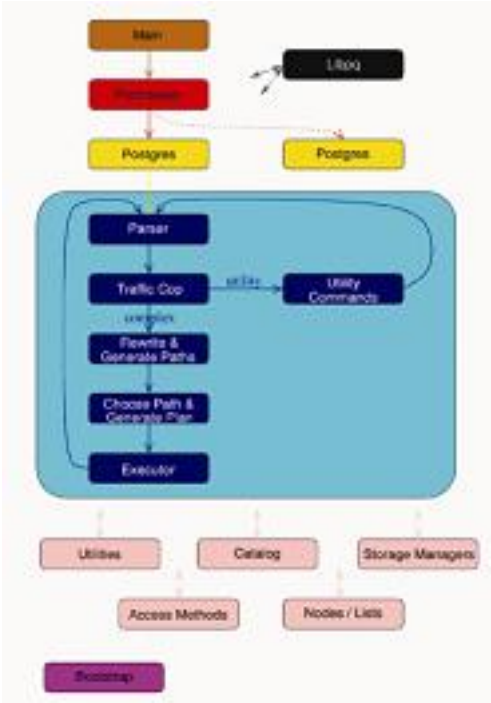

Figure 3.1: Flow of Query through PostgreSQL

of multilingual operators, mainly parser, rewritter, planner, executer is modified.

3.1.2 Implementation choices

In PostgreSQL we have the following choices to implement these operators.

1. Outside-the-server-implementation: This is the quick way to add functionality to Postgres. But the new function will not be executed inside the server address space. Thus Postgres is not aware of new operators.

2. Inside-the-server-implementation: In this the new function will be executed inside the server address space. But in this case also, the optimizer is not aware of the operator.

3. Core implementation: In this implementation optimizer is aware of the new function added.

Core implementation is chosen to implement the MLLexEqual, MLSemEqual and MLLike operators inside the PostgreSQL database engine because this allows optimizer to be aware

of new operators. Thus optimizer can optimize the query involving these operators. This allows the operators to execute inside the server address space.

As an example to see how the operators work consider the following relations: ***Books***(*Title varchar, Author varchar, Category varchar*) consists of English tuples. ***किताब*** (*शीर्षक varchar, लेखक varchar, वर्ग varchar*) consists of Hindi tuples as shown in Figure 3.2.

Books

Title	Author	Category
Engineering Mahematics	R S Grewal	Mathematics
Letters to my daughter	Nehru	History
After Many a Summer	Aldous Huxley	Novel
Intoducing NLP	Sue Knight	NLP
The Age of Innocence	Edith Wharton	Novel
Development as Freedom	Amartya Sen	Economics
Modern Database System	Kim	Database
After the Funeral	Agatha Christie	Novel

किताब

शीर्षक	लेखक	वर्ग
आज का अतीत	भीष्म साहनी	आत्मकथा
गबन	प्रेमचंद	उपन्यास
सान्ख्यिकि किताब	महालिनोविस	सान्ख्यिकि
अंतिम अरण्य	निर्मल वर्मा	उपन्यास
शहरयार	कमलेश	कविता
मेरी कहानी	नेहरू	आत्मकथा
भारत एक खोज	नेहरू	इतिहास
गणित	आर्यभट	गणित

Figure 3.2: Books and किताब relations

Chapter 4

MLLike Operator in PostgreSQL

4.1 MLLike Operator in PostgreSQL

MLLike operator is used for *phonemic regular expression* matching in user specified languages.

4.1.1 Logical Design

Consider the relation ***प्रतिरूप*** (*नियमित-व्यंजक varchar*), which has the regular expression in English and Hindi as shown in Figure 4.1. In order to do MLLike-Join of this relation with Books relation, following query is given:

प्रतिरूप

नियमित-व्यंजक
क%
न%
B%
म%
स%
भ%

Figure 4.1: प्रतिरूप relation

```
Select
    Title, Author, Category,
from
    Books, प्रतिरूप
where
    Books.Author
    InLanguages {English, Hindi}
    MLLike
    प्रतिरूप. नियमित- व्यंजक
    InLanguages {English, Hindi}
```

There is only one choice to execute this query. Do the nested loop join of Books and प्रतिरूप relation. For each tuple of Books check whether it matches the regular expression in phonemic space. Note that regular expression is first converted into phonemic space. So it may happen that a regualar expression cannot be converted into phonemic space. With todays database engine, we do not know any way to create the index for LIKE or MLLIKE operator.

Chapter 5

MLLexEqual Operator in PostgreSQL

5.1 MLLexEqual Operator in PostgreSQL

5.1.1 Logical Design

For the query involving MLLexEqual operator, the column involved in MLLexEqual join has to be first converted into the phoneme space. For example, for the query like

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब
where
    Author InLanguages {English, Hindi}
    MLLexEqual
    लेखक InLanguages {English, Hindi}
    Withthreshold 0.5
```

The *Author* column and *लेखक* column has to be converted into its phonemic equivalents. This can be done in the following ways:-

Global Dictionary approach

Maintain one relation say ***Phoneme_Equl*** (*name varchar, phoneme_Equivalent varchar*) with name as the primary key. This relation contains all the words and their equivalent phoneme strings. When the user gives the above query it is equivalently converted into:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब, phoneme_Equl as A,
    phoneme_Equl as B
where
    A.phoneme_Equivalent
    InLanguages{English, Hindi}
    MLLexEqual
    B.phoneme_Equivalent
    InLanguages{English, Hindi}
    Withthreshold 0.5
    and Author = A.name
    and लेखक = B.name
```

In this approach phoneme equivalent for each column is not required to be maintained explicitly. It saves a lot of space. But the time taken for the query is almost three-fold as the number of joins are increased from 1 to 3. In general if the query has n MLLexEqual joins, then the converted query will have 3n number of joins. At the cost of running time one can perform better in space dimension.

Local Column approach

For each column involved in the MLLexEqual join maintain its phoneme equivalent column in the table itself. Thus when the first time user gives the query, existence of phoneme column is checked for the column involved in the query. If it does not exist,

phoneme column is created first and populated with their phoneme strings. For example, Books.phoneme_Author column can be created for the Author column of Books table. Now user query can be modified as:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब
where
    Books.phoneme_Author
    InLanguages{English, Hindi}
    MLLexEqual
    किताब.phoneme_ लेखक
    InLanguages{English, Hindi}
    Withthreshold 0.5
```

Thus whenever the user inserts or updates the tuples in the relation, corresponding value in the phoneme column has to be updated. This is achieved with the help of a trigger. The drawback of this approach is that phoneme equivalent has to be generated for each word when it is inserted or modified even if phoneme string for this word has been generated earlier. This approach compromises space but performs better in time dimension.

Hybrid approach

This approach is a combination of the first two approaches. For each column involved in the MLLexEqual join its phoneme equivalent column is maintained in the table itself. Thus when the first time user gives the MLLexEqual query, existence of the phoneme column is checked for the column involved in the query. In this approach ***Phoneme_Equl***(*name varchar, phoneme_Equivalent*) table with name as the primary key is also maintained. If the phoneme column does not exist, phoneme column is created first and populated with

their phoneme strings by obtaining them from the ***Phoneme_Equl*** relation. Thus for the same word, phoneme equivalent is not generated each time. If the phoneme equivalent of some word does not exist in the ***Phoneme_Equl*** relation, phoneme equivalent is created first, inserted into the ***Phoneme_Equl*** relation and the phoneme column of the table. Now user query can be modified as in Local approach discussed earlier.

This approach does not require any extra join as compared to first approach. Whenever the user inserts or modifies the tuples in the relation, corresponding values in the phoneme column has to be updated. This is achieved with the help of triggers. This is done by obtaining the new phoneme string of the word from the ***Phoneme_Equl*** relation if it exists. If it does not exist in the ***Phoneme_Equl*** relation, its phoneme equivalent is created and inserted into the ***Phoneme_Equl*** relation and used. Thus it does not require generating the phoneme for the same word again and again as in second approach. In this approach space required is more than the second approach but the time required for generating the phoneme equivalent of the word is saved. This approach compromises the space but outperforms in terms of running time as shown by experimental results.

For the MLLexEqual operator implementation, third strategy is chosen because of the reasons given above. In this implementation in order to give a MLLexEqual query, user has to create the phoneme column first. For that a new SQL create statement is added whose syntax is:

```
create MLColumn on
<Tablename>('<columnname>')
```

Here columnname is the name of column for which phoneme column is to be created. For example to create the ML column for column Author of relation Books user has to give the following MLColumn creation command.

```
create MLColumn on
Books('Author')
```

When the user gives the MLColumn creation query, first verify the table and column for existence. If they exist, TTP(Text-to-phoneme) converter is used to generate the phoneme. TTP obtained from Dhvani software [17] is integrated inside the Postgres excuter component. So there is no extra overhead to call the TTP function.

Separate SQL statement for the phoneme column creation is provided so that it does not affect the performance of MLLexEqual (or MLLike). Because the MLLexEqual (or MLLike) query does not create the phoneme column. If MLLexEqual (or MLLike) query creates the phoneme column then due to column creation it will slow down the performance of first MLLexEqual (or MLLike) query. Due to presence of phoneme column, usual inserts and updates cannot be performed by the user as the phoneme string cannot be supplied. The position of phoneme column in the table is hidden from the user, so that usual insers/updates is not effected.

In order to solve this problem, there is a flag in PostgreSQL 'attisdropped'. If this flag for a column is set to true Postgres assumes that column is logically dropped by the user. When the phoneme column is created, 'attisdropped' is set to true so that phoneme column is logically dropped. While performing normal inserts/updates, user need not worry about the phoneme column. Phoneme column value is updated by running a trigger on inserts/updates. When user gives the MLLexEqual (or MLLike) query, 'attisdropped' flag is set to false so that Postgres knows that phoneme column logically exists. Once the MLLexEqual (or MLLike) query completes phoneme column is logically dropped again.

But when the user gives the **clusterdb** or **alter table alter column type** SQL statement, Postgres physically removes all the logically dropped columns. At that point the phoneme column will also be removed. To solve this problem, while executing the **clusterdb** or **alter table alter column type** command, the column to be removed is checked to see if it is the phoneme column. If it is phoneme column, it is not removed physically.

5.1.2 GiST

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template 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.

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

1. 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.

2. 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.

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

4. 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.

5. 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.

6. 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.

7. same:- Returns true if two entries are identical, false otherwise.

5.1.3 Index

GiST support is provided in PostgreSQL for the implementation of balanced index. Following choices are there:-

1. M-Tree [9]:- M-Tree can be used for range query and k-nearest neighbor query. But for large dimension the selectivity of M-Tree is very poor.

2. Slim-Tree [11]:- Slim-tree uses the slim down algorithm which leads to better tree, decreasing the absolute fat factor. It also makes use of faster splitting algorithm based on the minimal spanning tree. It makes use of chooseSubtree algorithm for the slim-tree (minoccup) which leads to tighter trees, thus have fewer disk pages, and faster retrievals.

3. DF-Tree [12]:- DF-Tree uses the multiple global representatives. With the multiple global representatives its pruning with respect to number of distance computation is very high. It is less efficient than the slim-tree in number of disk access.

With the above available choices, only M-tree can be implemented currently with the current seven-functions API provided by GiST. Slim-tree cannot be implemented because it requires slim-down algorithm once the tree is built. With current GiST API slim-down algorithm cannot be run. We are trying to add a 'Slim-Down' function to GiST API, so that tree can be slim down once it is built. DF-tree cannot be implemented with current GiST because it does not support multiple representatives of the node. As noted earlier, the number of distance computations for the slim-tree is more than the DF-tree, but the distance computation is cheaper than the cost involved in the disk access. The number of disk accesses of slim-tree is much smaller than the DF-tree.

Index creation

The SQL statement for creation of multilingual index is:

```
create MLIndex on
<Tablename>('<columnname>')
```

Index will be created on the phoneme column. For example to create the index on column लेखक of relation किताब user has to give the following MLIndex creation command.

```
create MLIndex on
किताब(' लेखक')
```

When user gives the MLIndex creation query, the existence of phoneme column is checked first. If it does not exist, 'First create the phoneme column' error is thrown. If it exists 'attisdropped' flag is set to false for the phoneme column and index on phoneme column is created.

Chapter 6

MLSemEqual Operator

6.1 MLSemEqual Operator

MLSemEqual operator is available in two sub forms - MLSemEqualall and MLSemEqual. MLSemEqualall provides the ontology matching for which it descends through whole subtree under the given word in the wordnet. MLSemEqual provides semantic matching with the use of crosslink tables.

6.1.1 MLSemEqualall Operator

Logical Design

For the query like:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब
where
    Books.category InLanguages {English, Hindi}
    MLSemEqualall
    किताब. वर्ग InLanguages {English, Hindi}
```

There are two choices to execute this query:

1. Recursive closure:- Do the nested loop join of Books and किताब table. For each tuple of Books calculate the closure of किताब.वर्ग from the English and Hindi wordnets. Check whether the Books.category is in the closure calculated above. If it is in the closure output this tuple in the join. This can be done with todays database engines by rewriting the query like:

```
Select
     Title, Author, Category,
     शीर्षक, लेखक, वर्ग
from
     Books, किताब
where
    Books.Category in
    ((Select child
      from english
      connect by prior child = parent
      start with parent in
     (Select equivalent
      from crosslink
      where word =  किताब. वर्ग
     )
   )
union

     (Select child
      from hindi
      connect by prior child = parent
      start with parent in
     (Select equivalent
      from crosslink
```

```
    where word =  किताब. वर्ग
   )
  )
 )
```

In the above query the semantic equivalent of किताब.वर्ग is calculated in the languages English and Hindi from the crosslink table. For this following query is used:

```
Select equivalent
from crosslink
where word =  किताब. वर्ग
```

Once the semantic equivalent from crosslink table is calculated, the closure of this equivalent tuple has to be calculated from the wordnet. In order to calculate the closure from the English wordnet following query is used:

```
Select child
from english
connect by prior child = parent
start with parent in
(Select equivalent
 from crosslink
 where word =  किताब. वर्ग
)
```

In a similar way closure from the Hindi wordnet can be calculated. We then compute the union of the two closure outputs and check whether Books.Category is in the closure. If it is in the closure then join these two tuples.

2. Hopi Index [10]:- Perform the nested loop join of Books and किताब table. For each tuple of Books checks from the hopi index table whether there is path from किताब.वर्ग to Books.Category in the given wordnet. If there is path output this tuple in the join. This can be done with current database engines by rewriting the query as:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
     Books, किताब
where
    exists
        (Select
            english_hopi_lin.element
        from
            english_hopi_lin,
            english_hopi_lout,
            crosslink
        where
            english_hopi_lin.node
                  = Books.Category
            and english_hopi_lin.element
                  = english_hopi_lout.element
            and english_hopi_lout.node
                  = crosslink.equivalent
            and crosslink.word
                  =  किताब. वर्ग
        )
    or exits
        (Select
            hindi_hopi_lin.element
        from
            hindi_hopi_lin,
            hindi_hopi_lout,
            crosslink
        where
```

```
            hindi_hopi_lin.node
                  = Books.Category
            and hindi_hopi_lin.element
                  = hindi_hopi_lout.element
            and hindi_hopi_lout.node
                  = crosslink.equivalent
            and crosslink.word
                  = किताब. वर्ग
      )
```

The inner query checks whether the Books.Category is present in the closure of किताब.वर्ग. For example, to check whether the Books.Category is in the closure of किताब.वर्ग of English wordnet following query is used:

```
Select
    english_hopi_lin.element
from
    english_hopi_lin,
    english_hopi_lout,
    crosslink
where
    english_hopi_lin.node
              = Books.Category
    and english_hopi_lin.element
              = english_hopi_lout.element
    and english_hopi_lout.node
              = crosslink.equivalent
    and crosslink.word
              = किताब. वर्ग
```

Implementation Issue of MLSemEqual Operator

To support the MLSemEqualall query crosslink table, Wordnet table and hopi table has to be stored for each language. To get the information of crosslink, wordnet and hopi table, the relation ***pg_multilingual*** is designed whose format is:

```
 Column      Type     Modifiers

language    name     not null
wordnet     name     not null
hasindex    boolean  not null
lin         name     not null
lout        name     not null
crosslink   name     not null
```

When a MLSemEqualall query is encountered, the existence of wordnet and crosslink table for each corresponding target language is checked. If it is defined, the MLSemEqualall query is rewritten based on the hasindex value for that language. If the language has hopi index, query is rewritten in the second way else in the first way.

6.1.2 MLSemEqual Operator

Logical Design

For the query like:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब
where
    Books.Category
    InLanguages {English, Hindi}
```

```
MLSemEqual
किताब. वर्ग
InLanguages {English, Hindi}
```

There is only one choice to execute this query. Perform a nested loop join of Books and किताब table. For each tuple of Books, calculate the semantic equivalent of किताब.वर्ग from the crosslink table. If they are semantically equal, output this tuple in the join. This can be done with current database engines by rewriting the query as:

```
Select
    Title, Author, Category,
    शीर्षक, लेखक, वर्ग
from
    Books, किताब
where
    exists
    (Select
        equivalent
    from
        crosslink
    where
        word =  किताब. वर्ग
        and equivalent = Books.Category
    )
```

In the above query, the inner query is calculating whether the Books.Category is semantically equal to किताब.वर्ग. If it is, the tuple is output. For the implementation of MLSemEqual only crosslink table is needed for each language. This information is retrieved from the ***pg_multilingual*** table defined earlier.

Chapter 7

Experimental Result

7.1 Experimental Result

Experiment setup: Dual core AMD Opteron processor 2.40GHz having 4GB of main memory and Linux Redhat Enterprise IV O/S. Postgres version 8.1.2 is used.

In order to check the performance of MLLike, MLLexEqual and MLSemEqual query following tables are used: ***MLBooks*** (*Title varchar, Author varchar, Category varchar*) consists of 100 unique tuples. By repeating, we created 10,000 to 100,000 strings of both English and Hindi data, ***Books***(*Title varchar, Author varchar, Category varchar*) with 10,000 to 100,000 strings of English data, ***किताब*** (*शीर्षक varchar, लेखक varchar, वर्ग varchar*) with 10,000 to 100,000 strings of Hindi data. Hopi index for both the languages is created.

7.1.1 Mulitilingual column creation experiment

The *Text-to-Phoneme*[TTP] converter is obtained from Dhvani software. Experiment is run with both the local approach and hybrid approach. The performance for phoneme string creation for 10,000 to 100,000 strings is shown in Figure 7.1. The time taken is roughly linear with respect to the number of strings for both the approaches. But the time taken with dictionary is less than the time taken without dictionary.

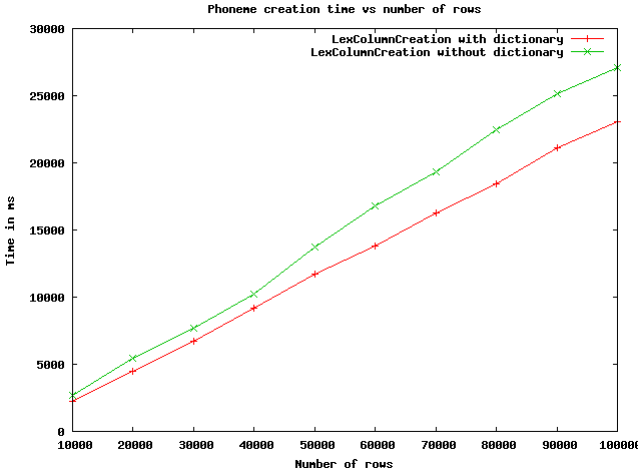

Figure 7.1: Phoneme creation time

7.1.2 MLLike experiments

To check the performance of MLLike involving a constant,following query has been given:

```
Select
    Title, Author, Category
from
    MLBooks
where
    Author InLanguages {English, Hindi}
    MLLike
    ' न%'
```

The performance of all the three data sets is found to be nearly identical shown in Figure 7.2.

To check the performance of MLLike join the following query is used: self join of ***MLBooks*** having both English and Hindi data is done, and join of ***Books*** having English data with ***किताब*** having Hindi data is performed. The following query is given:

```
Select
    t1.Title, t1.Author, t1.Category,
```

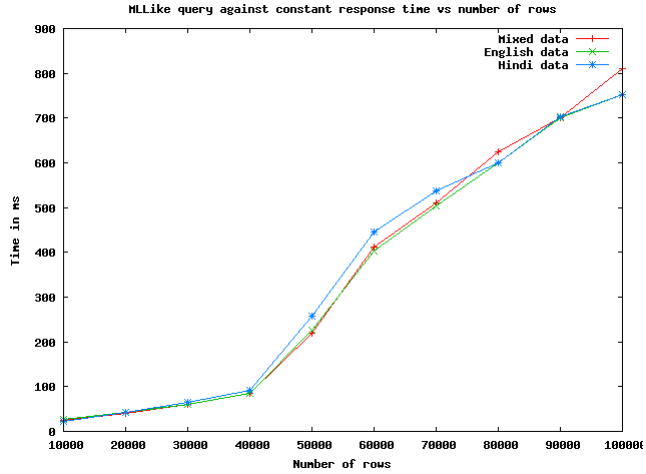

Figure 7.2: Performance of MLLike query for the constant

```
    t2.Title, t2.Author, t2.Category
from
    MLBooks as t1, MLBooks as t2
where
    t1.Author InLanguages {English, Hindi}
    MLLike
    t2.Author InLanguages {English, Hindi}
```

The performance of both data sets is almost identical as shown in Figure 7.3.

7.1.3 MLLexEqual experiments

Next, the MTree index for the 10,000 to 100,000 phoneme strings is created. In this case also, the time taken for the index creation is linear with respect to the number of phoneme strings as shown in Figure 7.4.

For the MLLexEqual query, the ***MLBooks*** relation is used. Following MLLexEqual query involving a constant is run:

```
Select
    Title, Author, Category
```

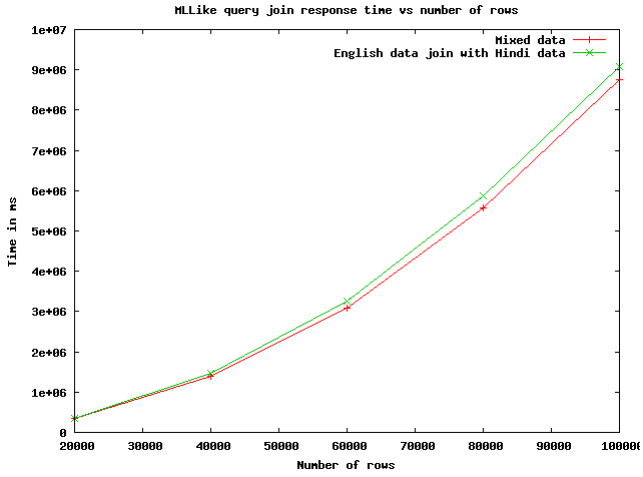

Figure 7.3: Performance of MLLike query join

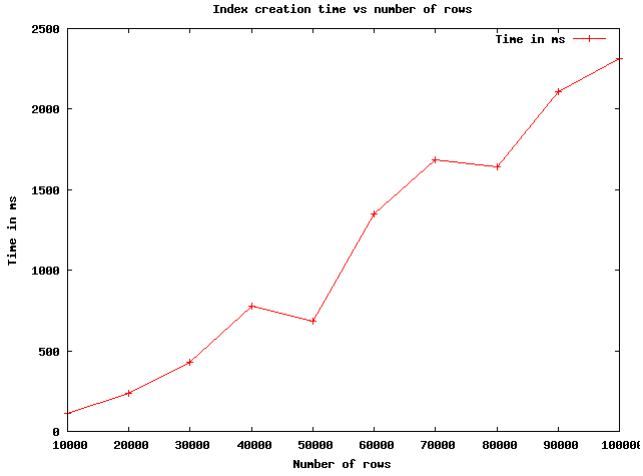

Figure 7.4: Index creation time for the phoneme string

```
from
    MLBooks
where
    Author InLanguages {English, Hindi}
    MLLexEqual
    'Nehru'
    Withthreshold 0
```

Performance of Sequential scan, Index scan and Bitmap index is checked for this query. All are linear in terms of number of input tuples. The performance of Bitmap index and Index scan is far better than Sequential scan. Performance of Bitmap index scan and Index scan is almost equal as shown in Figure 7.5.

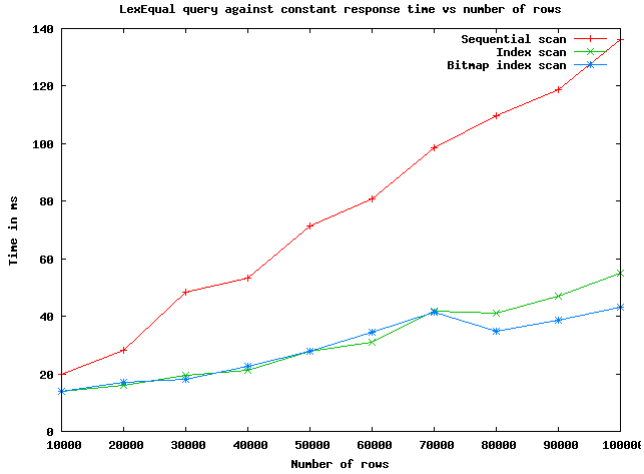

Figure 7.5: Performance of MLLexEqual query for the constant

To check the performance of MLLexEqual join ***MLBooks*** is used. The following query is run to check MLLexEqual join performance:

```
Select
    t1.Title, t1.Author, t1.Category,
    t2.Title, t2.Author, t2.Category
from
```

```
    MLBooks as t1, MLBooks as t2
where
    t1.Author InLanguages {English, Hindi}
    MLLexEqual
    t2.Author InLanguages {English, Hindi}
    Withthreshold 0
```

Performance of nested loop join with sequential scan, nested loop join with index scan and nested loop join with bitmap index scan is evaluated for this query. The performance of Bitmap index scan and Index scan is far better than Sequential scan. Performance of Bitmap index scan and Index scan is almost equal as shown in Figure 7.6.

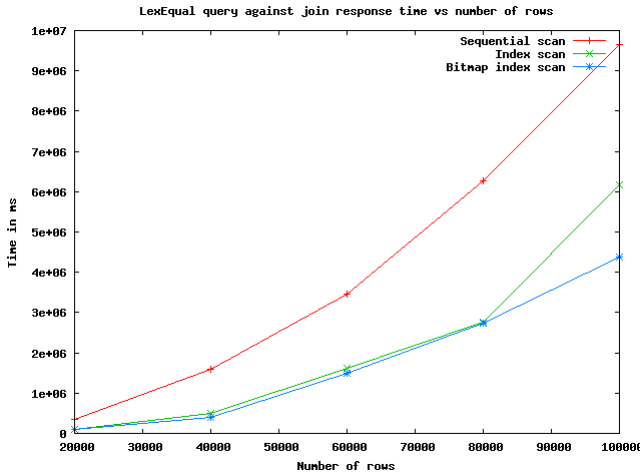

Figure 7.6: Performance of MLLexEqual query for the join

7.1.4 MLSemEqual experiments

Following query is used to check the performance of MLSemEqualAll involving a constant.

```
Select
    Title, Author, Category
from
    MLBooks
```

```
where
    Category InLanguages {English, Hindi}
    MLSemEqualall
    'science'
```

Similar query is given for the ***Books*** and ***किताब*** relations. The performance of mixed data (English and Hindi) is almost same as Hindi data. The performance of Hindi data and mixed data is found to be poorer than English data as shown in Figure 7.8. Reason for this behavior is, for the word 'science' the number of semantic equal words from the crosslink table in Hindi is 9 times more as shown in Figure 7.7.

Crosslink

Word	Equivalent
science	कौशल
science	विषय
science	ज्ञान
science	तकनीक
science	अध्ययन
science	क्षेत्र
science	विज्ञान
science	प्रवृति-विज्ञान
science	प्रक्रिया
science	science

Figure 7.7: Semantic equal words of 'science'

Time required by the recursive query for the MLSemEqualall is also checked, its performance is found to be very poor compare to the hopi index query. As the time required to calculate the closure by recursion is very high as shown in Figure 7.9. The compiled plan diagram (obtained from Picasso) shows that recursive query is almost 20 times costlier

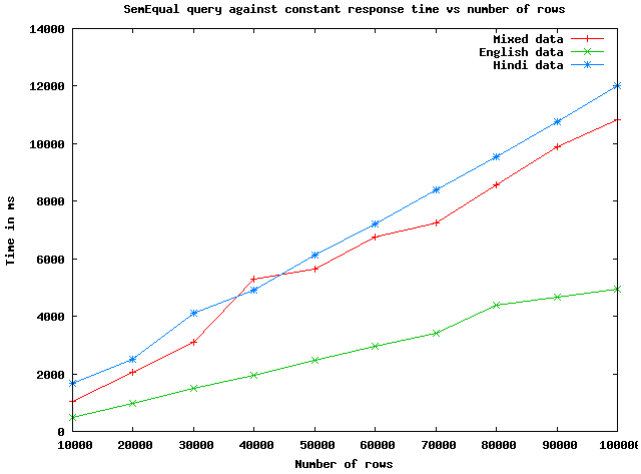

Figure 7.8: Performance of MLSemEqualall query for the constant

than the hopi query.

Also, the time required to join the two tables with MLSemEqualall operator is high even with the hopi index defined. Since the subquery is executed for each row when the join is performed, which is quite costly, the performance cannot be improved further. This is shown in Figure 7.10. The compile cost diagram is obtained from Picasso.

Following query is used to check the performance of MLSemEqualAll involving a constant.

```
Select
    Title, Author, Category
from
    MLBooks
where
    Category InLanguages {English, Hindi}
    MLSemEqual
    'science'
```

Similar query is given for the ***Books*** and ***किताब*** relations. The performance of all the three data set are almost the same as shown in Figure 7.11.

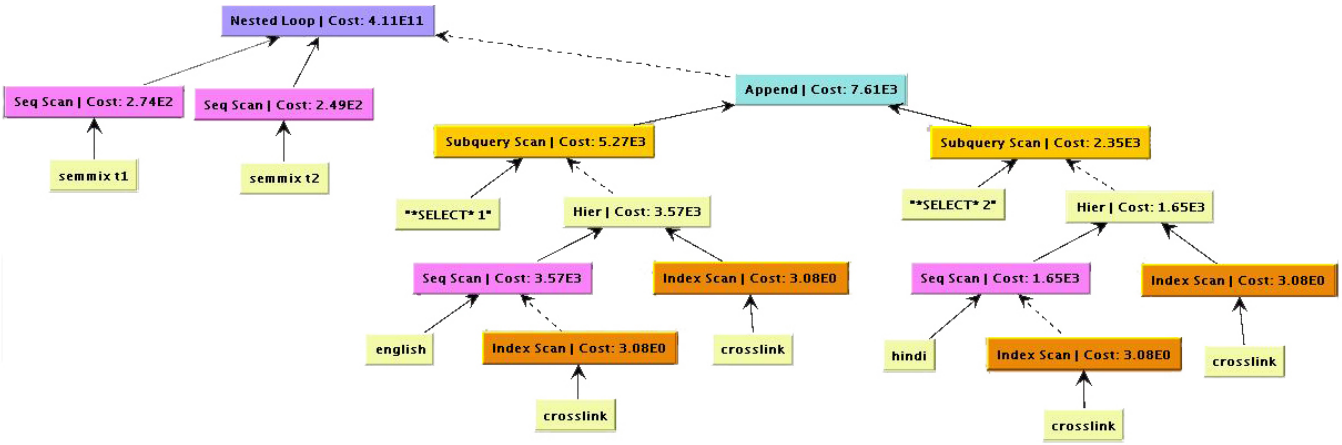

Figure 7.9: Plan diagram for MLSemEqualall query rewritten using recursion

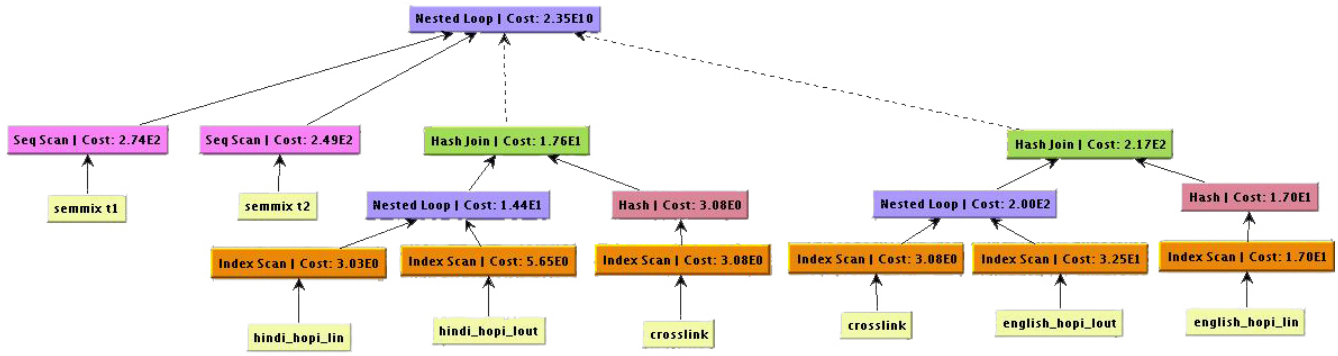

Figure 7.10: Plan diagram for MLSemEqualall query rewritten using hopi index

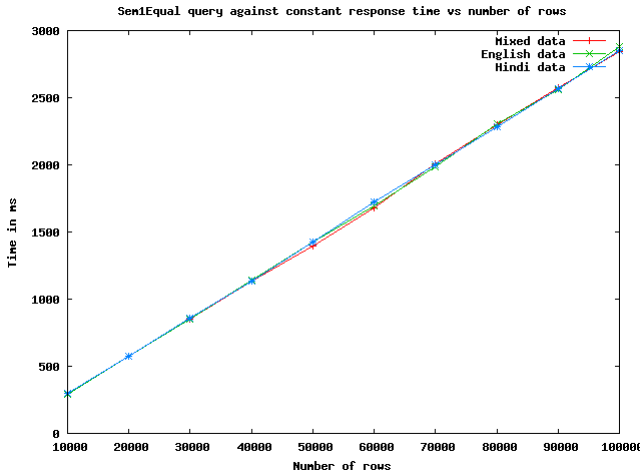

Figure 7.11: Performance of MLSemEqual query for the constant

Chapter 8

Conclusion

8.1 Conclusion

In this project, we have done the persistent implementation of MLLexEqual, MLSemEqual and MLLike operators. Various logical and design issues for the implementation of such operators inside the PostgreSQL database engine have been investigated. Core implementation of these operators is done inside PostgreSQL as it is faster and can be optimized by the optimizer. We investigated the impact of index for MLLexEqual operator. Performance of MLSemEqual has been improved by the addition of hopi index.

Bibliography

[1] A. Kumaran, P. K. Chowdary and J. R. Haritsa. *On Pushing Multilingual Query Operators into Relational Engines*, ICDE 2006.

[2] A. Kumaran and J. R. Haritsa. *MLLexEqual: Supporting Multiscript Matching in Relational Systems*, EDBT 2004.

[3] A. Kumaran and J. R. Haritsa. *MLSemEqual: Multilingual Semantic Matching in Relational Systems*, DASFAA 2005.

[4] A. Kumaran. *MIRA: Multilingual Information Processing on Relational Architecture*, EDBT 2004.

[5] A. Kumaran and J. R. Haritsa. *On the Cost of Multi-Lingualism in Database Systems*, VLDB 2003.

[6] A. Kumaran and J. R. Haritsa. *Multilingual Semantic Matching Operator in SQL*, TR-2004-03, DSL/SERC.

[7] A. Kumaran. *Multilingual Information Processing on Relational Database Architectures*, PhD Thesis, CSA Dec, 2005.

[8] P. Pavan Kumar Chowdary. *MLPostgres: Implementing Multilingual Functionalities inside PostgreSQL Database Engine*, M.E. Thesis, CSA June, 2005.

[9] P. Ciaccia, M. Patella and P. Zezula. *An Efficient Access Method for Similarity Search in Metric Space.* VLDB 1997

[10] E. Cohen, Eran Halperin, Haim Kaplan and Uri Zwick. *Reachability and distance queries via 2-hop labels* SODA 2002.

[11] C. Traina, A. Traina, C. Faloutsos and B. Seeger. *Fast Indexing and Visualization of Metric Data Sets Using Slim-Trees*, KDE 2002.

[12] C. Traina, A. Traina, R. Filho and C. Faloutsos. *How to Improve the Pruning Ability of Dynamic Metric Access Methods*, CIKM 2002.

[13] *PostgreSQL Database System.*
http://www.postgresql.org

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

[15] www.cfilt.iitb.ac.in/wordnet/webhwn/

[16] http://www.postgresql.org/docs/8.1/static/gist.html

[17] http://www.simputer.org/simputer/downloads/software/dhvani