# A Persistent SEMEQUAL Operator

A project report submitted in partial fulfilment of the
requirements for the Degree of

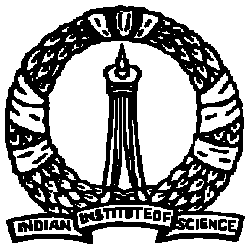
**Master of Engineering**

in

Computer Science and Engineering

by

*Prateem Mandal*




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

JULY 2006

# 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 would also like the thank Dr. A. Kumaran for his help and clarifications. Indeed this project is a final embodiment of his work on cross lingual operators. I would also like to thank all my colleagues who directly or indirectly helped me in completing the project.

# 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 language query operators has been previously dealt with and two cross lingual operators **LexEQUAL**[2] and **SemEQUAL**[3] were introduced. In this work, we focus towards defining SemEQUAL operator, investigating issues related to implementation of the operator inside a relational engine and approaches towards further optimization. Specifically, we define the SemEQUAL operator from an implementation point of view, and present a successful implementation of the SemEQUAL operator inside **PostgreSQL**[17] database system which is **persistent** in nature and **optimized** by the optimizer of PostgreSQL. Also, we investigate **HOPI Index**[10] as a method towards optimization of SemEQUAL operator and address issues related to adapting it to large scale graph structures like **WordNets**[18] and implementing it in a seamless fashion inside PostgreSQL. Our experiments demonstrate that SemEQUAL operator has a reasonable performance when used with constants but is quite expensive for join operations when used without indexes. With indexes however, the performance of join operation improves manifold. This prototype implementation is the first internal implementation of SemEQUAL which addresses all the issues related to its implementation inside a database except for indexes. It is also for the first time that the impact of HOPI indexes has been investigated in relation to SemEQUAL and WordNets.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

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, the 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 is in languages other than English. Currently, multilingual data are 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 of 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 for certain kind of information over all the languages. Currently, this is not possible in most information systems. This requirement for cross lingual queries has been addressed through two operators called **LexEQUAL**[2] and **SemEQUAL**[3]. Consider a hypothetical e-Commerce application, *Books.com*, that sells books across the globe; the **Book** table storing data in multiple languages or a logical view assembled from data sourced off several databases, shown in Figure 1.1, can be a possible schema for viewing data about all the books that

| Author | Title | Price | Category |
|---|---|---|---|
| नेहरू जवाहरलाल | भारत एक खोज | INR 175 | इतिहास |
| Franklin Benjamin | Un Amëricain Autobiographie | € 25.00 | Autobiographie |
| Gilderhus Mark T. | History and Historians | $ 49.95 | Historography |
| Durant Ariel | History of Civilization | $ 148.95 | History |
| Lebrun Francois | L' Histoire De La France | € 19.95 | History |
| Descartes Renē | Les Mëditations Metaphysiques | € 49.00 | Philosophie |
| গৌতম ভাদ্রা | নিক্সোবার্গের ইতিহাস | INR 425 | ইতিহাস |
| തകഴി | ചെമ്മീന് | INR 300 | സാഹിത്യം |

Figure 1.1: Multilingual Books.com

*Books.com* has in its inventory.

In such an environment, a user may wish to ask basically two kinds of cross lingual queries:

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

2. A user might wish to find out all the 'History' books that are available in a certain set of languages (possibly all languages).

For the first type of query the **LexEQUAL**[2] operator is used. This operator takes as input, a name in one language ('Nehru' in English for example), and returns all the *phonemically close* names in a user specified set of languages.

The second type of query presents a different type of problem. Here, one has to find out *semantic* similarity between words. This requires some kind of semantic analysis of the word form. In the currently available SQL functionality one may write a query as shown below.

The result of this query is shown in Figure 1.2. However we know that the entries written

**Simple SQL query**

```
SELECT * from books
WHERE Category = 'History';
```

| Author | Title | Price | Category |
|---|---|---|---|
| नेहरु जवाहरलाल | भारत एक खोज | INR 175 | इतिहास |
| Franklin Benjamin | Un Amëricain Autobiographie | £ 25.00 | Autobiographie |
| Gilderhus Mark T. | History and Historians | $ 49.95 | Historography |
| Durant Ariel | History of Civilization | $ 148.95 | History |
| Lebrun Francois | L' Histoire De La France | € 19.95 | History |
| Descartes Renë | Les Mëditations Metaphysiques | € 49.00 | Philosophie |
| গৌতম ভদ্রা | নিক্কোবার্লের ইতিহাস | INR 425 | ইতিহাস |
| ककषी | ചെമ്മീന് | INR 300 | സാഹിത്യം |

Figure 1.2: Result for simple SQL query

in Hindi and Bengali are also books on 'History'. For this purpose, **SemEQUAL**[3] operator is used. It finds out the words belonging to different languages that are semantically similar to each other. In order to do this, it uses available ontology like **WordNets**[18]. The output of SEMEQUAL query is given in Figure 1.3.

In this work, we will concentrate on the issues related to native, persistent implementation of **SemEQUAL**[3] inside **PostgreSQL**[17] and its optimization. The rest of the paper is organized as follows. Sections 1.1 and 1.2 give a brief introduction about the **PostgreSQL**[17] and the **WordNet**[18]. Section 2 describes **SemEQUAL**[3] operator from an implementation perspective. Section 3 describes in detail the design and implementation of a persistent **SemEQUAL**[3] operator inside **PostgreSQL**[17] database system. Section 4 deals with the investigation that we have done related to efficacy of employing index structures to **SemEQUAL**[3]. In Section 5, we give the experimental results and future work.

| Author | Title | Price | Category |
|---|---|---|---|
| नेहरु जवाहरलाल | भारत एक खोज | INR 175 | इतिहास |
| Franklin Benjamin | Un Amëricain Autobiographie | € 25.00 | Autobiographie |
| Gilderhus Mark T. | History and Historians | $ 49.95 | Historography |
| Durant Ariel | History of Civilization | $ 148.95 | History |
| Lebrun Francois | L' Histoire De La France | € 19.95 | History |
| Descartes Renë | Les Mëditations Metaphysiques | € 49.00 | Philosophie |
| গৌতম ভদ্র | নিম্নোবার্গের ইতিহাস | INR 425 | ইতিহাস |
| തകഴി | ചെമ്മീന് | INR 300 | സാഹിത്യം |

Figure 1.3: Result for SEMEQUAL query

## 1.2 The PostgreSQL Database Management System

**PostgreSQL**[17] is an **Object Relational Database Management System** that is arguably the most advanced database in Open Source domain. It was developed at Computer Science Department of University of California, Berkeley. **POSTGRES** pioneered many concepts that only became available in some commercial databases much later. 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**[17] 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 modifications.

## 1.3 The WORDNET

A **WordNet**[18] is a *semantic lexicon* for a given language. It groups words into set of synonyms called *synsets*, provides short general definitions and records various *semantic* relations between *synonym* sets. It has been created primarily for producing a combination of dictionary and thesaurus that is more intuitively usable, and to support automatic

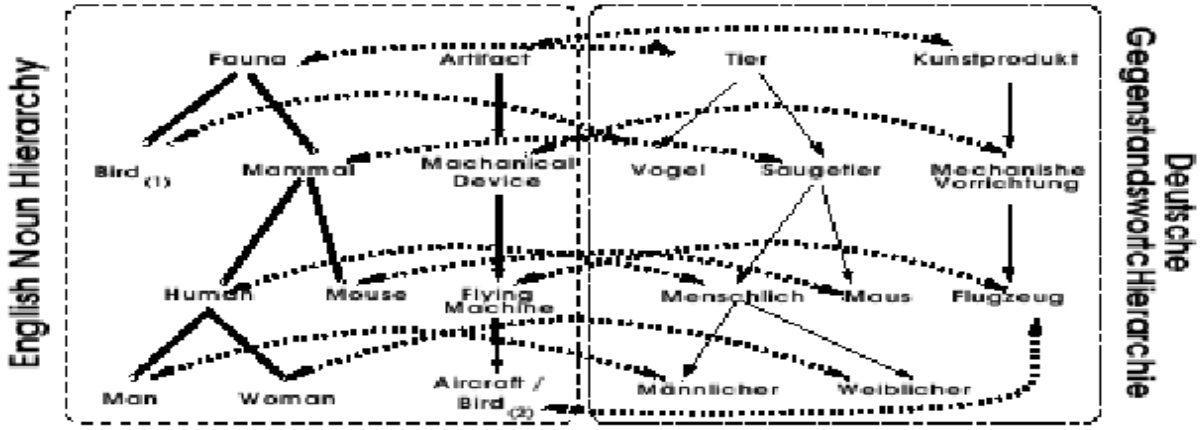



Figure 1.4: Sample Interlinked WordNet Noun Hierarchy

text analysis and artificial intelligence applications. The defining philosophy in the design of **WordNet**[18] is that a *synset* is sufficient to identify a concept for the user. Two words are said to be synonymous or *semantically the same*, if they have the same synset and hence map to the same mental concept. WordNet organizes all relationships between the concepts of a language as a semantic network between synsets. A lexical matrix that maps word forms to word senses constitutes the basis for mapping a word-form to synsets.

**WordNets** also provide a semantic functionality by providing inter WordNet links between WordNets of different languages as shown in Figure 1.4. Using the lexical matrix function that is a part of WordNet linguistic resources, the operands (i.e., multilingual word-forms), may be mapped onto distinct set of synsets associated with languages of the respective operands. This facility of inter-language links will be later used to figure out the semantically equivalent words in the target language. The **WordNet**[18] for English language is currently the most developed one and the related database and software tools have been released under a BSD style license and can be downloaded and used easily. The database can also be browsed online. The WordNet used here is a English WordNet(version 1.5) which has about 110,000 word forms, 80,000 synsets and about 140,000

relationships. The approximate disk size of this particular WordNet is 4MB.

**WordNets**[18] are also available in other languages but are in their infancy and are not as usable as English. **EuroWordNet**[20] is a project for producing WordNets for several European languages and to link them together; these are not freely available however. The **GlobalWordNet**[21] project attempts to coordinate the production and linking of WordNets of all languages. A WordNet for Hindi has also been initiated [19] and is in its early stages of development.

# Chapter 2

# The SEMEQUAL Operator

The **SemEQUAL**[3] operator is a *semantic* matching functionality that is used in cases where one has to determine if two words are semantically equivalent. For example, let us take the example of *Books.com*. Suppose, a user wants to find out all the books whose *Category* is semantically equivalent to 'History' in a set of languages. In today's databases if you give a query with (**Category = 'History'**) selection condition, only those books whose category is 'History' in English will be returned despite the fact that the catalog also contains history books in Hindi, Tamil and French. A multilingual user would be better served if history books in the given language set were returned. If **SemEQUAL**[3] operator is used for the same query, then it will return all the books on history in the given language set, available in catalog. For an input given in Figure 2.1, a query using **SemEQUAL**[3] and the consequent result is given in Figure 2.2.

It should be noted however that the **SemEQUAL**[3] operator shown here is generalized to return not only the tuples that are equivalent in meaning, but also with respect to semantic *generalizations* and *specializations*, as evident by the last three tuples that are reported in the output.

In order to determine the semantic equivalence of word-forms across languages in **SemEQUAL**[3] operator, we rely on the **WordNets**[18]. The basic idea is to use the **WordNets**[18] to

| Author | Author_FN | Title | Price | Category |
|---|---|---|---|---|
| descartes | René | Les Méditations Metaphysiques | €49.00 | Philosophie |
| [illegible] | [illegible] | [illegible] | INR 250 | [illegible] |
| [illegible] | [illegible] | [illegible] | ¥475000 | 伝記 |
| Lebrun | François | L'Histoire De La France | €19.95 | Histoire |
| Durant | Will/Ariel | History of civilization | $149.95 | History |
| [illegible] | [illegible] | [illegible] | INR 175 | [illegible] |
| Franklin | Benjamin | Un Américain Autobiographie | €25.00 | Autobiographie |
| Gilderhus | Mark T. | History and Historians | $49.95 | Historiography |
| [illegible] | [illegible] | [illegible] | INR 250 | [illegible] |

Figure 2.1: Result for SEMEQUAL query

```
SELECT Author,Title,Category FROM Books
WHERE Category SemEQUAL ALL 'History'
InLanguages {English, French, Tamil}
```

| Author | Title | Category |
|---|---|---|
| Durant | History of Civilization | History |
| Lebrun | L'Histoire De La France | Histoire |
| [illegible] | [illegible] | [illegible] |
| Franklin | Un Américain Autobiographie | Autobiographie |
| Gilderhus | History and Historians | Historiography |
| [illegible] | [illegible] | [illegible] |

Figure 2.2: Multilingual Semantic Selection

find out the inter-language and intra-language semantic equivalence to match words. The methodology that we use to implement **SemEQUAL**[3] is shown in Algorithm SemEQUAL. As one can see, **SemEQUAL**[3] operation has three important steps:

**Algorithm 1** Algorithm SemEQUAL

**Input:** Data String $w$
Query String $q$
Language Set $L$
**Output:** TRUE or FALSE

1. for each $l \in L$
   (a) $W \leftarrow$ WordNet of $l$.
   (b) $C \leftarrow$ Closure $(q_l, W)$.
   (c) $TC \leftarrow TC \cap C$.
2. if $w \in TC$
   return TRUE.
3. else
   return FALSE.

1. Finding out per language equivalent word of the RHS operand through inter-language links.
2. Computation of closure of synsets corresponding to the equivalent of the RHS operand in the specified languages.
3. Testing if the LHS component belongs in the closure.

We define 'a' SemEQUAL 'b' INLANGUAGESx,y as true iff 'a' belongs to the union of closures of word form 'b' in wordnet of languages 'x' and 'y'. We will continue with the above definition of **SemEQUAL**[3] for the rest of the paper. The second step involving computation of closure is a computation intensive process and the most costly and the most crucial of all the steps in computation of **SemEQUAL**[3]. Note that the SemEQUAL operation according to the above definition is not commutative. This is because one word-form may belong to the closure of another but the reverse may not be true.

Also note that if the inter WordNet links are transitive then the SemEQUAL operator is transitive.

# Chapter 3

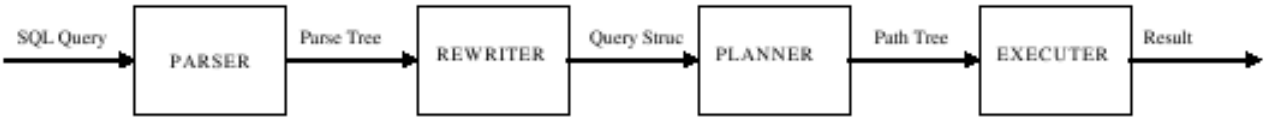
# SEMEQUAL in PostgreSQL

## 3.1 Logical Design Process and Implementation Challenges

As a platform for implementing **SemEQUAL**[3], PostgreSQL was chosen. It was a natural choice as **PostgreSQL**[17] is the most complete database system available in open source world and hence all the issues related to SemEQUAL 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.

At the outset, the primary goal of the project was to implement a **SemEQUAL** functionality inside **PostgreSQL** that can scale for any sized **WordNet** or dataset used. That is, if the size of the **WordNet** or total size of all the **WordNets** of all the languages exceed the main memory size, the closure computation operation should continue seamlessly. A related objective was to implement the functionality in a manner that enmeshed into the rest of database modules. This basically means that all the database modules should be able to interact and do their job if a query involving **SemEQUAL** comes through. That is, the *query rewriter* should still be able to rewrite queries, the *query optimizer* should still be able to optimize the query, all the sanity checks could still be run on the query

Figure 3.1: Flow of Query through PostgreSQL




structure, it should still be possible to pull up sub-queries where ever the opportunity presents itself, all the new data structures introduced should be recognizable to rest of the database modules and they should know how to deal with them; for example, printing function for debugging, data structure tree copying functions, *explain* modules etc. should not be broken. This is important if the implementation has to be kept bug free and practically usable.

There were some daunting challenges in implementing SemEQUAL inside PostgreSQL.

1. There are many details in SemEQUAL implementation that can be chosen from a set of choices. These choices will effect the complexity of final implementation considerably.

2. PostgreSQL doesn't have support for computing closure. This is partly because of the structure of PostgreSQL implementation which is based on recursive invocation of sub-queries.

3. There is not enough available literature on various aspects of closure computation and optimization both theoretical and implementation wise.

4. Though WordNets are reportedly DAGs[16] i.e. Directed Acyclic Graphs[16], on analysis of actual data they were in fact found to have cycles. Thus any implementation has to handle cycles.

5. Finally whatever method is used for implementation, it should be persistent in

nature i.e. it should be able to handle data (both input and data produced during computation) beyond the capacity of main memory.

All these challenges and constraints pose a difficult problem if reliable implementation is to be secured. The first design decision was with regards to the choice of schema used for WordNet table. The WordNet used here for experiments is an English WordNet (Version 1.5), totaling about 110,000 word forms and 80,000 synsets and about 140,000 relationships between them. The fan out from any given node in this particular WordNet is 16 for more than 90% of cases. Thus, at least two possible schemas are possible. One was the simple (parent, child) schema which we will from now on call **PC** and the other was (parent, child1, ..., child16) schema which we will from now on call **PC16**. The PC16 schema does offer some advantages over PC schema like it would have less number of records then the simpler PC schema. However, it is much harder to process than PC schema while finding closure. PC schema on the other hand, could be used to compute closure by using recursive queries in standardized, conventional way if support is available. Thus, PC schema was chosen for representation of WordNet data in the database.

The second design decision was regarding the interpretation of query by the DBMS and the format and content of the meta data. As is clear from the query syntax for SemEQUAL, the information provided by the query includes the name of the languages besides the two operators. The actual execution of the query however requires the crosslink tables between languages and the wordnets of the languages or the HOPI index, in case they were available, besides the operators. Thus it was decided that the language name will be used to resolve the names of the wordnet table and the HOPI index tables, while a single crosslink file is maintained for all the languages. All the other information regarding the schema of the tables etc are predetermined. The predetermined data is hard coded in the implementation whereas the rest of the meta data is stored in the catalog table named **pg_multilingual**. All the data used for computation of SemEQUAL i.e. wordnet, crosslink table etc. are stored in separate **multilingual** schema.

The next design decision was with regards to the implementation strategy for closure. It was decided to make the least complicated and least intrusive implementation decisions so as to achieve above mentioned goals. Emphasis was laid on reusing as much of the existing infrastructure as possible and limit the modification of functions to an extent where they are just an extension of the original so that cross dependencies are not broken. Very few new data structures were added and mostly the existing data structures were modified. This made the extentions of existing copy and traversal functions easy or non necessary. No existing function definitions were changed and although a lot of new functions were added they mostly called each other and only a few were called by existing functions.

## 3.2 Persistent SEMEQUAL Implementation

PostgreSQL database system, being a full fledged system, includes many kind of functionalities. From the input of a SQL query to the output of results, functions implementing these functionalities take in the query as input and try to apply their transformations. While implementing SemEQUAL or any new functionality for that matter, one has to be careful that the data structures introduced or modified, still works with rest of the functions. PostgreSQL has a structure, where sub queries are recursively invoked while computing the query. This intrinsic assumption here is that the query is like a tree with sub queries like subtrees. While this brings a logical clarity to various implementation details, this assumption that the query is a tree structure runs counter to the idea of recursive queries. The effects of this assumption are visible in all the modules of PostgreSQL. In Figure 3.1 we can see the path of a query through the PostgreSQL database system.

The *parser* stage parses the raw SQL query into a parse tree. This is *dumb parsing*, in the sense that there are only syntactic checks done at this stage. This stage converts the raw query string into a *parse tree* structure. The changes required in this stage are related to supporting the SemEQUAL syntax and converting it into a meaningful structure. The optimal design of the structure depends on its use in later stages and future extension

possibilities. Basically, all relevant information needs to be retained in a hierarchical manner so that later stages can see information that they need.

The next stage which has been named here as *rewriter*, converts the *parse tree* structure into a *query tree* structure. This is the stage where many other checks are done and query parameters are classified, located and linked with each other. For example, it is checked if the tables mentioned actually exist, and if so then the *rel cache* is updated and the disk identifier of the table is found and stored in a special data structure for relations. Then, the column names are searched through the tables and their corresponding relation structures are linked to them. Similar operations are done for conditions as well. The major task here is that appropriate structure is prepared for the new kind of parse tree that is now generated by the parser for SemEQUAL. One of the major operations here is to see that the tree structure of a normal query is changed to a graph structure for the recursive SemEQUAL query. There are several rewriting tasks also that are carried out in this stage. We make sure that in case of SemEQUAL, where not necessary they are safely skipped. Another important task done during this stage is to complete the query in the sense that the language names are looked up in the catalog tables and the corresponding wordnet tables are identified and inserted into the query structure. Further if the language has index support, then the query structure is modified recursively to incorporate the index into the query. In case of conflicts or inconsistent data, appropriate error is returned.

The last stage is the *executor* stage. It follows the instructions in the *path tree* to the letter and retrieves tuples and stores them in **tuple store** as intermediate results. At this stage, we had to extend the executor so that whenever it sees the sub query related to SemEQUAL, it calls it repeatedly till the whole operation is done. **Tuple store** is a place where intermediate results are stored. It has a predefined memory budget and if it gets more tuples to store than its budget would allow, it stores them in flat disk files. Thus, we do not need to do anything special to make this whole operation disk persistent as PostgreSQL already deals with this problem. If we can ensure that we use

the atomic operations and functions repeatedly to execute SemEQUAL query then we are sure that it is a persistent implementation, as each of its constituent functions are persistent. The only remaining issue is existence of cycles. This is handled by adding an extra condition that if tuple store already has a tuple which is being entered then backtrack to last unfinished path.

# Chapter 4

# Optimizing SEMEQUAL through Index

Though the current implementation of **SemEQUAL** gives reasonable performance (couple of seconds for queries involving huge closure sizes) for queries asked against some constant, i.e. say one has to find all the books in Hindi whose category is 'History', the performance deteriorates quickly as number of target languages or ontology increase. This is because closure (which happens to be the most computationally expensive part in the whole process) has to be computed for each of the target languages during every comparison. The situation becomes even more grim when a SemEQUAL join is done instead of comparison against a constant. This effect is evident from Figure 5.4 in Section 5. Thus, there is a case for further optimization of SemEQUAL.

The primary target for such an optimization would be the time taken by closure computation. There is some scope in making closure computation fast. But as already mentioned, these have been already implemented. Some more careful implementation might bring down this time further. But, it is a fundamental nature of closure computation that it is costly. Any optimization of closure computation is likely to bring down the constants but the complexity still remains. One way to solve this problem is to forego the closure computation all together. One way to do this is to precompute the whole closure and use

it. Then the potential recursive nature of SemEQUAL will be converted to a simple join operation. This will bring down the time taken to compute closure to a constant value regardless of closure size. Precomputation of closure however can be a costly affair, but since it has to be done only once, it seems justified. However, it also bloats the space overhead for supporting an additional language. As the WordNets of the languages mature there sizes will grow considerably and that of their closure even more. Thus, supporting multiple languages in precomputed format will be a drag on system's resources. One way around this problem is to compress the closure so that the size is in the order of the WordNet size itself. HOPI index is a method to accomplish this.

## 4.1 The HOPI Index

**HOPI Index**[10] is basically a compressed representation of all possible paths in a graph. It is based on the concept of **2-hop cover**[9] which will be explained subsequently. It has found application as a connection index for XML documents, which provide space and time efficient reachability tests along the ancestor, descendent and link axes to support path expressions with wild cards in XML search engines. The problem presented in case of SemEQUAL is also concerned with finding out the reachability of one node from other (in this case nodes are word forms). Thus, HOPI index is applicable in this case too.

A **2-hop cover** of a graph $G = (V, E)$ is a compact representation of connections in the graph that has been developed by Cohen et al. [9]. Let, $C(G) = (V, T(G))$ be the reflexive and transitive closure of $G$, i.e. $T(G) = \{(x, y)|$ there is a path from x to y in G$\}$ is the set of all connections in G. For each connection $(x, y)$ in $G$ i.e. $((x, y) \in T)$, we choose a node $w$ on a path from $x$ to $y$ as so called *center node* and add $w$ to a set $L_{out}(x)$ of descendents of $x$ and to a set $L_{in}(y)$ of ancestors of $y$. Now, we can test efficiently if two nodes $u$ and $v$ are connected by a path in $G$ by checking if $L_{out}(u) \cap L_{in}(v) = \phi$. There is a path from $u$ to $v$ iff $L_{out}(u) \cap L_{in}(v) \neq \phi$; and this connection from $u$ to $v$ is given by first hop from $u$ to some node $w \in (L_{out}(u) \cap L_{in}(v))$ and second hop from $w$ to $v$. Thus, $w$ is a 2-hop node in a path from $u$ to $v$. For a node $x$, we say that $L(x) = (L_{in}(x), L_{out}(x))$

is a 2-hop label of $x$. A *2-hop cover* of $G$ is a set of 2-hop labels for each node in $G$ that covers all the connections in $G$, i.e. for each edge $(x, y) \in T(G)$, $L_{out}(x) \cap L_{in}(y) \neq \phi$. This 2-hop cover i.e. sets $L_{in}(x)$ and $L_{out}(x)$ for each node $x$ in graph $G$ is called the HOPI Index for the graph. The size of HOPI Index is the sum of sizes of all node labels: $|L| = \sum_{v \in V}(|L_{in}(v)| + |L_{out}(v)|)$.

Building an optimal 2-hop cover is an NP hard problem. Thus, one generally uses approximation algorithms. We describe approximation algorithms for building 2-hop cover. The first one is a naïve algorithm and the second is the algorithm given in [9] and [10]. We need to define a few more terms before we look at these two algorithms.

For a node $w \in V$, $C_{in}(w) = \{v \in V | (w, v) \in T\}$ denotes the set of ancestors of $w$ in $G$, $C_{out}(w) = \{v \in V | (w, v) \in T\}$ the set of descendents of $w$. For subsets $C'_{in}$ of $C_{in}(w)$ and $C'_{out}$ of $C_{out}(w)$, the set

$$S(C'_{in}, w, C'_{out}) = \{(u, v) \in T | u \in C'_{in} \text{ and } v \in C'_{out}\}$$

All these algorithms take the reflexive and transitive closure of the graph $G$, i.e. $C(G) = (V, T(G))$, as input and produce $L_{in}(x)$ and $L_{out}(x)$ sets for every node $x \in V$ as the output.

Algorithm 2 is a pretty straight forward algorithm and is quite fast in execution and easy to implement. This is because of the fact that it does not try to cover the graph in some intelligent fashion. Instead, it covers the graph with nodes as they come to it one by one. This will still lead to compression but will not approach the best that can be done. This is because, it is not trying to find fittest nodes and corresponding links that could be covered in certain order to achieve much better compression. Thus, this algorithm is very fast but does not select nodes intelligently and only achieves minimal compression.

**Algorithm 2** Naïve Algorithm

**Input:** Reflexive and Transitive Closure $C(G)$
**Output:** $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$

1. Initialize $T^{'} \longleftarrow C(G)$. $T^{'}$ contains yet to be uncovered paths.
2. for each node $w \in V$ in the graph.
   (a) Find the sets $C_{in}(w)$ and $C_{out}(w)$.
   (b) Find all the paths from nodes in $C_{in}(w)$ to nodes in $C_{out}(w)$ i.e. $S(C_{in}(w), w, C_{out}(w))$.
   (c) Delete all the path links in the set $S(C_{in}(w), w, C_{out}(w) \bigcap T^{'})$ from $T^{'}$.
   (d) Insert the node $w$ into the set $L_{out}(x)$ for each node $x \in C_{in}(w)$.
   (e) Insert the node $w$ into the set $L_{in}(y)$ for each node $y \in C_{out}(w)$.
3. Return $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$.

The next algorithm proposed in [9] and [10] tries to make choices in an intelligent fashion in order to exploit redundancies in the graph better and achieve far better compression.

To decide which node to pick in order to arrive at a cover with a small size, we consider for a node $w$ and $C^{'}_{in}(w)$ and $C^{'}_{out}(w)$ and the set $S(C^{'}_{in}(w), w, C^{'}_{out}(w)) \cap T^{'}$ that contains all the paths in $G$ from nodes in $C^{'}_{in}(w)$ to nodes in $C^{'}_{out}(w)$ that are not yet covered and for which the score $r(w)$ is maximized. The value of $r(w)$ describes the optimal relation between the number of connections via $w$ that are not yet covered and total number of nodes that lie on such connections. If we choose $w$ with highest $r(w)$ among all nodes, we have to update labels of only a small set of nodes while covering many of the uncovered connections, thus deriving the most benefit out of corresponding increase in size of $L$.

The problem of finding the sets $C^{'}_{in}(w)$ and $C^{'}_{out}(w)$ for a given node $w \in V$ that maximizes the quotient $r(w)$ is equivalent to the problem of finding *densest subgraph* of the so-called *center graph* of $w$. This undirected bipartite graph $CG_w = (V_w, E_w)$ contains two nodes $v_{in}$ and $v_{out}$ for each node $v \in V$ of the original graph. There is an undirected edge $(u_{out}, v_{in}) \in E_w)$ if $(u, v) \in T^{'}$ is a not yet covered connection, $u \in C_{in}(w)$ and

$v \in C_{out}(w)$. All isolated nodes are removed from $V_w$.

The density $\delta_w$ of the subgraph $CG'_w = (V'_w, E'_w)$ is the average degree i.e. the number of incoming and out going edges, of its nodes ($\delta_w = \frac{|E'_w|}{|V'_w|}$), and the densest subgraph of $G_w$ is the subgraph with the highest density. It can be computed by linear time 2-approximation algorithm which interactively removes a node of minimum degree from the graph. This generates a sequence of subgraphs and their densities and the algorithm returns the subgraph with the highest density. The algorithm for computing a 2-hop cover chooses in each step the node $w$ whose center graph has the subgraph $G'_w$ with highest density among all nodes in $V$. From $G'_w$, the sets $C'_{in}(w)$ and $C'_{out}(w)$ are derived and used for updating the cover.

Unlike the naïve algorithm, this one tries to make choices of nodes in an intelligent order, based on some heuristics, to achieve maximum compression. To do this it computes score in linear time for each node in every iteration. There are some improvements by which the number of nodes for which score is recomputed in each iteration can be further reduced, by maintaining all the nodes in some type of priority queue.

## 4.2 Adaptation for WORDNETs

As we saw in the previous section, we can make intelligent choice of the order of node selection thereby covering all the paths with less number of nodes. Also, we can see that Algorithm 3 scales reasonably as the size of graph increases. However if the graph is so large that it cannot fit into main memory then the algorithm does not scale that well. Though the score comparison is linear time theoretically, if the reflexive and transitive closure is large enough, then there will be considerable disk access in which case linear theoretical complexity holds little value. The optimizations proposed rely on maintaining nodes in some form of data structures which becomes harder to implement and less efficient to run.WordNet also presents the problem of handling large graphs. Hence, it becomes imperative that any algorithm employed to construct 2-hop cover for WordNet

has to be backed by disk i.e. most part of the data required for computation remains on the hard disk (in this case, in form of database tables) and only a small portion is copied into main memory and worked on, at any given time. Thus, the algorithm has to be disk aware and try to minimize disk access as much as possible. Furthermore, any arrangement of data based on some kind of data structure which does not perform well on disk can't be used.

In this paper we modified Algorithm 3 so that:

1. It is backed by disk during the entire computation period, so that large WordNets can be handled.

2. It performs reasonably fast enough while still making intelligent choices of order of node selection to cover all the path links in the WordNet with less number of nodes.

3. Also of importance is reduction of CPU time which too is substantial for big WordNets.

First change was to discard instances of direct or intrinsic assumption that the substantial part of data will be in main memory at any given point of time. Also, any optimization based on priority queue was discarded as it would entail sorting and searching. At the same time, scoring of all the nodes at every iteration has to be somehow eschewed and reduced as the computation proceeds, so as to reduce time between successive node selection. Another observation made was that all nodes will figure in $L_{in}$ and $L_{out}$ of at least some node in the graph, hence it would be efficient to identify those nodes which won't produce big savings and not to spend much time intelligently optimizing for such nodes. Also, the implementation should give a bound on maximum main memory required during any point of execution to avoid running out of memory. We present the modified algorithm as Algorithm 4.

First thing to note is that $T'$ i.e. the reflexive and transitive closure, is stored as a table in a database in a (ancestor, descendent) schema and only the nodes are read from disk

one by one from the *result set*. This set of nodes is only sequentially scanned in chunks of some fixed constant size from database. No other operation is done as this is expected to be a huge number of nodes which are not going to be in main memory all at the same time. The only substantial data stored in main memory is the $C_{in}$'s and $C_{out}$'s of nodes as they are scanned sequentially. These are also read in fixed chunks from the data base and also freed as the algorithm moves to next node. Rest of miscellaneous variables are all constant memory requirements. This gives a reasonable upper bound on memory requirement. Also the required memory remains constant as the size of the graph, in this case WordNet grows.

At first look the algorithm seems like a coarser version of Algorithm 3 which it is, since the 2-hop cover that it produces is expected to be inferior to that produced by Algorithm 3. However, this algorithm is much more disk friendly. First, the scoring method is much simpler as it does not look into subsets of $C_{in}$ and $C_{out}$. This reduces additional call to disk and extra processing. The rationale behind the score is that we want to pick those nodes for storing in $L_{in}$ and $L_{out}$ which have a large number of ancestors and descendents both. This is because all the paths connecting all ancestors with all descendents will be covered which accounts for almost entire compression. However, this method may give a higher score to a node whose links are already covered as WordNet is a DAG. That is why it maintains a priority queue called *best node list* of very limited size (say 5) which helps in ensuring (with high probability) that a really good node will be also caught along with some bad nodes. As the computation proceeds more and more bad nodes will be caught. In order to avoid this, they are removed from $V'$ to $B'$ which is actually a list of bad nodes. Thus, after each iteration the total number of nodes to be checked reduces by at least one and as computation proceeds more than one node is removed making subsequent iterations even faster. This leads to shortening of a long tail where expensive computations and disk accesses are made to select best node from lots of bad nodes with little path links to cover. In Algorithm 3, all the bad nodes are disposed of with naïve algorithm(which is much faster) with little loss in compression as these nodes have little

value to offer from point of view of compression.

Regardless of which algorithm is used, certain tricks will improve the performance manifold. One is virtually indispensable. This is the creation of indexes. While calculating $C_{in}$ and $C_{out}$, we find out all the descendents for a particular ancestor and vice-versa. Also, while checking if a particular path link exists between a node in $C_{in}$ to a node in $C_{out}$, we specify both ancestor and descendent. Thus, we create two B-Tree indexes; one on (ancestor, descendent) and other on(descendent, ancestor) on $T'$. This speeds up the process many times as raw table is of large size, making sequential scan infeasible. The sets $T'$, $B'$, $L^{in}$ and $L^{out}$ are resident in table format in the database backing up the program to create HOPI index. These sets, combined together represents the current state of the program. Thus the entire program's state is persistent on the disk except the calculation of fitness of the nodes which are computed one by one, then compared to the previous best nodes and finally discarded. Thus the main memory budget of the program is constant which mans that the program is scalable to graphs (in this case WordNets) of any size.

## 4.3 Issues in adapting HOPI index inside PostgreSQL

Let's look at how the HOPI index changes the computation of SemEQUAL query. We show in Figure 5, two equivalent queries that return a tuple if the word form 'b' belongs in the closure of word form 'a'. Note that this is the most expensive part of the whole SemEQUAL operation. Here we assume that that WordNet is in (parent, child) schema and the HOPI index (i.e. LIN and LOUT) is in (node, element) schema where 'node' represents the particular word form and 'element' represents the word forms that belong to the LIN or LOUT set of the word form in corresponding 'node' column.

The first thing to notice is that the two queries, although equivalent, are totally different. They use different constructs. While the first query calculates the whole closure and then checks if 'b' belongs in the closure of 'a', the second one is a simple join operation.

**Classical Closure**

```
WITH anscdesc as
(SELECT * from parentchild
 UNION
 SELECT anscdesc.parent, parentchild.child
 WHERE anscdesc.child = parentchild.parent
)
SELECT * from anscdesc
where parent = 'a'
and child = 'b';
```

**Closure with HOPI Index**

```
SELECT * from lin, lout
where lout.node = 'a'
and lin.element = 'b'
and lout.element = lin.element;
```

Figure 4.1: Example Rewritten Query

In the previous section, we saw how to efficiently compute HOPI index. This computation is however done by an external program from outside the core engine. This logic can also be pushed into the database engine and done from inside by a **create index** query. However, there are many issues that has to be addressed before this can be done and the index to be subsequently used. There are certain fundamental differences between HOPI index and other classical indexes like B tree, B+ tree or the R trees.

First, HOPI index is not a tree unlike all the other indexes. B trees etc. have a hierarchical structure and have certain properties. They may be balanced (e.g. B trees) or unbalanced (e.g. suffix trees) but they are all trees. In contrast, HOPI index has a flat structure and is in a form of table (or two tables depending on schema). This makes it logically and technically difficult to implement through standard index implementation interfaces like **GiST**[22]. A related problem is that while other indexes traverse from root

downwards to answer queries, the way to use a HOPI Index is to run a union operation on two selections and checking if there is any result.

A far more serious problem both logically and implementation wise is that while all the other indexes are some kind of access methods, HOPI index is a tool for making the operation of checking membership of closure fast. This makes it difficult to place it logically as a index. For example, while in case of other indexes, it is the optimizer which decides whether to use them or not, based on some cost model; it seems that the module that will decide whether HOPI index has to be used or not is the rewriting module. Even this is not going to be straight forward since membership of closure can be represented as a complex query rather than a single operator.

Due to these problems, HOPI index implementation is a complex process. The process is further complicated by the fact that during the bottom up parsing of the query the parent pointer information is lost to the child query structures. This requires maintenance of extra data structures and in place modifications of the data structures during rewriting so that the parents still point to the correct positions. This transformation is done during rewriting phase of the query processing and requires accessing catalog table **pg_multilingual** to resolve wordnet table names corresponding to the language and HOPI index tables is available.

**Algorithm 3** Dense Set Algorithm

**Input:** Reflexive and Transitive Closure $C(G)$
**Output:** $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$

1. Initialize $T^{'} \longleftarrow C(G)$. $T^{'}$ contains yet to be uncovered paths.
2. for each node $w \in V$ in $G$.
   (a) Find out the sets $C_{in}(w)$ and $C_{out}(w)$.
   (b) Pick sets $C^{'}_{in}$ and $C^{'}_{out}$ such that $C^{'}_{in} \in C_{in}(w)$ and $C^{'}_{out} \in C_{out}(w)$ according to some criteria.
   (c) Score the node $w$ according to the following formulae
   $$r(w) = \max_{\substack{C^{'}_{in} \subset C_{in}(w) \\ C^{'}_{out} \subset C_{out}(w)}} \frac{S(C^{'}_{in}, w, C^{'}_{out} \cap T^{'})}{|C^{'}_{in}| + |C^{'}_{out}|}$$
3. Find node $w$ such that,
   $$r(w) = \max_{n \in V} r(n)$$
4. Find out all the paths from nodes in $C^{'}_{in}(w)$ to nodes in $C^{'}_{out}(w)$ i.e. $S(C^{'}_{in}(w), w, C^{'}_{out}(w))$.
5. Delete all the path links in the set $S(C^{'}_{in}(w), w, C^{'}_{out}(w))$ from $T$.
6. Insert the node $w$ into the set $L_{out}(x)$ for each node $x \in C^{'}_{in}(w)$.
7. Insert the node $w$ into the set $L_{in}(y)$ for each node $y \in C^{'}_{out}(w)$.
8. if $(T^{'} \neq \phi)$
   goto step 2.
9. else
   Return $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$.

**Algorithm 4** Modified Algorithm

**Input:** Reflexive and Transitive Closure $C(G)$
**Output:** $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$

1. Initialize $T^{'} \longleftarrow C(G)$. $T^{'}$ contains yet to be uncovered paths.
2. Initialize $V^{'} \longleftarrow V$. $V^{'}$ contains yet to selected nodes.
3. Initialize $B^{'} \longleftarrow \phi$. $B^{'}$ contains suspected low saving nodes.
4. for each node $w \in V^{'}$ in $G$
   (a) Find out the sets $C_{in}(w)$ and $C_{out}(w)$ for node $w$.
   (b) Score the node $w$ according to the following formulae
   $$r(w) = \{1 - \{\frac{|(|C_{in}(w)| - |C_{out}(w)|)|}{(|C_{in}(w)| + |C_{out}(w)|)}\}\} \times (|C_{in}(w)| + |C_{out}(w)|)$$
   (c) If the score of node $w$ is better than the least scoring node in *best node list* OR *best node list* is still not full then
      i. Store $w$ in *best node list*.
      ii. Delete the previous least scoring node from *best node list* in case it was full.
5. for each node $w$ in *best node list*.
   (a) Count the size of the set $S(C_{in}(w), w, C_{out}(w) \cap T^{'})$ for $w$.
   (b) If $|S(C_{in}(w), w, C_{out}(w) \cap T^{'})| \leq threshold \times |V|$ then move $w$ from $V^{'}$ to $B^{'}$.
   (c) Else
      i. Delete all the path links in the set $S(C_{in}(w), w, C_{out}(w) \cap T^{'})$ from $T^{'}$.
      ii. Insert the node $w$ into the set $L_{out}(x)$ for each node $x \in C_{in}(w)$.
      iii. Insert the node $w$ into the set $L_{in}(y)$ for each node $y \in C_{out}(w)$.
6. If $(V^{'} = \phi \| T^{'} = \phi)$ goto step 8
7. Else goto step 4 if a good node is found or if *best node list* is empty.
8. If $(T^{'} = \phi)$ then return $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$.
9. Else use Algorithm 1 for node list $B^{'}$ and rest of path links in $T^{'}$ to further update $L_{in}$ and $L_{out}$.
10. Return $L_{in}(x)$ and $L_{out}(x)$ for each $x \in V$.

# Chapter 5

# Experimental results and Performance studies

## 5.1 Setup

In this section, we experimentally evaluate the persistent database implementation of SemEQUAL and give the results related to building and effect of HOPI index. The WordNet used here is the English WordNet (version 1.5) which has about 110,000 word forms, 80,000 synsets and about 140,000 relationships. The WordNet is represented as a table in **PC** schema. The basic PostgreSQL on which all the studies were performed is of version 8.1.2. The SemEQUAL implementation inside PostgreSQL is in native C language, while the HOPI index creation program is written in Java and connects to the backend PostgreSQL database through JDBC interface. All the experiments were done on 32bit Pentium 4 machine with 1GB RAM and 160GB hard disk. All the timing information provided in the following tables are wall clock times of execution.

## 5.2 Performance Studies

The first study that was performed was the time taken to compute closure for different word forms in the WordNet. This is shown as a graph in Figure 5.1. The x-axis represents

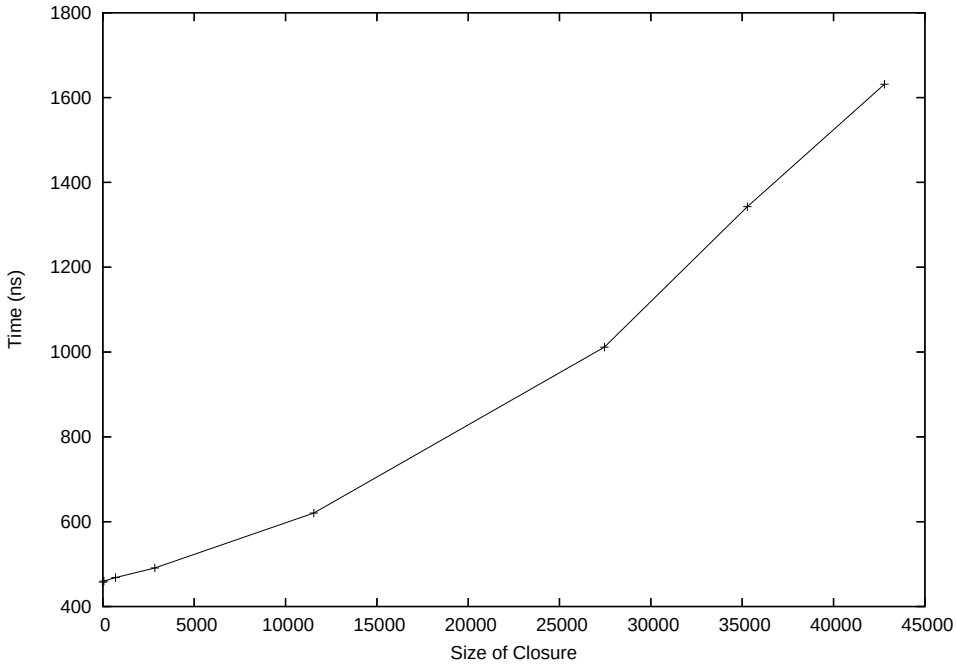



Figure 5.1: Baseline Performance of Closure Computation

the size of closure in number of words. The y-axis represents the time taken to compute closure in milliseconds.

As can be seen in the Figure 5.1, as the size of the closure increases the time required to compute it also increases. But these times in absolute terms are around a second which might be reasonable for most applications and users.

After this, we move our attention to the time taken by SemEQUAL query, when run against a given word form. We plot the time taken by SemEQUAL against a given word form with increasing sizes of closure. This is shown in Figure 5.2.

We see a similar behavior as that of closure computation. This is along expected lines as SemEQUAL does a membership test over base closure computation. Thus, the behavior of SemEQUAL almost mirrors that of closure computation. It also shows that just like

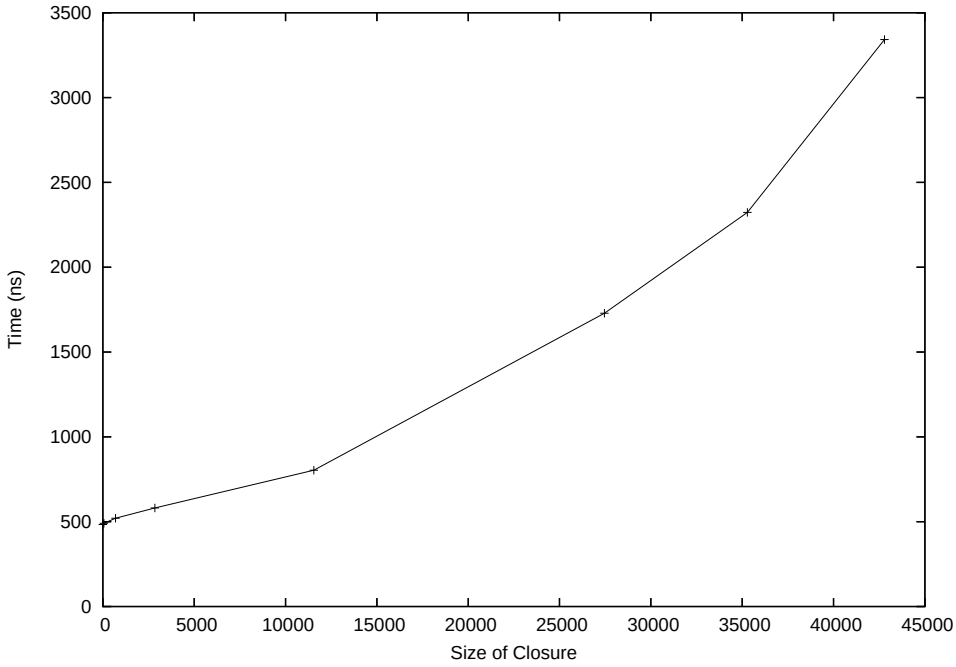



Figure 5.2: Baseline Performance of SemEQUAL Operator

closure computation, the time taken by SemEQUAL for various closure sizes is tolerable.

Next, we compare the performance of SemEQUAL against a given word form but multiple target languages. Currently, we do not have WordNets for languages other than English. Thus, to simulate the desired effect we repeat English in the target language list as many times as number of languages we want. Of course, the assumption here is that WordNets of other languages will be of comparable sizes. Nevertheless, it helps us to see if the response times will be tolerable enough in case of multiple target languages. This is shown in Figure 5.3.

As is evident from Figure 5.3, the performance of SemEQUAL with multiple target languages detoriates rapidly and is not very usable for more than a couple of languages. Another important test is to analyze the performance of SemEQUAL in case of join

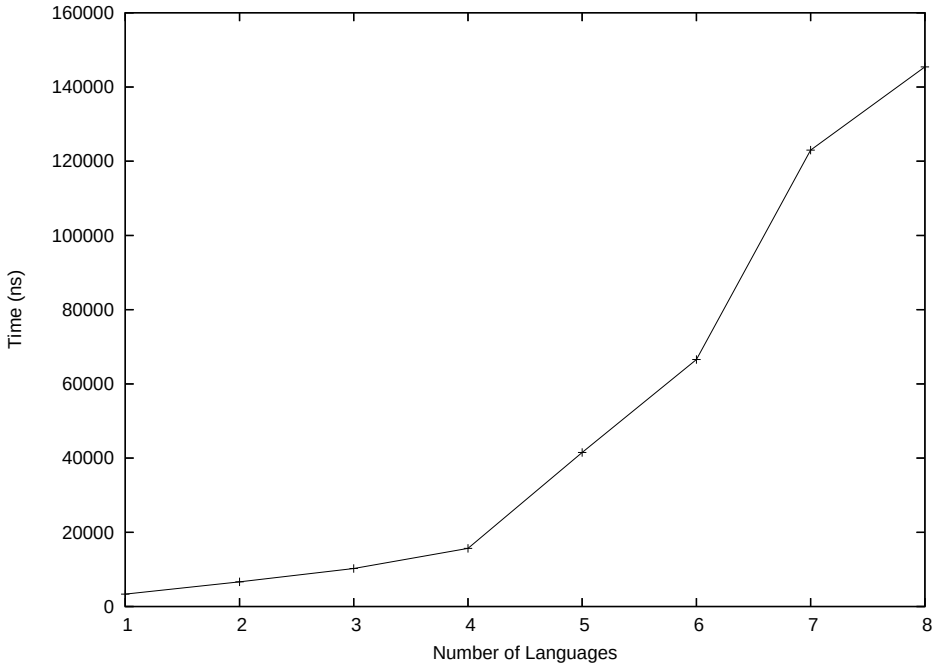



Figure 5.3: Scaling of SemEQUAL with number of Languages

operation. In this case, we join two columns of same or different tables through the SemEQUAL operator. The result of the analysis is presented in Figure 5.4.

As can be seen from Figure 5.4 the performance of SemEQUAL is poor for table sizes larger than few dozens of rows. One should remember however, that these values in Figure 5.3 and Figure 5.4 are dependent on the contents of table and the word forms used. If the closure sizes are large these operations will take more time. In these experiments we have created tables with word forms which have closure size on the higher side. In that sense these are conservative. Still, for large table sizes we can safely conclude from the above experiments that SemEQUAL will be a costly operation to perform.

Thus, there is a motivation towards improving on this performance. One way to do this is to precompute the closure. We analyzed the time required to find out closure from precomputed closure table, both with index and without index. The results are presented

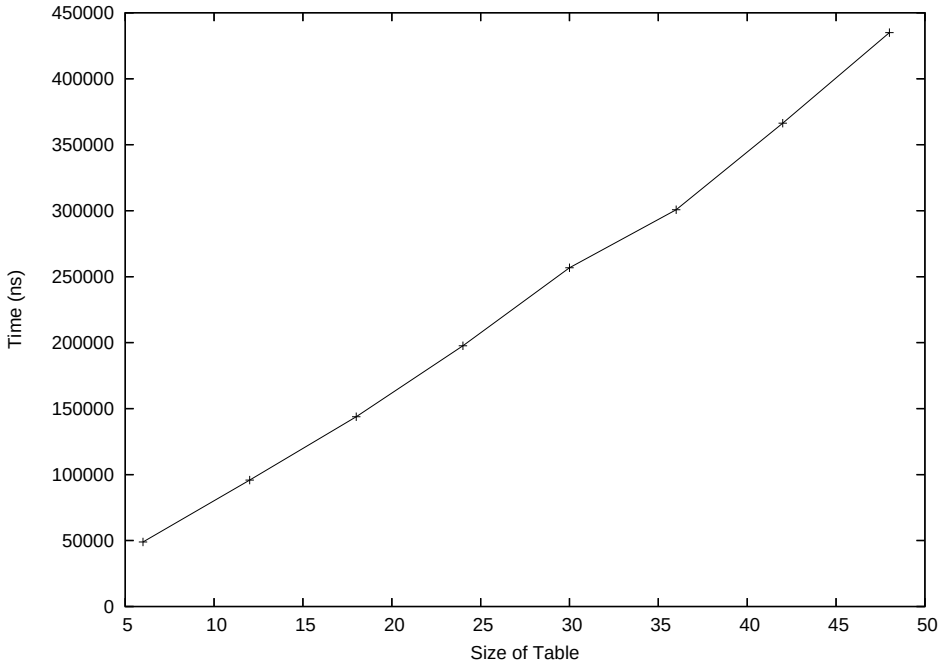



Figure 5.4: Scaling of SemEQUAL with increasing table size

in Figure 5.5.

Figure 5.5 plots the time taken for finding out closure from precomputed tables, with and without B-Tree index, in log scale. It can be clearly seen, that the performance without indexes is unacceptably high, but when indexes are used the results are quite encouraging. Closures of all sizes can be found in under one second. However, the drawback of precomputed closure is that it takes lot of time to compute. However, since this is a one time job, this can be justified in lieu of tremendous improvement in SemEQUAL performance. The more serious problem with precomputed closure is the blow up in the space required. While, the WordNet size is only about 4MB the closure size is 300MB. When supporting multiple languages this can be a strain on systems resources. With the WordNets getting larger the problem will only aggravate. Thus, there is a case for investigating HOPI indexes.

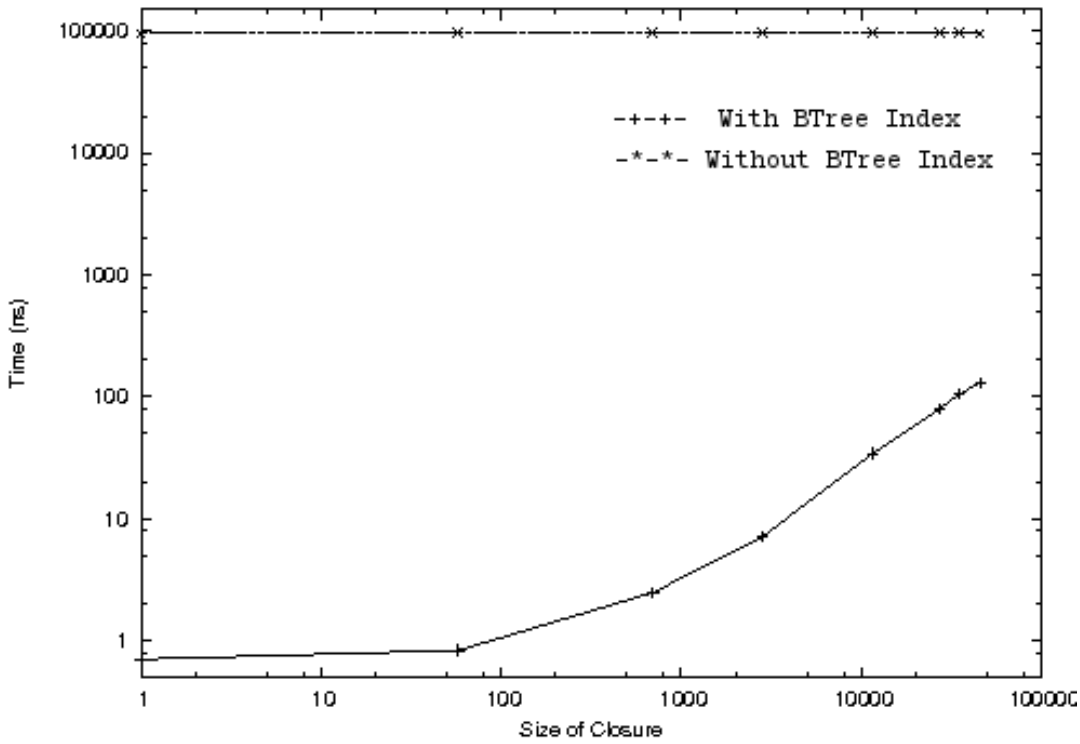



Figure 5.5: Closure Computation for Precomputed Closure With and Without Index

| Name | Size | Compression |
|---|---|---|
| Full Closure | 44,395,764 | - |
| Dense Set Algorithm (Ralf) | 1,183,769 | 37.5 times |
| Modified Algorithm | 351,370 | 126.3 times |

Figure 5.6: Performance of Modified Algorithm compared to Dense Set Algorithm

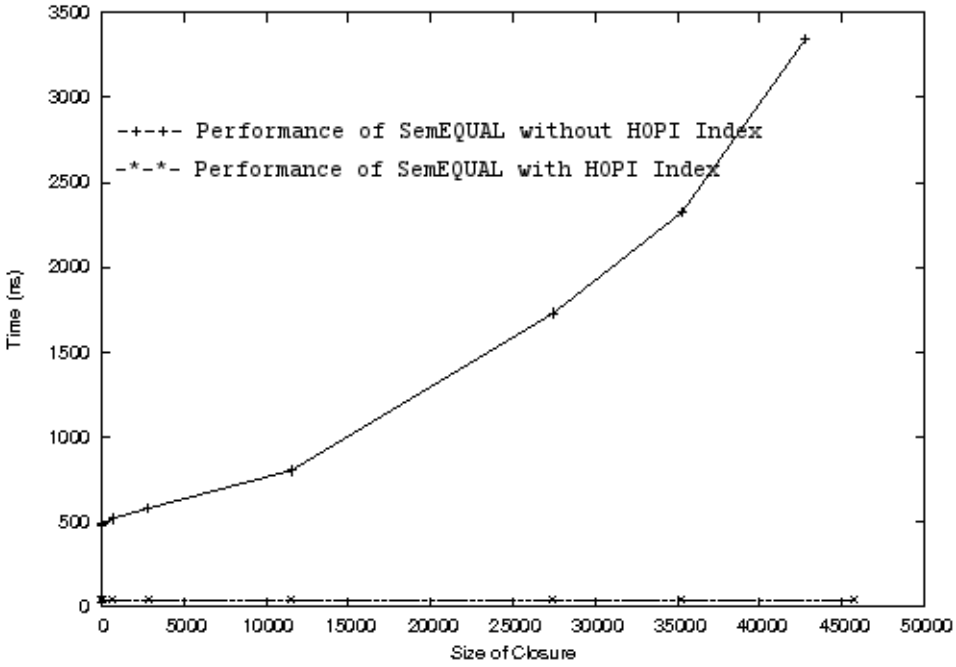



Figure 5.7: Performance of SemEQUAL with and without HOPI Index

Figure 5.6 shows how the performance of modified algorithm given in Algorithm 4 compares with dense set algorithm given in Algorithm 3. Though Algorithm 3 is more sophisticated in its choice of nodes, it can not scale in terms of memory requirement for large graphs. The dense set algorithm has been modified by Prof. Ralf Schenkel and his team to work on large graphs by partitioning them. The performance results for dense set algorithm shown here has been done by his team and has been used by us for comparison purposes. It can be seen quite clearly that modified algorithm gives much better compression then dense set algorithm working on partitioned graphs, as modified algorithm works on the full graph.

As can be seen from Figure 5.7, while the normal SemEQUAL implementation based on recursive computation of closure, takes increasingly more time as the closure size increases, the equivalent query running on HOPI index is immune to increase in closure size. This is expected as because of HOPI index, the recursive closure computation is replaced by a join operation which has almost constant time requirement across different closure sizes.

HOPI index calculation took around five days time. Around fifty good nodes were found to cover more than ninety percent of the entire path links. The final size of the HOPI index was of the same order as that of the original WordNet.

# Chapter 6

# Conclusions and Future Work

This paper defines SemEQUAL functionality from a implementation perspective and presents a complete, persistent and seamless implementation of SemEQUAL functionality on PostgreSQL database management system. The paper also investigates the effectiveness of HOPI index for computation of SemEQUAL and proposes an disk friendly approximation algorithm to compute HOPI index for large ontology.

The future work in this field will focus on the following areas:

1. Implementing LEXEQUAL functionality so that complete multilingual functionality is provided.

2. Further exploration into issues related to HOPI index compatibility with existing index infrastructure with the ultimate goal to seamlessly integrate HOPI index into a database engine such that all recursive queries can benefit from HOPI index and not just SEMEQUAL operator.

# 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. *LexEQUAL: Supporting Multiscript Matching in Relational Systems*, EDBT 2004.

[3] A. Kumaran and J. R. Haritsa. *SemEQUAL: 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 Multilingual Performance Database Systems*, VLDB 2003.

[6] A. Kumaran and J. R. Haritsa. *Multilingual Semantic Operator in SQL*, Technical Report, DSL/SERC.

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

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

[9] E. Cohen et al. *Reachability and distance queries via 2-hop labels*, SODA 2002.

[10] R. Schenkel, A. Theobald and G. Weikum. *Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections*, ICDE 2005.

[11] C. Ordonez. *Optimizing Recursive Queries in SQL*, SIGMOD 2005.

[12] E. Cohen, H. Kaplan and T. Milo. *Labeling Dynamic XML Trees*, PODS 2002.

[13] R. Kaushik et al. *Covering Indexes for Branching Path Queries*, SIGMOD 2002.

[14] J. Han et al. *Some Performance Results on Recursive Query Processing in Relational Database Systems*, ICDE 1986.

[15] Y. Ioannidis. *On the Computation of TC of Relational Operators*, VLDB 1986.

[16] C. Fellbaum. *Wordnet: An Electronic Lexical Database*, page 382, The MIT Press.

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

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

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

[20] http://www.illc.uva.nl/EuroWordNet

[21] http://www.globalwordnet.org

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