Hosting Keyword Search Engine on RDBMS

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Karthik

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2012

TO

My Family

Acknowledgements

I am deeply grateful to Prof. Jayant Haritsa for his unmatched guidance, enthusiasm and supervision. He has always been a source of inspiration for me. I have been extremely lucky to work with him.

Also, I am thankful to M S Vinay for his valuable suggestions. It had been a great experience to work with him.

My sincere thanks goes to my fellow labmates for all the help and suggestions. Also I thank my friends who made my stay at IISc pleasant, and for all the fun we had together.

Finally, I am indebted with gratitude to my parents and brothers for their love and inspiration that no amount of thanks can suffice. This project would not have been possible without their constant support and motivation.

Abstract

Keyword search (KWS) gives an easy interface to RDBMS, which does not require the knowledge of schema information of the published database. Most of the works on KWS engines [3, 7, 5, 9], use main memory data structures to perform required computations to get good performance on execution time and RDBMS is used as storage repository. Our work focuses on effective utilization of RDBMS technologies to process computations involved in providing KWS interface. By this we can get additional benefits from RDBMS back-end technologies to handle large databases and to have persistent KWS indexes.

Two prominent database models used by KWS engines are schema graph based *and* data graph based *KWS models. We have chosen* Labrador *KWS engine [9] as a representative of schema based KWS model and built* DBLabrador, *which is functionally similar to* Labrador, *uses RDBMS to perform all computations and uses additional keyword index. In data graph based KWS model, we have taken 'Providing built-in keyword search capabilities in RDBMS(*PBKSC*)'[8] work, which is based on distinct root semantic answer model. We introduced an alternative keyword index,* Node-Node, *instead of* Node-Keyword *index to reduce the storage space consumed by the keyword index. By using properties of* Node-Node *index, similar to concept mentioned in [4], issues related to storage space of keyword index can be effectively solved by compromising with query search time. Also* Node-Node *index can be effectively used to produce answers for search query in* connected tree semantic model.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

KWS interface to RDBMS is a simple, user-friendly and schema-less text based interface, where user queries the database with a set of keyword terms. Structured queries, like SQL queries, are usual interface to RDBMS which gives precise answers to the search query. Since usage of structured query interface is difficult for naive users and popularity of KWS in World Wide Web, has given motivation to provide keyword search interface to RDBMS as an alternative to structured query interface.

Let us consider *Bollywood* database having relational tables *movie(id, name, year, rating)*, *actor(id, name)* and *movie_actor(movie_id, actor_id, role_name)*. Suppose if we want information about names of movies acted by *'Shahrukh'*, containing term *'Dil'*, with SQL query language interface, we need to construct SQL query like *Table 1.1* and corresponding answer published would be like *Table 1.2*.

```
SELECT a.name
FROM movie as a, movie_actor as b, actor as c
WHERE a.id = b.movie_id AND
c.id = b.actor_id AND
a.name LIKE '%Dil%' AND
c.name LIKE '%Shahrukh%'
```

Table 1.1: SQL query language interface to RDBMS

But usage of KWS interface requires only to type *'Shahrukh Dil'* on text box provided for searching and answers are published like in *Table 1.3*. Important benefits and

Movie_Name
Har Dil Jo Pyar Karega
Phir Bhi Dil Hai Hindustani
Dil Se
Dil To Pagal Hai
Dil Aashna Hai

Table 1.2: Search result of SQL query for 'Shahrukh' and 'Dil'

drawbacks of KWS interface are

- It does not require knowledge of schema information of the database.
- Part of information produced maybe irrelevant to user, i.e. it is not precise like SQL queries.
- Since potential answers for a query are large, it ranks answers by calculating relevance score for each answer.

Movie_name	Rating	Actor_name	Role_name	Relevance-Score
Dil Se	7.4	Shahrukh	Amarkanth Varma	4
Dil To Pagal Hai	6.7	Shahrukh	Rahul	3.5
Phir Bhi Dil Hai Hindustani	5.8	Shahrukh	Ajay Bakshi	1
Dil Aashna Hai	5.2	Shahrukh	Karan D. Singh	0.7
Har Dil Jo Pyar Karega	4.8	Shahrukh	Rahul	0.5

Table 1.3: Search result of KWS query for 'Shahrukh' and 'Dil'

There is a subtle difference in providing KWS interface to web documents and to RDBMS. In web documents, an answer for a keyword query has clear boundary, i.e. a document. In RDBMS world, because of normalization, published database is split into multiple relational tables. So for a given keyword query, a candidate answer can be obtained by joining related relational tuples which, as a whole, contain all the keyword terms. To obtain answers for keyword query, two prominent database models, *schema graph based model* and *data graph based model*, are used.

1.1 Schema Graph based KWS Engines

It uses schema graph of the published database which gives information about relationship between set of relational tables present in it. The relationship between relational tables may be due to foreign key - primary key or by user(DBA) defined relationships.

In our work, we have chosen *Labrador* [9] KWS engine as a representative of schema graph based KWS engines. It uses *column-granularity term frequency hash map* as a data structure, which is a main memory construct. For a keyword query, with the use of column-granularity term frequency hash map, Labrador generates ranked candidate structured queries. A structured query is a form of query where each keyword term is associated with an attribute of the database. User can choose one of the preferred structured query for which Labrador generates appropriate SQL query. Labrador queries RDBMS with generated SQL query and obtains answers for the keyword query. Finally it ranks the answers produced and outputs to user.

We have built DBLabrador engine which follows Labrador's approach for generating ordered answers. Instead of using main memory, relational tables are used for storing keyword indexes. By using declarative language, all processing tasks are pushed to RDBMS. By shifting from Labrador to DBLabrador we get following advantages.

- KWS data structures are *feasible* to handle large databases as they do not have main memory dependency.
- KWS data structures are *persistent*.
- Most of the computational tasks are performed in RDBMS. By this we are able to utilize the automated optimization of function calculation feature and index facilities of RDBMS.
- Removed the dependency on having full text indexes on published attributes by utilizing *cell-level granularity term frequency* keyword index.
- *Cell-level granularity term frequency* keyword index removes the necessity of splitting of string attribute values of answers into terms, which is a necessary step to

calculate relevance scores of answers.

1.2 Data Graph based KWS Engines

It uses data graph of the published database, which represents relationships between published relational tuples, and use it for generating answers for keyword queries. They are schema-less approach as they do not need schema information of the published database. Most of the KWS engines based on data graph, use main memory data structures for storing data graph [3, 7, 5]. Advantage of using data graph based model is that for small databases their search time is faster as they directly get the answers from data graph. Because of main memory constraint they cannot keep data graph of huge database.

We have taken *PBKSC* [8] as a representative of data graph based KWS engines. It uses a *node-keyword* index, which stores *Voronoi paths* from node to keywords within threshold path weight. This keyword index is stored as relational table. Answers for keyword query is obtained from *Node-Keyword* index by appropriate SQL query. Storage space for this keyword index is huge, especially for text-based databases.

We are introducing an alternative *Node-Node* data structure for *PBKSC* KWS engine, which is also stored as relational table. *Node-Node* data structure is inspired from work in [4], which discusses effective data structures to store and retrieve shortest distance between relational tuples. This keyword index stores the shortest paths between nodes within threshold path length of the data graph. Advantages of using *Node-Node* data structure over *Node-Keyword* data structure is

- Takes less storage space for text based database.
- Can be used for connected tree semantic answer model [3].
- Can operate effectively with small threshold path weight to get same quality of answer produced by high threshold path weight Node-Keyword index.

Chapter 2

Schema Graph based KWS Model

In this section, we discuss about our contribution for schema based KWS engines which use main memory data structures and imperative languages. As a representative, we have chosen Labrador [9] KWS engine. We have developed DBLabrador, a version of Labrador which uses RDBMS technologies to store keyword data structures and to perform required computations. Initially basic terminologies used in this section are presented in Section 2.1. Later Labrador's approach is discussed in Section 2.2. After that DBLabrador's approach is explained in Section 2.3. Finally Labrador and DBLabrador KWS engines are compared in Section 2.4. Here both KWS engines implementation is specific to PostgreSQL 8.4. To move these KWS engines to other RDBMS engine requires change in construction of SQL queries, as RDBMS engines differ in the usage of full text indexes and their in built functions.

2.1 Preliminaries

Let $KQ = \{k_1, k_2, ..., k_n\}$ denote set of keyword terms present in the keyword query, $R = \{R_1, R_2, ..., R_m\}$ denote set of relational tables present in the published database. A denote set of published attributes and $A_i = \{a_{i1}, a_{i2}, ..., a_{iq}\}$, $A_i \in A$ denote set of attributes belonging to relational table R_i.

Domain Relational tables are the relational tables whose attributes are going to be published. We assumed that each tuple of domain relational table has unique id, *tuple_id*. It can be obtained from RDBMS, for example PostgreSQL have option of having *oid* field, which will give unique number to each tuple within the relational table. Also we assume, we have index built on *tuple_id* field, which allows fast retrieval of tuples of a domain relational table. In our work we support only string attributes to be published. Also Database Administrator has the option of publishing only selected attributes of the Domain relational tables.

Keyword query contains set of words entered by user in text search box. In the context of *information retrieval*, each such word is called as a *term*. Order of the keyword terms are not important. We follow exact keyword semantics where each stemmed keyword term must be present in the answer.

Schema Graph gives the information about relationship between domain relational tables. Nodes of schema graph are domain relational tables. Edges of the graph represent relationships between corresponding relational tables. In our work we followed Labrador's convention that there is a relationship between two relational tables if they have at least one common attribute name syntactically. This makes natural join between these two relational tables not to be Cartesian product. Here edges are considered to be undirected.

Structured query is a form of query in which each keyword term is assigned to one of the published attributes. Attributes present in a structured query are called *query attributes*. Relational tables involved in a structured query are called *query relational tables*. A keyword term, along with associated attribute of a structured query is called *query predicate*.

A *candidate structured query* is a structured query with following property

- For each *query predicate* $< k_i, a_j >$ of the structured query, a_j must contain k_i.
- There exist at least one spanning subtree in the schema graph containing all *query relational tables* as nodes.

We retain only candidate structured queries and discard other structured queries.

Answer tuple is an information unit produced as an answer to the keyword query. These answer tuples are generated with the help of candidate structured queries.

Symbol table is used to efficiently get location in the published database where keyword terms are present. One of the important design issues of symbol table is the selection of granularity level in which terms in the published database needs to be stored in the index. Two prominent granularity [1] levels are

- *Cell-level Granularity* stores terms in $Table \rightarrow Column \rightarrow Tuple$ level.
- *Column-level Granularity* stores terms in $Table \rightarrow Column$ level.

Comparison of performance of these two granularity level for search algorithms are studied extensively by [1]. Summary of that is, *column-level granularity* performs better when there is an availability of full-text index on published database. Otherwise column-level granularity performs poorly because it needs to make sequential scan of the relational tables during keyword search.

In this section, we need symbol table to keep *term frequency* information. We use *cell-level granularity term frequency* data structure which stores frequency information of terms present in the published data base at cell-level granularity. Similarly we use *column-level granularity term frequency* data structure which stores frequency information of terms present in the published database at column-level granularity.

2.2 Labrador

Interface and processing steps of Labrador are explained in *Figure 2.1*:

1. Labrador provides a single text box interface where user enters keyword query.
2. Corresponding to the keyword query, Labrador generates ranked candidate structured queries.

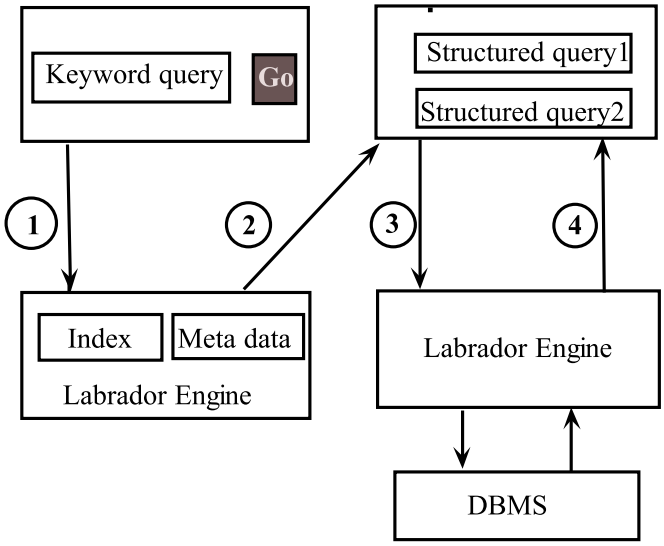

Figure 2.1: Labrador interface

3. Labrador generates and queries DBMS with appropriate dynamic SQL query corresponding to the submitted structured query by the user.

4. Labrador outputs ranked answer tuples to user.

Generation of structured queries helps to classify the possible answer tuples for the keyword query and helps to generate answer tuples belonging to one class.

In Labrador engine, all the computations are performed by *Labrador engine* module (*Figure 2.2*) and RDBMS is used just as storage repository. Major computational part include following tasks.

- Building main memory hash map of column-level granularity term frequency.
- Generating candidate structured queries for keyword query and ranking them.
- Generating SQL query corresponding to a structured query and calculating relevance scores for each of the answer tuple.

2.2.1 Keyword Index

Labrador maintains hash map of *column-level granularity term frequency* data structure in main memory as keyword index. Every time before making Labrador engine functional, hash map needs to be built. Column-level granularity term frequency information helps in

- To get set of published attributes $A_{k_i} \subseteq A$ containing keyword term k_i.
- To get the frequency information of term t_i in an published attribute $a_j \in A$.

The storage space required for column-level granularity term frequency keyword index, is directly proportional to number of distinct terms present in each published attribute. Labrador has not addressed cell level granularity keyword index as it takes more space, which is in the order of database size, and cannot be handled effectively in main memory.

2.2.2 Generation of candidate structured queries and calculating relevance score

For a given keyword query KQ, steps involved to generate all candidate structured queries involves following steps.

- For each term $k_i \epsilon KQ$, identify the set of attributes $A_{k_i} \subseteq A$ containing k_i.
- Generate all possible candidate structured queries by performing Cartesian product of $A_{k_i}, k_i \in KQ$.
- Each of the structured query are checked for plausibility condition.

Relevance score for each of the candidate structured query is calculated using Bayesian network model. Let us denote query attributes of a structured query as QA and query relational tables as QR. Calculation of relevance score of the structured query involves following steps:

- For each term $k \in KQ$ calculate its fitness value using following formula
  $Fitness(k, a_j) = \frac{f_{kj}}{f_{jmax}} \frac{log(1+f_{kj})}{log(1+n_k)}$
  where
  $a_j \in QA$ is the attribute to which k_i is assigned.
  f_{kj}: frequency of the term k in a_j.
  f_{jmax}: frequency of the term having maximum occurrences in the attribute a_j.
  n_k: total number of occurrences of the term k in database.

- Calculate cos value of $\forall a_i \in QA$ by
  $COS(a_i) = \frac{\sum_{k \in K_i} Fitness(k, a_i)}{|K_i|}$.
  $K_i \subseteq KQ$: keyword terms assigned to attribute $a_i \in QA$.

- Relevance score of the structured query is calculated by $\frac{\sum_{a_i \in QA} COS(a_i)}{|QR|}$

2.2.3 Generating answer tuples for submitted structured query and ranking answer tuples

First SQL query is constructed corresponding to user submitted structured query. *FROM* clause of the SQL query is constructed by making natural join of the query relational tables. To generate *WHERE* clause, for each query predicate $< k_i : a_{k_i} >$, get the string a_{k_i} *LIKE* k_i. Since we are dealing with conjunction query, each query predicate is joined by AND condition. SQL query for structured query $< k_1{:}a_{k_1}, k_2{:}a_{k_2}, \ldots, k_n{:}a_{k_n} >$ without full-text index is shown in *Table 2.1.* If the published attributes have full-text

SELECT *
FROM r_1 natural join r_2 ...
WHERE a_{k_1} LIKE k_1 and a_{k_2} LIKE k_2 and ...
$a_{k_n} LIKE\ k_n$

Table 2.1: SQL query for without full-text index

index, then the SQL query corresponding to the structured query is shown in *Table 2.2.* To have a full-text index on an attribute a_i in Postgres, we need to add separate column a_i' where it maintains term informations present in each attribute value in *tsvector* type.

Then on a_i' *GIN/GiST* indexes, which are Postgres in-built indexes, can be built to find tuples containing terms. SQL query of type *Table 2.2* can be used to search tuples containing a term.

SELECT *
FROM r_1 natural join r_2 ...
WHERE a_{k_1}' @@ $to\_tsquery(k_1)$ and
a_{k_2}' @@ $to\_tsquery(k_2)$ and ...
a_{k_n}' @@ $to\_tsquery(k_n)$

Table 2.2: SQL query for with full-text index

Ranking result tuples: Relevance scores for each of the answer tuple is calculated based on Bayesian network model. Steps involved in calculating relevance score for each answer tuple are:

- For each term u_i belonging to query attribute a_j, calculate its weight in that attribute using
  $weight(u_i,a_j) = \log\ (1+ \frac{N}{f_{ij}})$
  where
  f_{ij}: frequency of the term u_i in a_j.
  N: total number of tuples in table r containing attribute a_j.

- For each query attribute a_j present in the result tuple $tuple_i$, calculate its cos value using following formula:
  $\cos(tuple_i,a_j) = \frac{\sum_{k_i \epsilon a_j} weight(k_i)}{\sqrt{\sum_{u_i \epsilon a_j} u_i^2}\sqrt{\sum_{k_i \epsilon a_j} 1}}$
  where
  k_i: represents any keyword term
  u_i represents any term $\in tuple_i$.

- relevance$(tuple_i) = \sum_{a_j} cos(tuple_i, a_j)$
  where $a_j \epsilon$ query attributes.

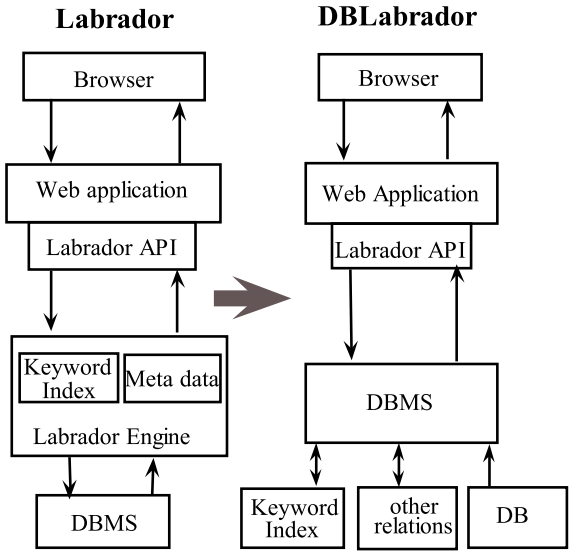

Figure 2.2: Shift in architecture

2.3 DBLabrador

DBLabrador is a modified version of Labrador which utilizes back-end database technology to build suitable keyword data structures and to perform all the computational tasks. Shift from Labrador to DBLabrador [*Figure 2.2*] is made by removing main processing part of Labrador, *Labrador engine*, by pushing all the processing task to DBMS in DBLabrador. Also in DBLabrador, required keyword indexes are maintained by DBMS as relational tables. In this section, we will discuss about implementation part of DBLabrador.

Complete architecture of DBLabrador is shown in *Figure 2.3*. Three main processing parts of DBLabrador are

- Building and storing keyword indexes off-line as relational tables.
- Generating candidate structured queries for a given keyword query and calculating relevance scores of the structured queries.
- Generating answer tuples for submitted structured query and calculating relevance scores of answer tuples.

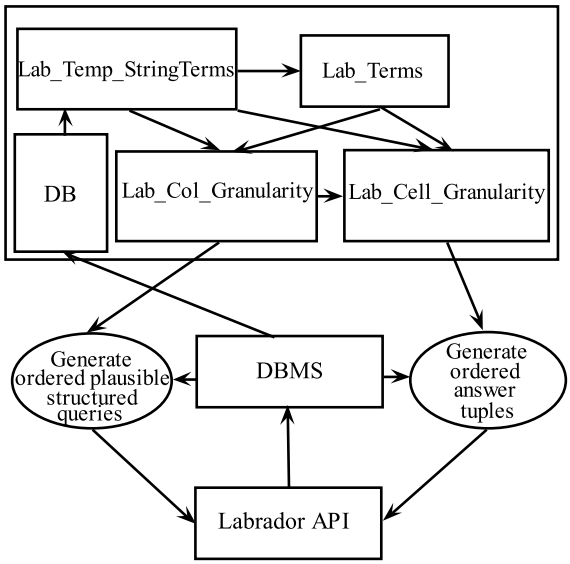

Figure 2.3: DBLabrador architecture

2.3.1 Keyword Indexes

DBLabrador addresses keyword index for both generation of structured queries and generation of answer tuples by using cell-level granularity term frequency information and column-level granularity term frequency information. *Lab_Col_Granularity* and *Lab_Cell_Granularity* are the two relational tables used to store term frequency information at column and cell granularity level, respectively. Important relational tables used in keyword indexing process are

Lab_Temp_StringTerms (tuple_id integer, attr_id integer, term_id integer, frequency integer) This *relational table* is used to store cell-level granularity term frequency information about published database temporarily. This is used to speed up the execution time to build keyword indexes.

Lab_Term (term text, term_id integer) Since string computations are costly, each distinct term present in the published database is assigned a unique integer value. *Lab_Term* relational table is used to store mapping information of distinct terms present in published database and corresponding integer value.

Lab_Col_Granularity (term_id integer, attr_id integer, frequency integer) This relational table is used to store column-level granularity term frequency information. Here *frequency* attribute gives frequency of a term at column-level granularity. To efficiently retrieve set of published attributes $A_{k_i} \subseteq A$ containing k_i, B-tree index is used on *term_id* attribute.

Lab_Cell_Granularity (term_id integer, attr_id integer, tuple_id integer, frequency integer, weight real) This relational table is used to store term frequency information along with weight of each term in cell-level granularity. Here *frequency* attribute stores frequency of the *term_id* at cell-level granularity. Attribute *weight* stores the weight of the term *(section 2.2.3)*. To efficiently retrieve set of tuples having a keyword term in one of its attributes, B-tree index is used on *(term_id, attr_id)* attribute pair. Also B-tree index is used on *(tuple_id, attr_id)* attribute pair to retrieve term frequency information at cell-level granularity.

Steps involved in building keyword indexes are

- Initially scan published database once to populate *Lab_Temp_StringTerms*. This involves scanning each string attribute value(cell) in the published database, splitting into terms and storing cell-id, distinct terms and corresponding frequency.
- Take distinct terms from *Lab_Temp_StringTerms* to populate *Lab_Term*.
- Populate *Lab_Col_Granularity* from *Lab_Temp_StringTerms* and *Lab_Term* relational tables.
- Populate *Lab_Cell_Granularity* from *Lab_Temp_StringTerms*, *Lab_Term* and *Lab_Col_Granularit* relational tables.

2.3.2 Generation of structured queries and calculation of relevance scores

All candidate structured queries are generated for given keyword query with the help of *Lab_Col_Granularity* relational table. The approach for generating candidate structured

queries and calculating relevance scores are similar to Labrador, but DBLabrador uses two dynamic SQL queries instead of using imperative language. *Lab_structure_query* relational table stores candidate structured query information.

2.3.3 Generation and calculation of relevance scores for answer tuples

DBLabrador uses *Lab_Cell_Granularity* keyword index to generate answer tuples as well as to calculate their relevance score.

Generation of answer tuples: Since providing full-text index requires huge storage space, every string attribute in the domain database may not have full-text indexes. But for provinding KWS interface, without full-text indexes on published attributes results in sequential scan of the corresponding relational tables. So DBLabrador maintains separate cell-level granularity term frequency information, without changing domain database, for those published attributes which do not have full-text index and avoids costly sequential scan at runtime. Answer for a structured query can be obtained by using *Table 2.2* approach if full-text index available on published attributes. Otherwise *Lab_Cell_Granularity* table can be used to generate answer tuples by following steps.

- Get distinct relational tables, $R' \subseteq R$, present in the submitted structured query.
- For each relational table, $r_i \in R'$, get all the query predicates, QP_i, whose attribute belongs to the relational table r_i.
- Using *Lab_Cell_Granularity* relational table, for each relational table, $r_i \in R'$, retrieve the tuples which satisfy all query predicates QP_i.
- Finally compute natural join of all the R' relational tables which gives answer tuples. Temporarily store generated answer tuples in *Lab_Result* relational table to calculate relevance scores.

Name	attributes	# tuples	#distinct terms	terms per tuple	size(MB)
Proceedings	(id,key,title,year)	17,982	30,558	20	4.3
InProceedings	(id,key,Proc_key, title,year)	1,112,035	1,353,575	13	206
InProc_Authors	(InProc_id, name)	3,155,373	291,477	2	187
Articles	(id,key,title, journal,year)	826,026	1,022,410	14	154
Article_Authors	(Article_id,name)	2,101,494	260,209	2	125
Books	(id,key,editor, publisher,year)	9,286	19,634	5	1.16
Book_Authors	(Book_id,name)	13,415	9,404	2	0.86
InCollections	(id,key,Book_key, title,year)	22,582	34,034	12	4.5

Table 2.3: DBLP domain relational tables

So it avoids full scan of the domain relations when one of its published attribute does not have full-text index.

Relevance Score calculation for an answer tuple: DBLabrador calculates relevance scores of each answer tuple with the help of *Lab_Cell_Granularity* table. By this splitting of answer tuple strings can be avoided, also weight of each term of the answer tuple need not be calculated as these informations are available from *Lab_Cell_Granularity* table. Approach to compute relevance score is similar to Labrador's approach, but DBLabrador uses one SQL query instead of imperative language.

2.4 Experiments

In this section, we seek to analyze three aspects from Labrador and DBLabrador KWS engines. First one is the importance of persistent keyword data structures for publishing large databases. Second one is the necessity of cell-granularity data structures for KWS engines handling large databases when full-text indexes are not available. Third one is to analyze effect on execution time to calculate relevance scores of answer tuples.

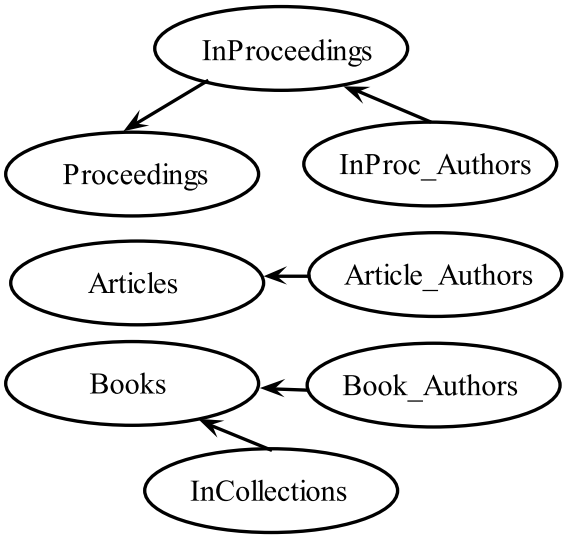

Figure 2.4: Schema graph representation

2.4.1 Experimental setup

All experiments are conducted in PostgreSQL 8.4.8 on Sun Ultra 24, Intel Core(TM) 2 Quad-Core CPU X9650, 3GHz with 8GB Main memory, Ubuntu 10.04 operating system. We also set Postgres parameters *shared_buffers* = 1GB and *work_mem* = 1GB.

We use DBLP dataset for testing purpose. Description of domain database is given in *Table 2.3*. The schema graph of the published database is shown in *Figure 2.4*. The keyword queries used in our experiments are listed in *Table 2.4*.

Search queries	**# output tuples**
paul system	1477
wang environment	722
lee architecture	737
michael algorithm conference	550
andrea network proceeding	738
daniel proceeding conference	499

Table 2.4: Keyword queries

2.4.2 Importance of persistence of keyword data structures

Time to build keyword index on the dataset depends on total number of tuples and terms present in each tuple. General operations involved are splitting of string values into terms, stemming operations of each term and calculating frequency of term.

Name	**Time(sec)**
Proceedings	2
InProceedings	5536
InProc_Authors	611
Articles	271
Article_Authors	157
Books	2
Book_Authors	1
InCollections	6
Total	6315

Table 2.5: Labrador keyword index populating time

Name	**Time(sec)**
Proceedings	5
InProceedings	337
InProc_Authors	181
Articles	220
Article_Authors	126
Books	2
Book_Authors	1
InCollections	7
Lab_Terms	106
Lab_Col_Granularity	148
Lab_Cell_Granularity	6350
TOTAL	7377

Table 2.6: DBLabrador keyword index populating time

Labrador needs to build keyword index every time before it is functional. It maintains a hash bucket for each published attribute to maintain distinct terms present in it and corresponding frequency of the term. This involves hash operation of each term to find its presence in the attribute hash bucket and update its frequency value. Note that

during scan of a domain relational table, all *hash maps of column-level granularity term frequency* of its published attributes are built. So *Table 2.5* shows time taken by Labrador to build keyword index per relational table. *Figure 2.5* shows exponential behavior of Labrador to build keyword index as the size of the database table increases.

By storing keyword indexes as relational tables, DBLabrador gives persistent keyword indexes. DBLabrador keep cell-granularity term frequency information along with column-level term frequency information and does extra computation to materialize weight of each term in cell-granularity level. Calculating weight of each term at cell granularity level involves costly floating point computations. Note that during scan of a domain relational table, *cell-level granularity term frequency* information is obtained. Later keyword indexes *Lab_Col_Granularity* and *Lab_Cell_Granularity* are populated. DBLabrador's time to populate keyword indexes are shown in *Table 2.6*. Here each domain relational table entry involves getting cell-level granularity term frequency information. Populating *Lab_Cell_Granularity* is costly operation as it involves floating point computations. In summary DBLabrador's performance in populating keyword index is comparable with Labrador's performance. In addition DBLabrador gives persistent keyword data structures and materialize weight information of terms at cell-level granularity.

2.4.3 Execution time for getting answer tuples

Without using cell-level granularity data structure or full-text index, keyword search involves sequential scan of the relational table to get answer tuples. To find the effect of this sequential scan, we have conducted experiments on listed keyword queries(*Table 2.4*). For keyword queries having two terms, we have chosen structured query having 2 query relational tables. Similarly for keyword queries having 3 terms, we have chosen structured query having three query relational tables.

Experiment involves three systems,

- Labrador without using full-text index on published attributes.

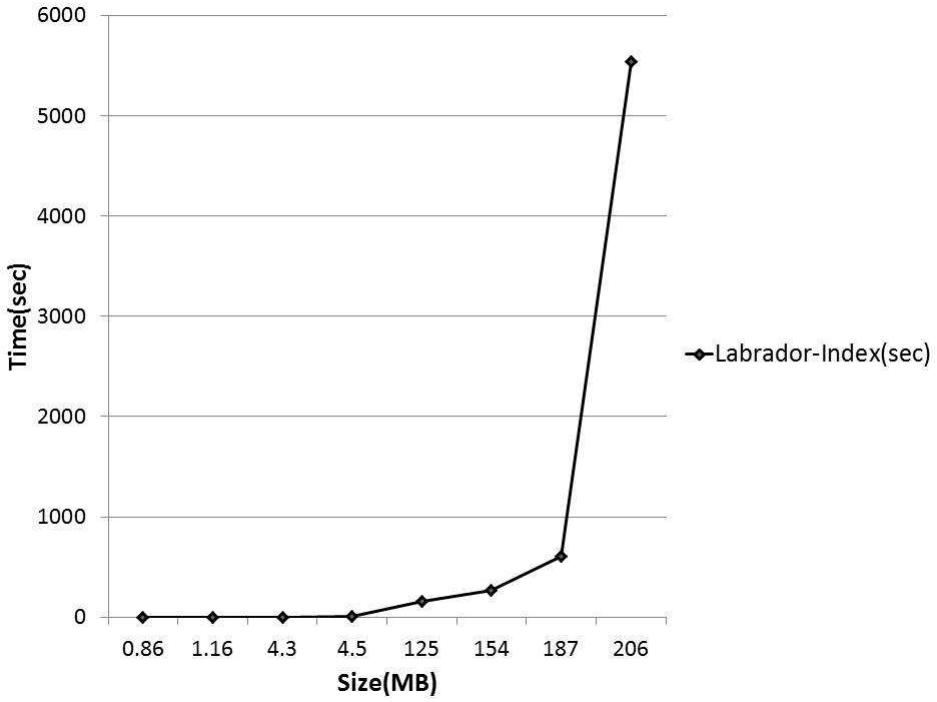

Figure 2.5: Importance of persistent keyword index

- DBLabrador using cell-level granularity term frequency relational table.
- Labrador with full-text index(PostgreSQL GIN index) on published attributes.

Figure 2.6 gives information about performance of three systems. Labrador without full-text index uses more time to get the answer tuples as it uses SQL $LIKE$ operator to get the tuples containing a keyword term. This involves sequential scan of the relational tables. Labrador with full-text index is faster as it uses in-built RDBMS index facilities. DBLabrador using *Lab_cell_granularity* keyword index which is not in-built index of RDBMS, performs less compared to full-text index approach. But its performance is much better compared to Labrador without full-text index. In summary, its better to use cell-level granularity keyword index if full-text index is not present on published attribute. So DBLabrador performs better when full-text index is not available on published attribute compared to Labrador and its performance is almost equal to Labrador when full-text index is present on attribute as both use same SQL query to generate answer tuples.

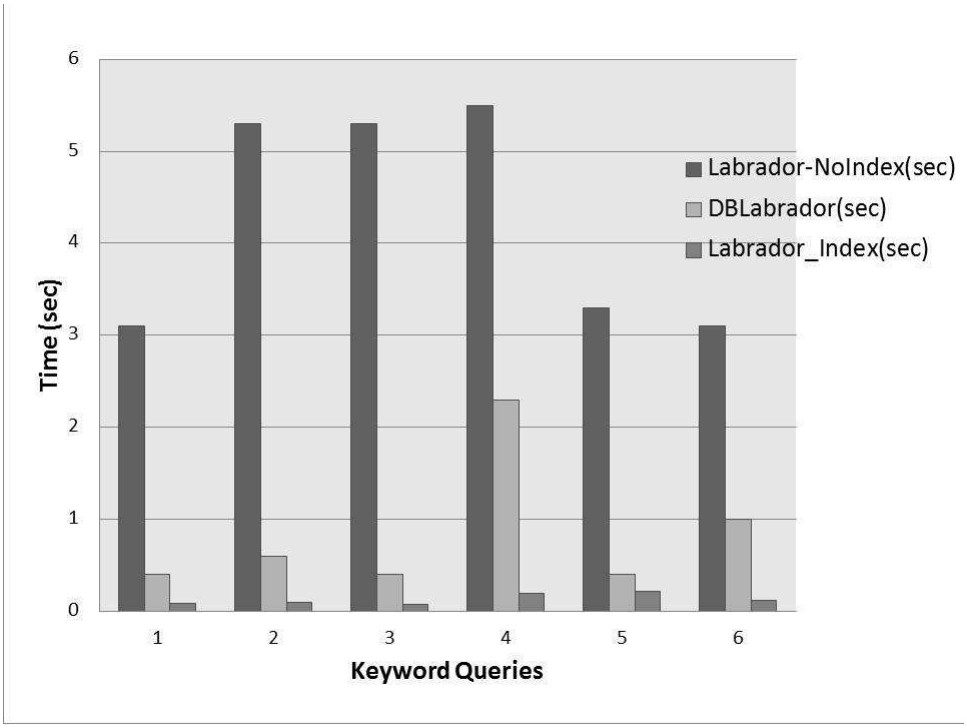

Figure 2.6: Importance of avoiding sequential scan of relational tables

2.4.4 Execution time for ordering answer tuples

Here we compare performance of Labrador and DBLabrador on execution time to order answer tuples. This operation involves calculation of relevance scores of each answer tuple and ordering them based on relevance scores. *Figure 2.7* shows the performance of the two systems. DBLabrador performs better than Labrador because it does not need to split the answer strings into terms and calculate their weight, as it gets these information directly from *Lab_Cell_Granularity* relational table.

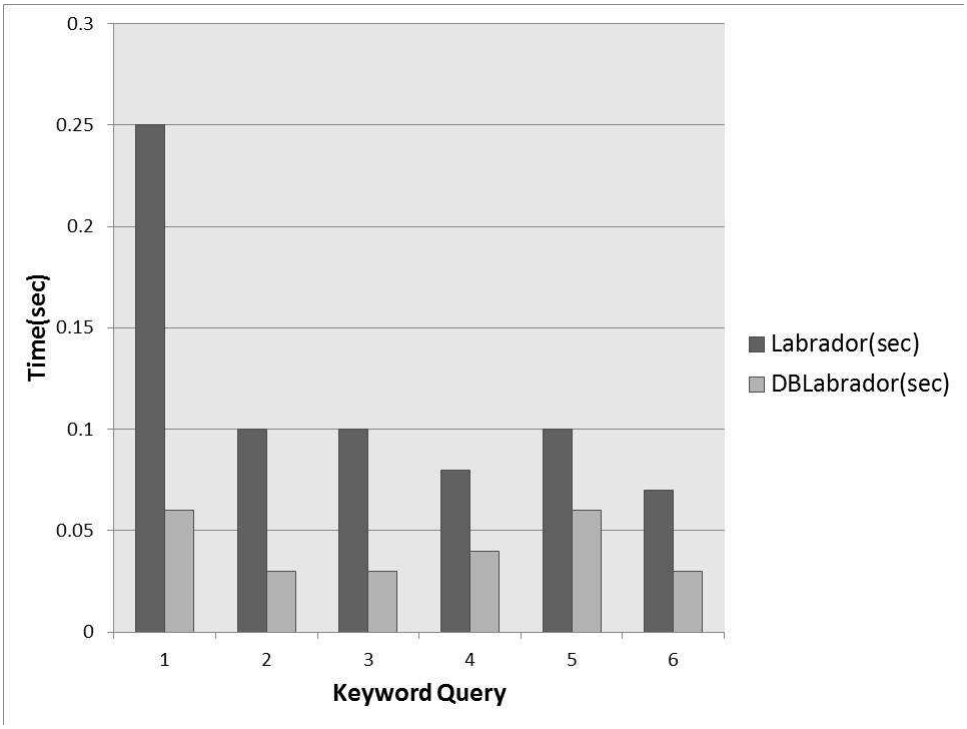

Figure 2.7: Ordering answer tuples

Chapter 3

Data Graph based KWS model

In this chapter we discuss our contribution to data graph based KWS engines. We have taken work of *PBKSC* [8] as representative of KWS engines using data graph approach. First basic concepts related to data graph approach for KWS engines are discussed. Later *PBKSC* KWS engine's approach is explained. Next our proposed *Node-Node* index approach is discussed for *PBKSC* KWS engine. Finally advantages of our approach is shown.

3.1 Basic concepts

Data graph $G = (V, E)$ represents data graph of the published database where $V = \{v_1, v_2, ..., v_n\}$ represents nodes of the data graph, which represents relational tuples of the database and $E = \{(v_i, v_j) | v_i, v_j \in V\}$ represents edges of the data graph. $T = \{t_1, t_2, ..., t_m\}$ represents distinct terms present in the published database. *Term node set* V_i are nodes containing term t_i. Let $V' = \{V_1, V_2, ..., V_m\}$, $V_i \subseteq V$ and V_i contains t_i.

Keyword Query Let $KQ = \{k_1, k_2, ..., k_l\}$ represents keyword query. Let us denote $V_{KQ} = \{V_{k_1}, V_{k_2}, ..., V_{k_l}\}$, $V_{k_i} \in V'$ and contains k_i term.

Steiner Tree ($ST(U, G)$) [8] is defined for a set $U \subseteq V$ on data graph $G(V, E)$. It is defined as a connected subtree of G, which covers all nodes in U. Each $ST(U, G)$

is associated with node r present in the Steiner tree, called root node. *Path weight* of a $ST(U, G)$ having root r, is the sum of path weights from root r to all $u_i \in U$. *Minimum Path Weight Steiner Tree* $MPWST(U, G)$ denotes a Steiner tree having least path weight among all possible Steiner tree on set U.

Group Steiner Tree ($GST(U', G)$) [8] is defined for a set $U' = \{U_1, U_2, ..., U_m\}, U_i \subseteq V$ on data graph $G(V, E)$. It is defined as a connected subtree of G, which covers at least one node from each $U_i, 1 \leqslant i \leqslant m$. *Minimum Path Weight Group Steiner Tree* $MPWGST(U', G)$ defined as minimum path weight Steiner tree of G, which covers at least one node from each $U_i, 1 \leqslant i \leqslant m$.

Voronoi Path $VP(v_i, t_j, G)$ [8] is defined for a node $v_i \in V$ and term $t_j \in T$ on data graph G. It is defined as path having shortest path weight from v_i to V_{t_j} if such path exists,
i.e. $min_{path-weigth}(path(v_i, v_{t_j}), \forall v_{t_j} \in V_{t_j})$.

Compact Steiner Tree $CST(v_i, T', G)$ [8] is defined for a node $v_i \in V$, terms set $T' \subseteq T$ on data graph G. It has following properties

- $CST(v_i, T', G) \in GST(V_{T'}, G)$.
- $\forall t_j \in T'$, $CST(v_i, T', G)$ should contain Voronoi path $VP(v_i, t_j, G)$.
- There should not be any subtree of $CST(v_i, T', G)$ which also satisfy above two conditions.

We refer v_i of a $CST(v_i, T', G)$ as root node. *Weight of* $CST(v_i, T', G)$ is defined as sum of path weight of $VP(v_i, V_{t_j}, G), \forall t_j \in T'$. *Minimum Path Weight Compact Steiner Tree* $MPWCST(T', G)$ is defined as $CST(v_i, T', G)$ having shortest path weight among all possible compact steiner tree on (T', G).
$min_{path-weight}(CST(v_i, T', G), \forall v_i \in V)$

Answer model for a keyword query KQ on data graph G can be based on either $MPWGST$ or $MPWCST$. Since user expects *Top-K* answers for a keyword query KQ, finding answers can be modeled as finding *Top-K* $MPWGST(V_{KQ}, G)$ or *Top-K* $MPWCST(V_{KQ}, G)$.

Connected Tree Semantic answer model (CTS) is based on finding *Top-K* $MPWGST(V_{KQ}, G$ on data graph G for keyword query KQ. In worst case each node $v_j \in V$ can have path to $\forall v_{k_i} \in V_{k_i}, 1 \leqslant i \leqslant l$ and $k_i \in KQ$. In this case total possible candidate answers are $|V| \times \prod_{i=0}^{l} |V_{k_i}|$. From these candidate answers, *Top-K* answers with minimum path-weight needs to be picked.

Distinct Root Semantic answer model (DRS) is based on finding *Top-K* $MPWCST(V_{KQ}, G)$ on data graph G for keyword query KQ. Here a node $v_j \in V$ can become root of an answer only once. So in worst case maximum number of candidate answers for a keyword query is $|V|$. Out of these candidate answers, *Top-K* answers needs to be picked.

Row-level Granularity Index is a keyword index used to store distinct terms present in each relational tuple. In our work, relational table *Row_Granularity(node, term)* is used to store this keyword index. Main operation with this keyword index is, $getNodes(t_i)$, to get set of nodes having term t_i. This can be efficiently performed by having B-tree index on *term* attribute.

3.2 General search algorithm for DRS model

Algorithm-1 [8] describes general approach of DRS model to find *Top-K* answers for keyword query KQ. First *keyword term node* sets $V_{k_i}, 1 \leqslant i \leqslant l$ and $k_i \in KQ$ are calculated. Next step is to generate all possible candidate answers by constructing $CST(v_i, V_{KQ}, G), \forall v_i \in V$. Each candidate answer is also associated with weight which is based on weight of CST. Among all candidate answers, *Top-K* answers are chosen based on weight of answers. Function $drop\_K(ANSWER)$ removes answer with large

weight.

Algorithm 1 Polynomial search algorithm for finding *Top-K* $MPWCST(V_K, G)$

1. **INPUT :** Data graph $G(V, E)$
2. Keyword query KQ
3. Row-level Granularity table RG
4. K:Required number of answers
5. **OUTPUT** $ANSWER$:*Top-K* answers for KQ
6. For each $k_j \in KQ$
   - $V_{k_j} \leftarrow RG.getNodes(k_j)$
7. For each $v_i \in V$
   - Find $CST(v_i, KQ, G)$
   - If ($CST(v_i, KQ, G)$ is among $Top - K$)
     - If($|ANSWER| \geqslant K$)
       - $drop_K(ANSWER)$
       - $ANSWER \leftarrow CST(v_i, KQ, G)$
     - Else
       - Discard $CST(v_i, KQ, G)$

3.3 Node-Keyword index based approach

First node-keyword index is discussed in detail. Next search algorithm used by *PBKSC* KWS engine is discussed.

Node-Keyword index

It stores Voronoi path information $\{VP(v_i, t_j, G) | \forall v_i \in V, t_j \in T\}$. Voronoi path having path-weight greater than a threshold path weight are not considered important for getting answers for search query, so those path informations are not stored.

Relational table *Node-Keyword(node, term, path, path-weight)* is used to store *Node-Keyword* index. Here <*node,term*> attribute value constitute primary key value. Attribute value *path* stores Voronoi path information and *path-weight* gives path weight of the corresponding Voronoi path.

Main operation on *Node-Keyword* relational table is $getVNodes(t_i)$, to get set of nodes NVP_{t_i} which have Voronoi path to a term t_i less than threshold path-weight. By using B-tree index on attribute *term* of *Node-Keyword* relational table, $getVNodes(t_i)$ operation can be effectively performed.

3.3.1 Search algorithm using Node-Keyword index

Algorithm-2 [8] describes the approach to get search results for keyword query using *Node-Keyword* index. First for each keyword term k_i, it gets set of nodes NVP_{k_i} having Voronoi path to k_i. Intersection of $NVP_{k_i}, \forall k_i \in KQ$ gives candidate answers. Function *ftopk* selects *Top-K* candidate answers based on weight of the answer. *PBKSC* KWS engine uses equivalent SQL query of *Algorithm-2* to get *Top-K* answers.

Algorithm 2 Search algorithm for finding *Top-K* $MPWCST(V_{KQ}, G)$ using Node-Keyword index

1. **INPUT :** Data graph $G(V, E)$
2. Keyword query KQ
3. Node-Keyword index NK
4. K:Required number of answers
5. **OUTPUT** $ANSWER$:*Top-K* answers for KQ
6. For each $k_j \in KQ$
   - $NVP_{k_i} \leftarrow NK.getVNodes(k_i)$
7. $CAN \leftarrow \bigcap_{i=0}^{l} NVP_{k_i}$
8. $ANSWER \leftarrow ftopk(CAN)$

3.4 Node-Node index based approach

In this section we discuss about keyword search algorithm based on *Node-Node* index approach. First we discuss *Node-Node* index, then later modified search algorithm is described.

Node-Node Index

Here we materialize shortest path information between pair of nodes. Like node-keyword index, only path information of nodes which is less than threshold path weight is stored. The motivation for *Node-Node* index is that for text based database, number of distinct terms present within the region of threshold path weight of a node, is very large compared to number of nodes present in the region.

Goldman et al. [4] has also discussed storing shortest path information between pair of nodes within threshold path-weight. They also address problem of storage space and use *hub indexing* mechanism. Their work is for *Find/Near* keyword query semantics, where they need to find shortest distance between nodes belonging to *Find* set and *Near* set. They also mentioned about advantages of *self joins* to compute shortest path between pair of nodes. Our work concerns about effectively utilizing *Node-Node* index for computing *Voronoi paths* as an alternative of *Node-Keyword* index, thus getting benefit of using less storage space.

Node-Node index is stored as relational table, *Node-Node(node1, node2, path, path-weight)*. Here *<node1,node2>* value forms primary key. Attribute *path* stores the path from *node1* to *node2* and attribute *path-weight* stores corresponding path weight.

Main operations of *Node-Node* index are $getNNodes(U)$, $getMinNodes(U)$. Operation $getNNodes(U)$ helps to get all nodes of the data graph G which have shortest path to $U \subseteq V$ having less than threshold path weight along with *all path information*. Operation $getMinNodes(U)$ is similar to $getNNodes(U)$, but gives only *shortest path information*. By using B-tree index on attribute *node1* these operations can be effectively performed.

3.4.1 Search algorithm using Node-Node index

Algorithm-3 describes the approach to get search results for keyword query using Node-Node index. Initially *keyword term node set*, V_{k_i} for each keyword term k_i is obtained from *Row-level granularity* relational table. Next set of nodes, NVP_{k_i}, for keyword term k_i having Voronoi path less than threshold path weight can be obtained from *Node-Node* keyword index by using $getMinNodes(V_{k_i})$ function. Intersection of $NVP_{k_i}, \forall k_i \in KQ$ gives candidate answers. Function *ftopk* selects *Top-K* candidate answers based on weight of the answer.

Algorithm 3 Search algorithm for finding *Top-K* $MPWCST(V_{KQ}, G)$ using Node-Node index

1. **INPUT :** Data graph $G(V, E)$
2. Keyword query KQ
3. Node-Node index NN
4. Row-level Granularity table RG
5. K:Required number of answers
6. **OUTPUT** $ANSWER$:*Top-K* answers for KQ
7. For each $k_j \in KQ$
   - $V_{k_i} \leftarrow RG.getNodes(k_i)$
8. For each $k_j \in KQ$
   - $NVP_{k_i} \leftarrow NN.getMinNodes(V_{k_i})$
9. $CAN \leftarrow \bigcap_{i=0}^{l} NVP_{k_i}$
10. $ANSWER \leftarrow ftopk(CAN)$

3.5 Effect of threshold path weight

The problem with *Node-Keyword* or *Node-Node* index is usage of huge storage space. Since answers for keyword search having large weight are not significant, they can be discarded. But there is no theoretical bound for this threshold path weight. So from user perspective, threshold path weight affects the quality of answer. Whereas from publisher's perspective, threshold path weight affects storage space required. In this section we analyze effect of this threshold path weight on storage space and quality of the answer.

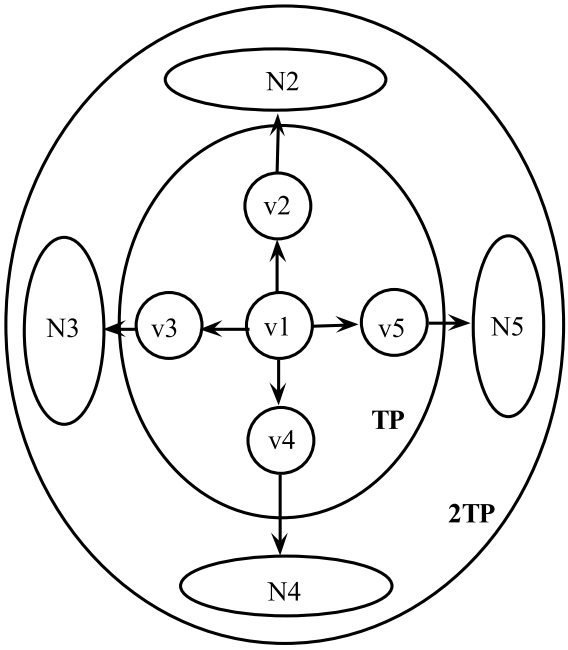

Figure 3.1: Effect of threshold path weight

Figure 3.1 shows a part of data graph G. Here $v_1 \in V$ is the interested node. $v_2, v_3, v_4, v_5 \in V$ are neighbors of v_1 and are only nodes which are within threshold path weight TP. $N_i, 2 \leqslant i \leqslant 5$ are the set of nodes which are within threshold path weight $2TP$, and their shortest path to v_1 is through $v_i, 2 \leqslant i \leqslant 5$ respectively.

3.5.1 Node-Keyword index

Node-Keyword index may not store shortest path information of all nodes which are within threshold path weight, as some nodes do not have any term or do not have any term $t_j \in T$ for which there is Voronoi path $VP(v_1, t_j)$. Suppose to make *Node-keyword* index for threshold path weight twice of previous threshold path weight, costly graph traversal algorithm needs to be used.

Suppose in the *Figure 8* if *Node-Keyword* index does not store shortest path information of $v_1 \rightarrow v_2$ then to get shortest path weight information of $v_1 \rightsquigarrow N_{v_2}$, costly graph traversal algorithm needs to be used.

3.5.2 Node-Node index

Node-Node index stores path information of all nodes which are within threshold path weight. This makes it easier to have *Node-Node* index of path weight twice of previous path weight by simple self join operation of *Node-Node* index.

In the *Figure 8* *Node-Node* index store shortest path information of $\{v_1 \rightarrow v_i, 2 \leqslant i \leqslant 5\}$. To get shortest path weight information of $\{v_1 \rightsquigarrow N_{v_i}, 2 \leqslant i \leqslant 5\}$, joining operation of $\{v_1 \rightarrow v_i, v_i \rightarrow N_{v_i}\}$ needs to be performed. This operation can be easily performed by self-join of *Node-Node* relational table.

3.6 Finding *Top-K* $MPWGST(V_{KQ}, G)$

In this section method mentioned in [8] to get *Top-K* $MPWGST(V_{KQ}, G)$ answer from $MPWCST(V_{KQ}, G)$ for keyword query KQ is discussed. ANS_{CST} represents *Top-K* $MPWCST(V_{KQ}, G)$ answers and set $Root_{CST}$ contains respective root nodes. ANS_{GST} represents *Top-K* $MPWGST(V_{KQ}, G)$ answers and set $Root_{ans}$ contains respective root nodes.

Algorithm 4 gives procedure to get *Top-K* $MPWGST(V_{KQ}, G)$ answer from $MPWCST(V_{KQ}, G)$ for keyword query KQ. Since weight of Kth element of ANS_{GST} cannot be greater than Kth element of ANS_{CST}, each member of ANS_{CST} is assigned to ANS_{GST} [1-3 line].

Next [4-14 lines] root $c'_i \in Root_{ans}$ of element of ANS_{GST} is taken in increasing order of weight if c'_i is not seen before. All possible $GST(V'_{KQ}, G)$ are generated having root as c'_i, represented as Can_{ans}, using function *allGST(c'_i)*. Each element of Can_{ans} is pruned if its weight is greater than or equal to weight of Kth element of ANS_{GST}. Else it is added in appropriate position of ANS_{GST} and previous Kth element is removed. Similarly $Root_{ans}$ is updated accordingly. This procedure is continued till $(K-1)th$ element of ANS_{GST} is reached.

The main operation in this process is, *allGST(r)*, generating all possible $GST(V'_K, G)$ having root node r. *PBKSC* KWS engine accomplishes this work by using Dijkstra's algorithm on data graph G as *Node-Keyword* index cannot be used. But *Node-Node* index helps in this process as it provides $getNNodes(U)$ function.

3.7 Experiments

In this section, we seek to analyze two aspects from *Node-Keyword* index and *Node-Node* index. First one is the utilization of storage space by the two keyword indexes. Second one is the performance of execution of search query by the two approaches. Currently our experiments include cases where database and keyword indexes fit in main memory.

Experimental setup

All experiments are conducted in PostgreSQL 8.4.8 on Sun Ultra 24, Intel Core(TM) 2 Quad-Core CPU X9650, 3GHz with 8GB Main memory, Ubuntu 10.04 operating system. We also set Postgres parameters *shared_buffers* = 1GB and *work_mem* = 1GB.

We use part of DBLP dataset for testing purpose. Description of domain database is given in *Table 3.1*. The schema graph of the published database is shown in *Figure 3.2*. The keyword queries used in our experiments are listed in *Table 3.2*

Algorithm 4 Getting *Top-K* $MPWGST(V_{KQ}, G)$ from *Top-K* $MPWCST(V_{KQ}, G)$

1. **INPUT :** Data graph $G(V, E)$
2. Keyword query KQ
3. $ANS_{CST} = \{a_{c_i}, 1 \leqslant i \leqslant K\}$
4. $Root_{dist} = \{c_i, 1 \leqslant i \leqslant K\}$
5. K:Required number of answers
6. **OUTPUT** $ANSWER$:*Top-K* answers for KQ
7. ANS_{GST}
8. $Root_{ans}$
9. For $i = 1 \rightarrow K$
   - $ANS_{GST} \leftarrow a_{c_i}$
   - $Root_{ans} \leftarrow c_i$
10. For $i = 1 \rightarrow K$
    - If$Root_{ans}.NotSeen(c'_i)$
      - $Can_{ans} \leftarrow allGST(c'_i)$
      - For $c_{GST} \in Can_{ans}$
        * If$IsTopk(c_{GST})$
        * $Update(ANS_{GST}, c_{GST})$
        * $Update(Root_{dist}, c_{GST})$

Name	attributes	# tuples	#distinct terms	#terms per tuple	size
Proceedings	(proc_id,title, seriesid,pub_id)	2,968	9,891	18	776kB
InProceedings	(inproc_id,proc_id, title,year)	212,268	98,310	9	24MB
Person	(personid, name)	174,709	101,671	2.3	10MB
Publication	(pub_id,name)	86	170	3	8kB
Series	(seriesid, title)	24	68	4.65	8kB
Inproc_Person	(personid, inproceedingid)	491,777	NA	NA	21MB

Table 3.1: DBLP domain relational tables

Search queries
Database system
Main Preprocess
operating system
learning programming language
System

Table 3.2: Keyword queries

3.7.1 Usage of storage space

Figure 3.3 gives information about number of path information stored by both keyword indexes. Clearly *Node-Node* index stores less path information. If we compare disk space utilization, *Node-Keyword* takes 1096MB storage space while *Node-Node* index takes 335MB.

3.7.2 Performance on search queries

Figure 3.4 shows the performance of two keyword indexes on set of keyword queries mentioned in *Table 3.2*. These keyword queries are chosen so that number of possible answers are more than 5,000. Here performance is almost equal, as time to handle large *Node-Keyword* index balances extra computation required for *Node-Node* index approach. We

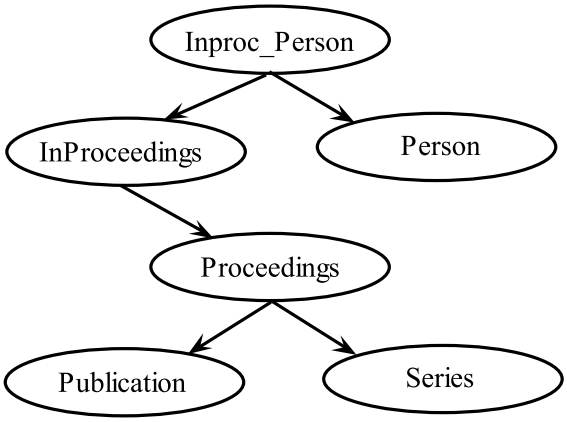

Figure 3.2: Schema graph

can clearly observe effect of handling large *Node-Keyword* index for keyword query 'system' as *Node-Node* index approach performs better as its computation overhead (only one keyword term) is less.

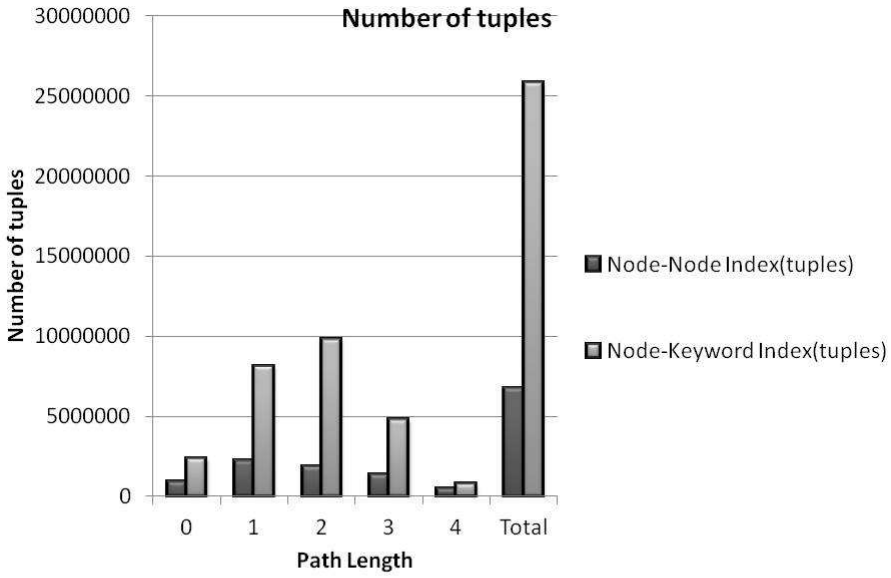

Figure 3.3: Storage space usage by keyword indexes

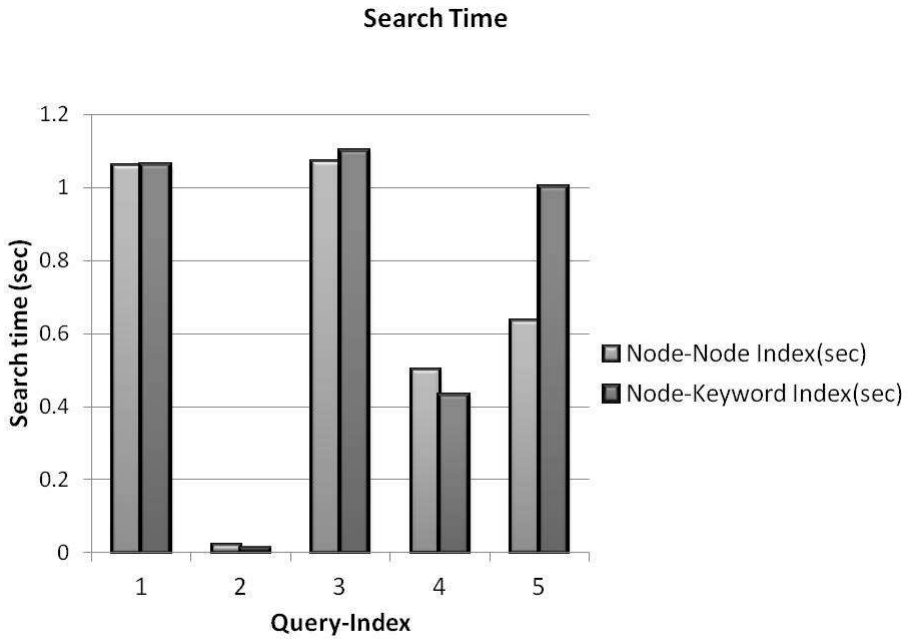

Figure 3.4: Performance on keyword search queries

Chapter 4

Related Work

Work to introduce KWS interface to RDBMS is mainly based on *schema graph* based approach [1, 6] and *data graph* based approach [3, 7, 5]. We have extensively discussed about [9] in *Section-2.2*, [8] in *Section 3.3* and [4] in *Section 3.4.1*. In this section we discuss other related work.

DBXplorer [1] and *DISCOVER* [6] are based on schema graph based, *Candidate Network(CN)* generation and evaluation approach. *CN*s are similar to structured queries, but is intended for CTS answer model. They generate answers for all *CN*s by constructing appropriate SQL queries. *DBXplorer* mainly discusses about symbol table design issues and effective storage space utilization. *DISCOVER* mainly discusses about effective way to evaluate *CN*s, by materializing intermediate results. Their ranking mechanism of answer tuples is weak, as it considers only structure of answer tuples and does not consider content of the answer tuples. *Labrador and DBLabrador*'s approach differs from these approaches as it classifies answer tuples by submitting ranked structured queries to user. Also it ranks answer tuples based on its structure as well as content of the answer.

[2] discusses about performance issues of KWS engines, that for some queries response time is very large. Their solution is fix response time, send generated answers within this time limit and send query forms which explore remaining possible answer space. Since *Labrador and DBLabrador*'s approach does classifies possible answers and generates answer for particular answer class, it does not affected by performance issues much.

BANKS [3] and *Bidirectional Traversal-BANKS* [7] are based on data graph approach, which use main memory data structure containing data graph information. For search queries, they uses data graph traversal algorithms, *Backward Search(BWS)* [3] and *Forward Search(FSW)* [7] to get answers. Problem with this approach is that, it cannot be used for large database. Also it does not use any keyword indexes to traverse the data graph.

BLINKS [5] is also based on data graph based approach which uses two main memory keyword indexes, *Keyword-Node* for *BWS* algorithm and *Node-Keyword* for *FWS* algorithm. To reduce storage space consumed by the keyword indexes, it partitions data graph into blocks and use bi-level keyword indexing. Still it consumes more storage space than *BANKS*, so it is not practical for large databases.

[10] discuss about effective utilization of RDBMS capabilities without usage of any additional KWS data structures for different KWS models. Main disadvantage of this approach is that its performance cannot be matched with main memory based KWS engines. Our approach is to utilize RDBMS capabilities to effectively build KWS data structures and we have showed that we are comparable with previous approaches.

Chapter 5

Conclusions

We have effectively utilized RDBMS back-end technologies to provide KWS interface to RDBMS on two prominent database models.

In schema graph based KWS model, we have taken Labrador engine [9] and built DBLabrador which uses RDBMS back-end technologies. DBLabrador gives persistent keyword indexes and removes the dependency on having full-text index on published attributes. In experiments we have shown that performance of DBLabrador is comparable with Labrador using full-text indexes.

In data graph based KWS model, we have introduced an alternative keyword index for [8] work. Compared to *Node-Keyword* index, *Node-Node* index uses less storage space for text based databases and gives comparable performance. Also *Node-Node* index uses RDBMS back-end technologies in *self join* procedure, which allows it be operated with small threshold path weight to get same quality of answer produced by high threshold path weight Node-Keyword index, and in the process to produce CTS answer model, where *Node-Keyword* index approach depends on main memory procedures.

For future work on *Node-Node* index approach includes trying to reduce storage space by using hub indexing method [4] and comparing it with our approach of increasing threshold path weight at runtime.

Bibliography

[1] S. Agrawal, S. Chaudhuri and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", *ICDE 2002*.

[2] A. Baid, I. Rae, J. Li, A. Doan and J. Naughton, "Toward Scalable Keyword Search over Relational Data", *VLDB 2010*.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti and S. Sudarshan, "Keyword Searching and Browsing in Databases using BANKS", *ICDE 2002*.

[4] R. Goldman, N. Shivakumar, S. Venkatasubramanian and H. Garcia-Molina, "Proximity Search in Databases", *VLDB 1998*.

[5] H. He, H. Wang, J. Yang and P.S. Yu, "BLINKS: Ranked Keyword Searches on Graphs", *SIGMOD 2007*.

[6] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases", *VLDB 2002*.

[7] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases", *VLDB 2005*.

[8] G. Li, J. Feng, X. Zhou and J. Wang, "Providing built-in keyword search capabilities in RDBMS", *VLDB Journal 2010*.

[9] F. Mesquita, A.S.d. Silva, E.S.d. Moura, P. Calado and A.H.F. Laender, "LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces", *Information Processing and Management, vol. 43, 2006*.

[10] L. Qin, J.X. Yu and L. Chang, "Keyword Search in Databases: The Power of RDBMS", *SIGMOD 2009*.