Keyword Index: Design and Implementation Inside RDBMS

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Amit Mehta

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2013

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 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 searching (KWS) is a widely accepted mechanism for querying in Information Retrieval (IR) systems and Internet search engines on the Web. In recent years, the problem of KWS in relational database systems is considered as one of the major research problems in database community. Various solutions are proposed for making whole or part of KWS processing efficient. These wide ranges of solutions touched all the engineering issues like testing of proposed KWS techniques, finding parameters which affect the relevance of output, their impact on quality of output, benefit of integrating KWS systems at various levels of Relational world etc. In this report, we have proposed a new trie based keyword index which stores term (keyword) related information such as schematic location of term in the database, frequency of term at cell level granularity (where a cell means data stored in particular <row, column>) etc. This keyword index is used to speed-up whole KWS processing. Previous KWS system stores term related information in inverted indexes which are either main memory data structures and used at the application level, or as relational tables [1]. In both the cases, DBMS engine is not aware of keyword index. So we have proposed implementing keyword index inside RDBMS, to benefit various stages of KWS and for efficient access of relations involved in KWS processing. We have compared KWS system using our trie based keyword index with DBLabrador KWS system [1], on end to end runtime performance basis. Experimental results shows better performance of proposed system compare to DBLabrador [1].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Keyword searching (KWS) in relational databases is a widely studied problem in last decade. It aims to provide an easy interface to RDBMS, which does not require the knowledge of schema information of the underlying database. For a given keyword query (set of terms given by the user to search), KWS interface tries to generate and execute suitable SQL query and gives resulting tuples back to the user. Various solutions are proposed for making whole or part of KWS processing efficient. These wide ranges of solutions touched all the engineering issues like testing of proposed KWS techniques, finding parameters which affect the relevance of output, their impact on quality of output, benefit of integrating KWS systems at various levels of Relational world etc.

[11] has compared state-of-the-art KWS systems which reveal the issues like lack of scalability and performance in these systems. Even many techniques cannot scale to moderately-sized datasets that contain roughly a million tuples. Also KWS systems which are scalable are not able to perform KWS in timely manner. As relational database store very large amount of information, KWS in such large database require efficient and faster way to construct and execute SQL query corresponding to given keyword query. Generally, for KWS in a relational database, an inverted index is needed which stores mapping of database terms to its schematic location in the database. We will refer to this inverted index in the report as keyword index. Previous KWS systems [1, 3, 5, 8] have proposed different types of inverted index structures to store term related information

but these are either main memory data structures and used at the application level, or as relational tables. In both the cases, DBMS engine is not aware of keyword index. So we have proposed implementing keyword index inside RDBMS, to benefit various stages of KWS and for efficient access of relations involved in KWS processing. In particular, we have proposed a new trie based keyword index which stores term (keyword) related information such as schematic location of term in the database, frequency of term at cell level granularity etc. This keyword index is used to speed-up whole KWS processing. Trie [15] is supposed to be an effective main memory data structure for look up operation on string data. We have used trie for storing all distinct terms in the database. We have implemented trie as a disk based data structure using SP-GiST (Space Partitioned Generalized Search Tree) [2] framework of PostgreSQL which provide a general framework for implementing the class of space partitioning trees like trie, k-D tree, quadtree etc. with inbuilt common functionalities and optimizations.

1.1 Keyword Search in Relational Databases

The success of KWS stems from what it does not require - namely, a specialized query language or knowledge of the underlying structure of the data. But for KWS in relational databases, the user requires writing SQL query and schema of the underlying data, even to pose simple query. So an easy to use KWS interface to RDBMS is provided which does not require SQL and schema information.

Let us consider a sample Bibliography database (Table 1.1 and 1.2) having relational tables Paper(id, title, con_id), and Conference(id, name, year).

Suppose if we want information about recent work on '*Keyword Search*' in relational database published in '*SIGMOD*' conference, with SQL query language interface, we need to construct SQL query like Figure 1.1 and corresponding answer published would be like Table 1.3. But usage of KWS interface requires only to type '*Keyword Search SIGMOD*' on text box provided for searching and answers are published like in Table 1.3.

id	title	con_id
P1	Keyword Search over Relational Databases: A Metadata Approach	C2
P2	Optimizing Index for Taxonomy Keyword Search	C1
P3	Scalable Top-K Spatial Keyword Search	C4
P4	Keymantic: Semantic Keyword based Searching in Data Integration Systems	C3

Table 1.1: Sample Paper Relation

id	name	year
C1	SIGMOD	2012
C2	SIGMOD	2011
C3	VLDB	2010
C4	EDBT	2013

Table 1.2: Sample Conference Relation

```
SELECT *
FROM   Paper, Conference
WHERE  Paper.con_id = Conference.id AND
       Paper.title LIKE %Keyword Search% AND
       Conference.name LIKE %SIGMOD%
```

Figure 1.1: Example SQL query

id	title	con_id	name	year
P1	Keyword Search over Relational Databases: A Metadata Approach	C2	SIGMOD	2012
P2	Optimizing Index for Taxonomy Keyword Search	C1	SIGMOD	2011

Table 1.3: Result for SQL query

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, information is split into multiple

relational tables. For a given keyword query, KWS interface tries to match the terms with the attributes in the underlying database schema, thus producing a set of candidate SQL queries. As the number of potential candidate queries might be high, a ranking model is used to calculate a score value that expresses the likelihood of each candidate SQL query corresponding to the original keyword query. Only the highest scored SQL query is executed by DBMS engine. This SQL query can potentially retrieve large sets of candidate results where a candidate result is obtained by joining related relational tuples which, as a whole, contain all the keyword terms. KWS interface ranks the query results by using another ranking model that evaluates the likelihood of a query result satisfying the original keyword query. Then KWS interface returns only top-k (for some fixed constant k) highest scored query results to the user. Whole process is also summarized in Figure 1.2.

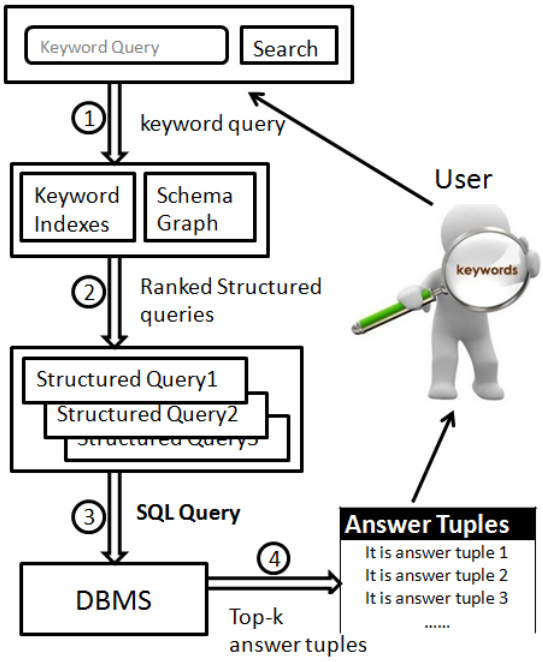

Figure 1.2: KWS Processing

1.2 Database Models for KWS

For KWS in relational databases, two database models are considered prominently in the literature.

Schema graph based KWS models uses schema graph of the database for information about relationship between set of relational tables. The relationship between relational tables could be due to primary key - foreign key or by user(DBA) defined relationships.

Data graph based KWS models uses data graph of the published database, which represents relationships between published relational tuples (nodes), and use it for generating answers for keyword queries.

We have not considered data graph based KWS model because no system in this category shows either better and high performance compare to schema based system or is free from various non-realistic limitations (memory limitation, bad performance evaluation, does not generalizes to real dataset etc) [11].

1.3 DBLabrador

In this report we explore the DBLabrador [1] system and inverted indexes used for KWS in DBLabrador. DBLabrador follows schema graph based approach for KWS in relational databases. The entire KWS processing performed by DBLabrador is done through indexes which are stored as relations and series of SQL queries. Still DBLabrador does not provide the required execution time performance which is required for keyword search systems. Hence we propose implementing the keyword indexes inside RDBMS. This disk resident keyword index not only preserves the scalability of the KWS system w.r.to large database sizes but also speed-up the KWS processing.

1.4 Contributions

We have designed and implemented a trie based keyword index inside PostgreSQL to speed-up whole KWS processing. Previous work did not look into implementation of keyword index inside the database server. It is the first work which implements the keyword index inside the database server and thus providing a useful index for performing keyword search.

Experiments were conducted to measure usage of proposed keyword index and comparing this new system with DBLabrador system on various representative real datasets and query workload. Experiments reveal new keyword index speed up the KWS process and provide benefits at different steps of KWS.

1.5 Organization

The rest of the report is organized as follows: In Chapter 2, first we formulate our problem of designing and using inbuilt keyword index in KWS processing, then we discuss proposed trie based design of keyword index. Chapter 3 discusses implementation of trie based keyword index using SP-GiST framework. Then Chapter 4 covers current and past related work. In Chapter 5, we discuss experimental setup used for comparing our system with DBLabrador [1] and results obtained. And Chapter 6 concludes the paper with some future directions to work on.

Chapter 2

Proposed Design of Keyword Index

2.1 Need of Keyword Index

Given a database instance, schema graph and set of attributes of database to be published, keyword indexes are required to store schematic location and frequency information of terms present in these attributes. Now for a given keyword query $KQ = \{k_1, k_2, \ldots, k_n\}$ where k_i, i=1 to n, is a term given by the user, these keyword indexes are required to benefit various stages of KWS processing which is as follows:

- For each term $k_i \in KQ$, identify the set of attributes containing k_i (Let this set be denoted by a_i).
- A **structured query** is a form of query where each keyword term is associated with an attribute of the database.
- Generate all possible candidate structured queries by performing Cartesian product of a_i, i=1 to n.
- Each structured query is checked for validity using schema graph of the database.
- Relevance score for each candidate structured query is calculated using Bayesian network model [3] which uses term's column granularity frequency information at database instance level.

- Structured query with highest score is chosen for SQL construction. FROM clause of SQL query is constructed by making natural join of relations involved in chosen structured query (let it be $\{r_1, r_2, \ldots\}$). For example, SQL query for structured query $< k_1{:}a_1,\ k_2{:}a_2, \ldots, k_n{:}a_n >$ is

  SELECT $*$

  FROM r_1 natural join r_2 ...

  WHERE $a_1\ LIKE\ k_1\ AND\ a_2\ LIKE\ k_2$

  $\ldots AND\ a_n\ LIKE\ k_n$

- Relevance score for each result tuple of selected query is calculated using Bayesian network model [3] which uses column granularity frequency information at query result set level.

KWS processing could be benefited by keyword index in initial stage for identifying attributes which contain keyword terms, as keyword index stores schematic location (table, column and row identifier) information of the terms. Next stage where keyword index is utilized is ranking of candidate structured queries, as keyword index stores term's column granularity frequency information. Finally keyword index can benefit execution of chosen structured query, as it provides direct access to tuples containing keyword terms.

Query optimizer chooses some execution plan for SQL query submitted for selected candidate structured query. In previous KWS approaches, keyword indexes are explicitly specified and/or used in the SQL query for efficient execution. But we want to make query optimizer aware of keyword indexes so that this keyword index access path is also considered for structured query.

2.2 Proposed Design

We have proposed a trie based keyword index for storing string term and term's information. A trie [15] is a tree in which the branching at any level is determined by only a

portion of the string term (i.e. character). Table 2.1 show a sample course relation and Figure 2.1 shows trie based keyword index for some terms in course relation.

Course_no	Course_name
1	Database Management System
2	Data Mining
3	Operating System
4	Spatial Databases

Table 2.1: Sample Course Relation

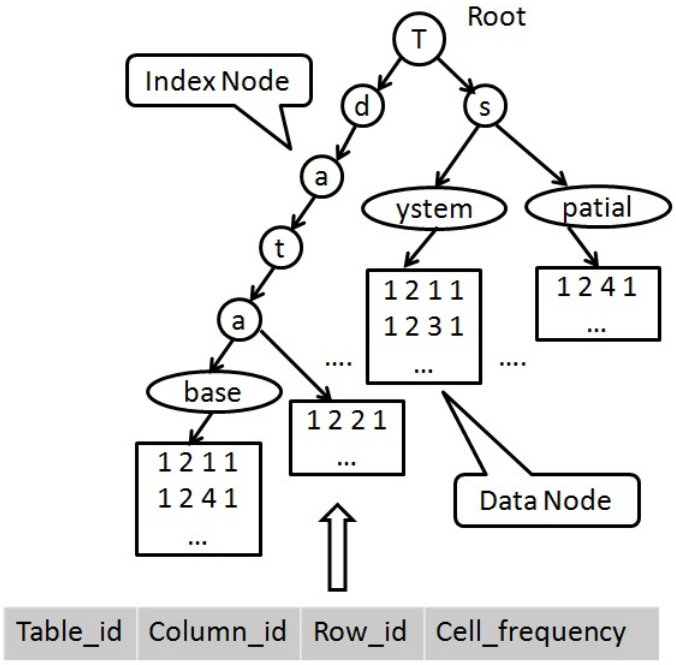

Figure 2.1: Trie Based Keyword Index

The trie contains two types of nodes; index nodes and data nodes. Index nodes are internal nodes which is used for branching, while data nodes are used for storing auxiliary information corresponding to string represented by root to leaf nodes. In the Figure 2.1, index nodes are represented by circles, while data nodes are represented by rectangles. All characters in the term are assumed to be one of the 26 letters of the English alphabet. A blank is used to terminate a term. At level 1, all terms are

partitioned into 27 disjoint classes depending on their first character. Thus, Link(T, i) points to a subtrie containing all terms beginning with the i^{th} character (T is the root of the trie). On the j^{th} level the branching is determined by the j^{th} character. When a subtrie contains only one character, it is replaced by a node of type data. This node contains the list of 4-tuple information corresponding to term represented by root to leaf: <table_id, column_id, row_id, cell_frequency>. Here (table_id, column_id, row_id) represents schematic location (or cell_id) of the term in the database and cell_frequency is the frequency of term at cell level granularity.

We has used trie for storing all distinct terms in the database. Trie is supposed to be an effective main memory data structure for look up operation on string data. In a trie, worst case time complexity of lookup/insert/delete operation for a string S of length l is O(l). Also, during lookup, simple operations such as array indexing using a character, are used which are fast on real machines. Tries are more space efficient when they contain a large number of short keys, since nodes are shared between keys with common initial subsequences.

Trie is a unbalanced tree structure and can be slower in some cases than hash table for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory. Also for each term it stores auxiliary information which could be large in size, causing more skew in the tree structure. So, a crucial issue for database systems, in using trie, could be that they are not optimized for I/O. But for overcoming these limitations we have implemented trie as a disk based data structure using SP-GiST (Space Partitioned Generalized Search Tree) [2] framework of PostgreSQL which provide node clustering algorithm that achieves minimum height and hence minimum I/O access.

2.3 Keyword Index Construction

For constructing trie based keyword index for a given database (and set of attributes to be published), we first scan database to be published, once. This involves scanning each

string attribute value(cell) in the published database, splitting string into terms, storing cell-id, distinct terms and corresponding frequency in a relation named:

DB_String_Terms(table_id integer, column_id integer, row_id integer, term text, frequency integer).

Then we create keyword index using our proposed index design on term attribute of DB_String_Terms relation. This relation, or in other words, this keyword index is used as a global inverted index for structured queries generation, their ranking, and efficient execution of highest ranked structured query.

Chapter 3

Implementation of Trie based Keyword Index

We have used PostgreSQL 9.2 for implementing our index structure. Our trie based keyword index is a unbalanced structure as opposed to traditionally used balanced index structures such as B-tree, R-tree and its variants. One of the major hurdles in implementing such a nontraditional index inside a database engine is the implementation overhead associated with realizing and integrating this index inside the engine. Hard wiring the implementation of a fully functional index structure with the appropriate concurrency and recovery mechanisms into the database engine is a non-trivial process. We have implemented trie based keyword index using a general index framework, SP-GiST (Space Partitioned Generalized Search Tree) [2] provided by the PostgreSQL 9.2.

3.1 SP-GiST

SP-GiST supports the class of space-partitioning trees, e.g. tries, quadtrees and k-D trees. The quadtree uses recursive decomposition of space into quadrants and used for indexing point data, rectangles, and polygonal data. While k-D trees are useful for answering range queries about a set of points in the k-dimensional space. Tries are effective for look up operation on string data. We have used trie for storing all distinct

terms in the database.

SP-GiST provides a set of internal methods that are common for all space-partitioning trees, e.g. the insertion, deletion, and updating algorithms. SP-GiST also provides concurrency control and recovery techniques and I/O access optimization for all space-partitioning trees in a general way.

SP-GiST has to support two distinct types of nodes: index and data nodes. Index nodes (non-leaf nodes) hold the various space partitions at each level. Each entry in an index node is a root of a subtree that holds all the entries that lie in this partition. The space partitions are disjoint. Besides having a slot for each space partition, the index node contains an extra blank slot to point to data nodes attached to the partition represented by this node. On the other hand, data nodes (leaf nodes) hold the key data and other pointer information to physical data records. We can think of data nodes as Buckets of data entries.

The SP-GiST core requires that inner and leaf nodes fit on a single index page, and also the list of leaf tuples reached from a single inner node, all be stored on the same index page. Restricting such lists to not cross pages reduces seeks.

3.2 SP-GiST Parameters and Methods

To handle the differences among the various SP-GiST based indexes, SP-GiST provides a set of interface parameters and a set of external method interfaces.

3.2.1 Interface Parameters

- **PathShrink**: This parameter specifies how the index tree can shrink at leaf level and internal level. It is useful in limiting the number of times the space is recursively decomposed in response to data insertion. PathShrink takes one of three possible values: *NeverShrik, LeafShrink* and *TreeShrink*. *LeafShrink* implies no index node will have single leaf node while in case of *TreeShrink* internal nodes are merged together to eliminate all single child internal nodes.

- **NodeShrink**: A Boolean parameter that specifies whether the empty partitions should be kept in the index tree or not.
- **NumberofSpacePartitions**: This parameter specifies the number of disjoint partitions produced at each decomposition.
- **NodePredicate**: This parameter specifies the predicate type at the index nodes.
- **KeyType**: This parameter specifies the data type stored at the leaf nodes.
- **BucketSize**: This parameter gives the maximum number of data items a data node can hold.

For example, to instantiate our trie based structure we have set these parameters as follows:

- PathShrink = LeafShrink
- NodeShrink = False
- NumberOfSpacePartitions = 26
- NodePredicate = letter or Blank
- KeyType = String
- BucketSize = 1 (i.e. only one string's related information is stored in a data node)

3.2.2 External Methods

The SP-GiST external methods include the method PickSplit() to specify how the space is decomposed and how the data items are distributed over the new partitions (Figure 3.1).

Here P is a set of BucketSize+1 entries that cannot fit in a node. PickSplit() defines a way of splitting the entries into a number of partitions equal to NumberOfSpacePartitions and returns a Boolean value indicating whether further partitioning should take place or

PickSplit(EntrySet P, Level $level$)

1. Partition the data strings in P according to the character values at position level
2. If any data string has length < level,
   insert data string in Partition blank
3. If any of the partitions is still over-full
   return True;
   else
   return False;

Figure 3.1: PickSplit method for partitioning

not. The parameter level is used in the splitting criterion because splitting will depend on the current decomposition level of the tree. For example, in a trie of English words, at level i, splitting will be according to the i^{th} character of each word in the over-full node. PickSplit() will return the entries of the split nodes in the output parameter splitnodes, which is an array of buckets, where each bucket contains the elements that should be inserted in the corresponding child node. The predicates of the children are also returned in splitpredicates. PickSplit() is invoked by the internal method Insert() when a node-split is needed.

Another external method is the Consistent() which specifies how to navigate through the index tree (Figure 3.2).

Consistent(Entry E, QueryPredicate q, Level $level$)

1. If (q.level == E.letter) OR (E.letter == BLANK AND level > length(q))
   return True;
   else
   return False;

Figure 3.2: Consistent method for trie navigation

Let E: (p, ptr) be an entry in an SP-GiST node, where p is a node predicate or a leaf data key and ptr is a pointer. When p is a node predicate, ptr points to the child node corresponding to its predicate. When p is a leaf data key, ptr points to the data record associated with this key. Consistent() is invoked by the internal methods Insert() and Search() to guide the tree navigation.

The various SP-GiST index structures have different sets of operators to work on. For the trie index structure, we define the operator =, to support the equality queries (Figure 3.3).

```
CREATE OPERATOR =
  leftarg = VARCHAR,
  rightarg = VARCHAR,
  procedure = trieword_equal,
  commutator = =,
  restrict = eqsel
```

```
CREATE OPERATOR CLASS
SP_GiST_Trie
  FOR TYPE VARCHAR
  USING SP_GiST
  AS OPERATOR 1 =,
  FUNCTION 1 trie_consistent,
  FUNCTION 2 trie_picksplit
```

Figure 3.3: Operator and Operator class for trie

3.3 Node Clustering in SP-GiST

SP-GiST provides different node clustering methods which can be used according to type and nature of the operations to be performed on the constructed index. We have used SP-GiST's default node clustering algorithm [6] that achieves minimum height and hence minimum I/O access. Other clustering alternatives are Fill-Factor clustering which mainly focuses on space utilization of index structure, Deep clustering which chooses the longest linked subtree from the collection of nodes to be stored together in the same page and Breadth clustering which chooses the maximum number of siblings of the same parent to be stored together in the same page.

Chapter 4

Literature Survey

Goldman et al. [5] had first recognized the usefulness of a keyword search system for databases, but significant research in this area did not commence for several years after these early publications. Interestingly, these proposed systems coincide with early formal attempts to integrate Database and IR technologies.

[1] has used data graph based KWS model, in which distinct root semantic answer model is considered and an alternative keyword index, Node-Node index (instead of Node-Keyword index [7]) has introduced, to solve issues related to storage space of keyword index but query search time needs to be compromised with less storage requirement. BANKS [8], BANKS II [9], BLINKS [10] are the earlier data graph based models.

DBXplorer [12] and Discover [13] are among the first Schema graph Based systems proposed in this category. DBXplorer [12] used a symbol table for mapping terms to tuples and explores various design alternatives of this inverted index. Discover [13] has used a greedy algorithm to prioritize candidate networks (which are equivalent of Structure Query) which give fewest results. Both [12] and [13] has considered no. of join in selected candidate network for ranking results.

In terms of KWS processing, we have followed Labrador's approach [3]. For a keyword query, Labrador generates ranked candidate structured queries, using column granularity term frequency hash map. User can choose one of the preferred structured queries, 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. But Labrador uses main memory data structures (e.g. column granularity term frequency hash map), for various intermediate calculations and KWS processing, and RDBMS is used as data repository only. So DBLabrador [1] have been proposed which is functionally similar to Labrador, and uses RDBMS to perform all computations. 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. Advantages of using RDBMS back-end technologies are, it can handle large databases and KWS indexes are persistent now. Although these keyword indexes (stored as relation table) are present in RDBMS, they are not a part of execution plan of structured queries, i.e. these keyword indexes are very efficient access path for relations used in a structured query, but this fact would not be utilized implicitly by RDBMS.

One important processing step of KWS in relational databases is the ranking of search results which is obtained after joining relations in selected structured query (Top-k Query execution). This Top-k query execution depends on result set dependant ranking function i.e. ranking function uses frequency of terms at result set level and not at original database instance level [3]. Final score of a result (tuple) is aggregated from multiple scores of each constituent tuples, but the final score is not monotonic with respect to any of its subcomponents. Several efficient query execution algorithms optimized for returning Top-k relevant results are presented in [4]. But existing work on top-k query optimization cannot be immediately applied as they all rely on the monotonicity of the score aggregation function.

Coffman [11] addresses the challenges inherent in transitioning relational keyword search techniques from the computer science community to practical systems that can be deployed against existing data repositories. It also presents an extensive benchmark specifically designed to evaluate relational keyword search techniques via published datasets, query workloads, and relevance assessments.

Chapter 5

Experiments

In this section, we compare KWS system using our trie based keyword index and DBLabrador [1] KWS system, on end to end runtime performance basis i.e. time elapsed between submission of keyword query and ranked result tuples are generated.

5.1 Experimental Setup

All experiments are conducted in PostgreSQL 9.2 on Intel(R) Core(TM) i3-2310M CPU 2.10GHz, 3GB Main memory, Ubuntu 12.04 operating system.

5.1.1 Datasets

We have used following representative datasets (Table 5.1). Among these 10DBLP and 100DBLP are the scaled version of the DBLP dataset. We have replicated the tuples in DBLP dataset by the scale factor 10 and 100 for generating this scaled version datasets. These scaled version datasets are used for testing scalability of the index. These datasets covered different dimensions of real world scenarios hence considered as benchmark datasets for testing KWS systems [11] in current literature.

- **DBLP** : Digital Bibliography & Library Project (DBLP) is one of the popular dataset used in previous KWS systems. This dataset contains paper titles, their authors and other bibliographic information on major computer science publications

Dataset	Size(MB)	Relations	Text attributes	Tuples
DBLP	688	9	25	7357K
10DBLP	6890	9	25	735M
100DBLP	68920	9	25	7357M
Wikipedia	550	42	121	206K
Mondial	9	28	70	17K

Table 5.1: Datasets

extracted from the DBLP repository (Table 5.2). Schema graph of the published database is shown in Figure 5.1.

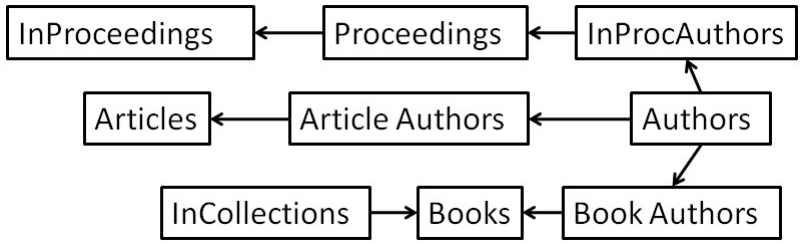

Figure 5.1: DBLP schema graph

- **Wikipedia** : This includes more than 5500 articles chosen for the 2008 - 2009 Wikipedia Schools DVD, a general purpose encyclopedia, which contains content roughly equal to a traditional 20 volume encyclopedia.

- **Mondial** : The Mondial dataset [14] comprises geographical and demographic information from the CIA World Factbook, the International Atlas, the TERRA database, and other web sources.

Among these datasets we have given details (relations, schema graph etc.) about one of the dataset i.e. DBLP (Table 5.2).

Relation	Size(MB)	Tuples	# Distinct Terms
Proceedings(id, key, title, year)	4.3	17,982	30,558
InProceedings(id, key, prockey, title, year)	206	1,112,035	1,353,575
InProcAuthors(inProcId, name)	187	3,155,373	291,477
Articles(id, key, title, journal, year)	154	826,026	1,022,410
ArticleAuthors(articleId, name)	125	2,101,494	260,209
Books(id, key, editor, publisher, year)	1.16	9,286	19,634
BookAuthors(bookId, name)	0.86	13,415	9,404
InCollections(id, key,bookKey, title, year)	4.5	22,582	34,034

Table 5.2: DBLP Dataset

5.1.2 Keyword Queries

We have considered 50 keyword queries for each dataset. No. of query terms varies from 1 to 4. We have shown here some representative keyword queries used in experiments in Table 5.7, 5.6, 5.3 for different datasets which reflects runtime performance of all considered queries.

5.1.3 PostgreSQL parameters

We have set Postgres parameters *shared_buffers* and *work_memory* according to size of dataset and size of keyword index. These parameters are kept lower than size of keyword index so that usage of this disk resident index can be evaluated. For Mondial dataset which is of smaller size (9MB), keyword index is of size 1.5MB and we have set shared_buffers = 128KB and work_mem = 512KB. For DBLP and Wikipedia datasets keyword index sizes are 8MB and 33MB respectively, so we have used shared_buffers and work_mem as 2MB for these datasets. For scaled versions of DBLP dataset, 10DBLP and 100DBLP keyword index sizes are 82MB and 886MB respectively and we have used shared_buffers and work_mem as 2MB.

5.2 Experimental Results

In all considered keyword queries, we found our approach has performed better than DBLabrador approach (Figure 5.6, 5.5, 5.2) for all datasets. One reason behind this improvement is trie based keyword index save processing such as aggregate and group by operation by pre-calculating and storing various information for each term in the database.

Also for testing scalability of the index usage, we have experimented with the larger datasets(10DBLP, 100DBLP), which although not changes the trie index's internal structure but scales the leaf levels. In these cases also, our trie based index performs better than DBLabrador(Figure 5.3, 5.4). Note that total time for these datasets are in minutes.

From experimental results(Figure 5.6, 5.5, 5.2), its clear that our trie based system got more benefit and improvement in total time, when more no. of joins are involved.

Although we got high total time values in the experiments if we look at absolute scale, but this is obtained under restricted memory limits, i.e. working memory is set to very small fraction of trie index size. As this working memory is increased, total time performance improves for both the system, but relative nature of results we have got is not changed much. Further if working memory is set to a value such that this trie based keyword index can be fit into it completely while KWS processing, then its performance and benefits gained are much more.

For Wikipedia and Mondial datasets, result set cardinalities are not much but this also represents a real world scenario where few result tuples are to be retrieved. Specifically Mondial datasets size is very small and most of the information stored in it are geographical facts and hence database information is not redundant and we have not got similar benefit in case of more no. of joins are involved.

no.	Keyword Query	#result tuples	#relations involved
1	network model	856	1
2	computer science	766	1
3	oliver generalized optimization	1532	2
4	approximation algorithm	1344	2
5	cooperative game theory	1192	2
6	eigenvalue calculation	566	3
7	classification data mining	1688	3
8	probability dimitri	764	3
9	multidimensional histogram usage	35	4
10	simulation software update	62	4

Table 5.3: Keyword Queries for DBLP Dataset

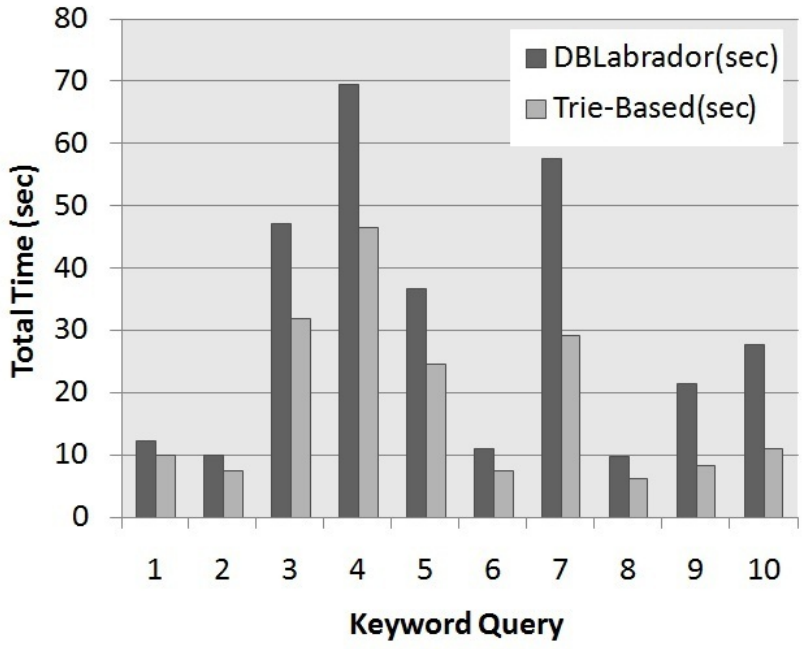

Figure 5.2: Comparing Total running time (DBLP)

no.	Keyword Query	#result tuples	#relations involved
1	network model	8560	1
2	computer science	7660	1
3	oliver generalized optimization	152K	2
4	approximation algorithm	129K	2
5	cooperative game theory	115K	2
6	eigenvalue calculation	560K	3
7	classification data mining	16M	3
8	probability dimitri	760K	3
9	multidimensional histogram usage	36K	4
10	simulation software update	60K	4

Table 5.4: Keyword Queries for 10DBLP Dataset

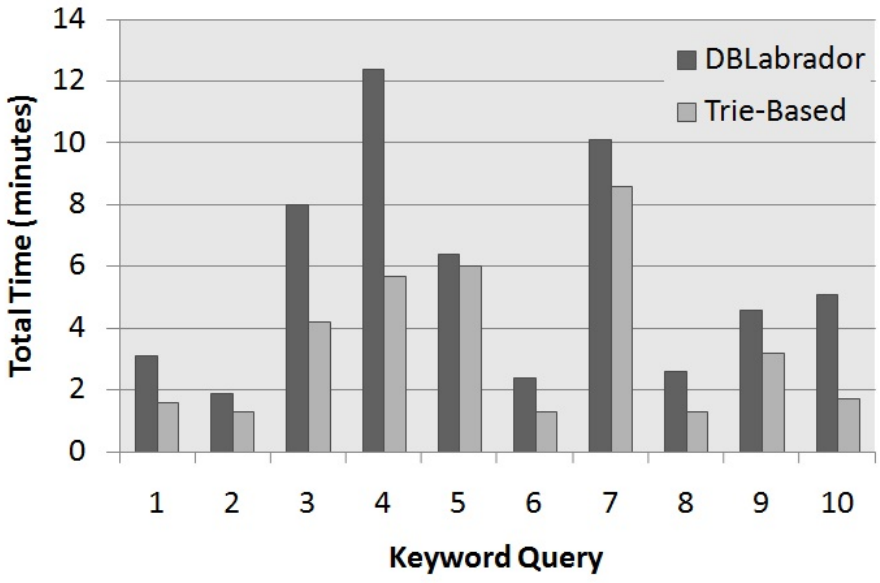

Figure 5.3: Comparing Total running time (10DBLP)

no.	Keyword Query	#result tuples	#relations involved
1	network model	85K	1
2	computer science	76K	1
3	oliver generalized optimization	15M	2
4	approximation algorithm	13M	2
5	cooperative game theory	11M	2
6	eigenvalue calculation	56M	3
7	classification data mining	168M	3
8	probability dimitri	76M	3
9	multidimensional histogram usage	3M	4
10	simulation software update	6M	4

Table 5.5: Keyword Queries for 100DBLP Dataset

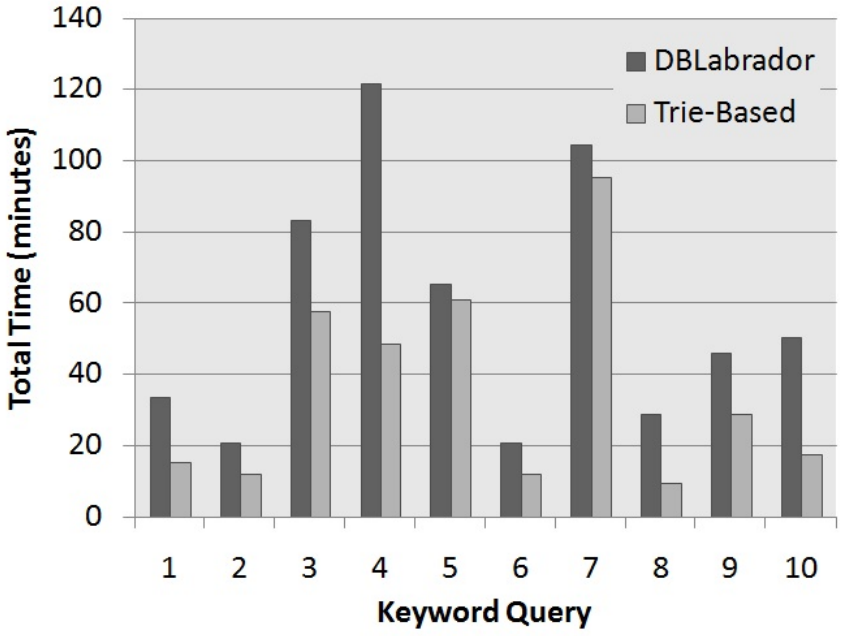

Figure 5.4: Comparing Total running time (100DBLP)

no.	Keyword Query	#result tuples	#relations involved
1	microscope	1	1
2	malwa madhya pradesh	1	1
3	tianjin china district beijing	1	3
4	european hawfinch	1	3
5	separation of powers	1	1
6	zelda series	1	1
7	irrational number	13	3
8	mona lisa artist	8	3
9	page AlleborgoBot	13	3
10	international development world bank	1	3

Table 5.6: Keyword Queries for Wikipedia Dataset

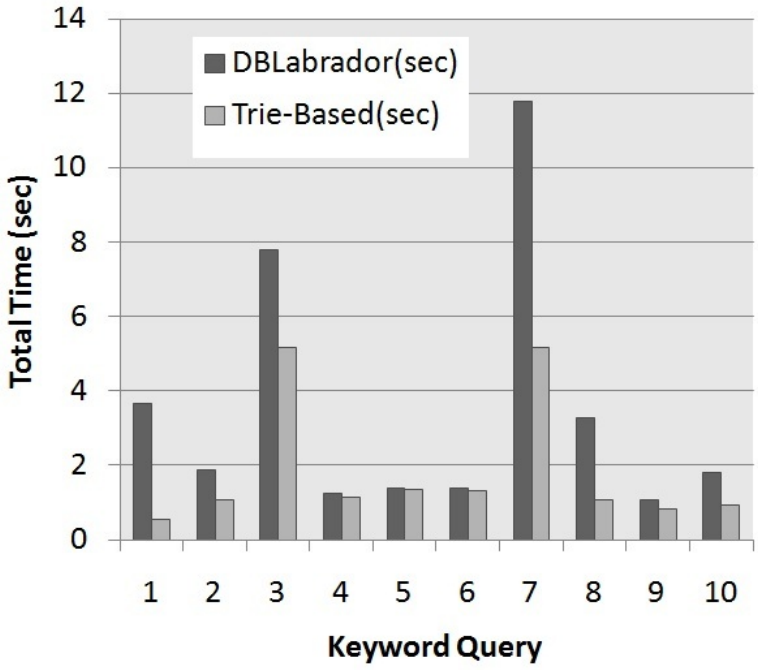

Figure 5.5: Comparing Total running time (Wikipedia)

no.	Keyword Query	#result tuples	#relations involved
1	thialand	1	1
2	alexandria	1	1
3	arabian sea	1	1
4	world labor	1	1
5	cameroon economy	1	2
6	poland language	1	2
7	marshall islands grenadines organization	9	5
8	guyana sierra leone	31	5
9	mauritius india	35	5
10	egypt nile	5	4

Table 5.7: Keyword Queries for Mondial Dataset

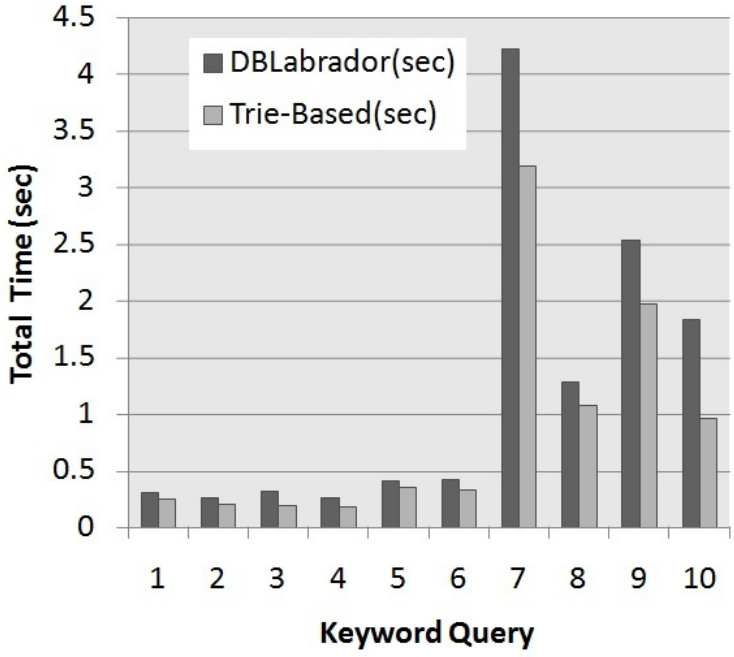

Figure 5.6: Comparing Total running time (Mondial)

Chapter 6

Conclusions

We have considered the problem of designing and implementing keyword index inside RDBMS for speeding up KWS in RDBMS. We have proposed a trie based keyword index for storing string term and term's information. Using SP-GiST framework of PostgreSQL, this trie based index is realized and used for KWS, for various representative datasets and query workload. Experiments are performed to consider different real world scenarios, scalability of the index usage etc. By using trie based keyword index for KWS in RDBMS, we get better performance compared to DBLabrador [1] in all the cases. Also our trie based system got more benefit and improvement in total time, when more no. of joins are involved.

In the future, we tune keyword index for more benefits on various parameters like, top-k ranking of result tuples, memory utilization and node clustering etc. Also we try to do cost modelling of the proposed index structure.

References

[1] Karthik Somayaji, "Hosting keyword search engine on RDBMS", *Master's Thesis, Dept. of Computer Science & Automation, IISc Bangalore*, `http://dsl.serc.iisc.ernet.in/projects/KWS/index.html`, 2012.

[2] Walid G. Aref and Ihab F. Ilyas, "SP-GiST: An extensible database index for supporting space partitioning trees", *Journal of Intelligent Information Systems*, 2001.

[3] Filipe Mesquita, Altigran S. da Silva, Edleno S. de Moura, Pavel Calado and Alberto H. F. Laender, "LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces", *Information Processing and Management*, Vol. 43, 2006.

[4] Ihab F. Ilyas, George Beskales, Mohamed A. Soliman, "A Survey of Top-k Query Processing Techniques in Relational Database Systems", *ACM Computing Surveys*, Vol. 40, 2008.

[5] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina, "Proximity search in databases", *VLDB*, 1998.

[6] A. A. Diwan, S. Rane, S. Seshadri and S. Sudarshan, "Clustering Techniques for Minimizing External Path Length", *VLDB*, 1996.

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

[8] S. Sudarshan, Bhalotia Gaurav, and Nakhe Charuta, "Keyword searching and browsing in databases using BANKS", *ICDE*, 2002.

[9] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar, "Bidirectional expansion for keyword search on graph databases", *VLDB*, 2005.

[10] Hao He, Haixun Wang, Jun Yang and Philip S. Yu, "BLINKS: Ranked keyword searches on graphs", *SIGMOD*, 2007.

[11] Joel Coffman and Alfred C. Weaver, "A framework for evaluating database keyword search strategies", *CIKM*, 2010.

[12] Sanjay Agrawal, Surajit Chaudhuri and Gautam Das, "DBXplorer: A system for keyword-based search over relational databases", *ICDE*, 2002.

[13] Vagelis Hristidis and Yannis Papakonstantinou, "Discover: Keyword search in relational databases", *VLDB*, 2002.

[14] W. May, "Information Extraction and Integration with Florid: The Mondial Case Study", *Technical Report 131, University at Freiburg, Institute for Informatik*, `http://dbis.informatik.uni-goettingen.de/Mondial`, 1999.

[15] Donald Knuth, "Digital Searching", *The Art of Computer Programming Volume 3: Sorting and Searching (2nd ed.), Addison-Wesley*, 1997.