RDBMS Operators for Ranking of Keyword Search Results

Vinay M S Jayant R. Haritsa

Technical Report
TR-2015-03

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

Abstract

Information Retrieval(IR) systems and RDBMS have remained largely disjoint. RDBMS has been used as a black box to store and obtain data on which IR scoring functions are applied to provide relevant answers to the users. Keyword Search on RDBMS is a classic example of IR systems which use RDBMS as a black box. Recently, result ranking in Keyword Search systems have been performed through SQL queries [7] and it has achieved loose coupling at the schematic level between RDBMS and Keyword Search systems.

Schema integration of *result set dependent* scoring functions, such as Labrador [1], is inefficient. The result set dependent property necessitates the creation of on-the-fly indexes to perform scoring which can degrade performance if the indexes are stored on the disk. However, in this work we will show that result set dependent scoring functions can be efficiently implemented through first class RDBMS operators. These new operators achieve tight coupling between RDBMS and IR systems.

In this work, the benefits of Labrador scoring function [1] wrt user relevance are demonstrated empirically. Three different techniques to integrate result ranking of Keyword Search systems inside RDBMS using Labrador scoring [1] function are proposed and evaluated. The first technique called SQL-Wrapper achieves schema integration. The second, called the Root Rank introduces first class operator inside RDBMS to perform the result ranking task. This operator is introduced at the root position of the plan tree. Finally, the third technique called the Join Rank performs the same task of result ranking but inside the top-most join node of the plan tree. These new operators are implemented and incorporated inside PostgreSQL(9.1.2). Accurate cost models have been developed to achieve seamless integration with the Query Optimizer. The cost models have been validated to obtain good correlation with actual run time costs. Empirical results over real data sets exhibit substantial performance gains of Root Rank and Join Rank operators over SQL-Wrapper.

1 Introduction

Keyword search on RDBMS has been an active area of research for over a decade due to the critical need of querying over relational systems through the World Wide Web [9]. Keyword search systems are classified as Candidate Network(CN) models and Data Graph models. In the quest of performing efficient keyword search on RDBMS, new graph search algorithms [21] and innovative systems [3] have been developed. It has also initiated efforts to couple RDBMS and Information Retrieval(IR) systems [7].

1.1 Candidate Network Systems

The CN model system was first proposed in DISCOVER [3]. The term Candidate Network refers to a joined network of relations whose result set provides answer to the keyword query. The result set of CN is a collection of tuple trees. Each row of the result set is formed by connecting various tuples of different relations. Hence, it is viewed as a tuple tree. Consider a keyword query *Data Mining* applied on DBLP dataset. For which a possible CN might be, $\mathsf{Proceeding.title}^{\mathsf{Data}} \times \mathsf{InProceeding.title}^{\mathsf{mining}}$. This means that, *Data* term is mapped to Proceeding relation wrt title attribute and *Mining* term is mapped to InProceeding relation wrt title attribute. The keyword query is answered by result set of the join of both these relations. This is a very basic definition and it needs to satisfy additional properties in order to be called a CN:

1. The CN has to be complete which means that each tuple tree $\in$ CN should contain all the terms in the keyword query.

2. The CN has to be minimal which means that if any relation is removed from the CN, it should not provide the required answer for the keyword query.

Scoring functions are applied on the tuple trees to provide top-K ranked tuple trees. Many scoring functions have been proposed in the literature. All the CN scoring functions are analyzed below to explore their structure and to find out why schema integration of these scoring functions with the singular exception of Labrador [1] are sufficient to couple them with RDBMS.

1.1.1 Efficient [20]

Discover system [3] was modified to include IR style keyword ranking mechanism. This system, is denoted as Efficient [20]. A new scoring function which includes the rich IR properties was adapted inside the Discover system [3] to perform scoring on Candidate Network result sets. Consider a tuple tree $T \in$ Candidate Network(CN). Each text attribute a_i of the tuple tree where the keyword gets mapped is scored by the following equation.

$$score(a_i, Q) = \sum_{w \in Q \cap a_i} \frac{1 + log(1 + log(tf))}{(1 - s) + s\frac{dl}{avdl}} * log\frac{N + 1}{df}$$

The variable Q indicates the keyword query, tf indicates the term frequency of the word w in a_i, dl is the size of a_i in characters, $avdl$ is the average attribute value size, N is the number of tuples in relation which contains a_i, s is a constant. The scoring function is given by $Finalscore(T)$ in Equation 1.

$$\boxed{Finalscore(T) = \frac{\sum_{a_i \in T} score(a_i, Q)}{size(T)}} \quad (1)$$

Let T,T' be the tuple trees of a given CN. Let a_1, a_2...a_n be the attributes of T and let b_1, b_2.....b_n be the attributes of T'. The scoring function is said to have *tuple monotonicity property* if it satisfies the below condition.

If $score(a_i, Q) \leq score(b_i, Q)\ \forall\ i$ then, $Finalscore(T, Q) \leq Finalscore(T', Q)$. The tuple monotonicity property is an essential requirement to develop time optimal [20] algorithms for providing top-K answers and it is satisfied by Equation 1.

1.1.2 Effective [28]

Effective is another IR style keyword search scoring function on Candidate Networks.

$$weight(k, D_i) = \frac{ntf * idf^g}{ndl * Nsize(T)}$$

$$weight(k, T) = comb(weight(k, D_1), weight(k, D_2)....., weight(k, D_m))$$

For a given tuple tree $T \in CN$, k is the keyword query term, D_i is an attribute to which term k is mapped. Every text attribute D_i is considered as a collection of documents wrt its base relation.

$$ntf = 1 + log(1 + log(tf))$$

$$Nsize(T) = (1-s) + s * \frac{size(T)}{avgsize}$$

$$ndl = ((1-s) + s + \frac{dl}{avgdl})(1 + log(avgdl))$$

$$idf^g = log(\frac{N^g}{df^g + 1})$$

Frequency of the term k in the attribute $D_i \in T$ is given by tf, s is a tunable constant, $Size(T)$ gives the number of relations in T, $Avgsize$ is the average size of all the trees that are generated for the keyword query, dl is the length of the attribute D_i in the tree T, $avgdl$ is the average length of all attributes in tuple tree, N^g is the total number of text columns in the database, df^g is the frequency of the term in the entire database.

$$comb = maxwgt * (1 + log(1 + log\frac{sumwgt}{maxwgt})) \tag{2}$$

The combine function is given by Equation 2, $maxwgt$ is the maximum weight of $weight(k, D_i)\ \forall i$, $sumwgt$ is the sum of the weights $weight(k, D_i)\ \forall i$, $weight(k, Q)$ is the weight of the keyword term k wrt query Q. This is given by the frequency of the term k in the query Q.

$$\boxed{score(Q, T) = \sum_{k \in Q} weight(k, Q) * weight(k, T)} \tag{3}$$

The scoring function in Equation 3 lacks the tuple monotonicity property.

1.1.3 Spark [23]

Spark is a candidate network keyword search system which improves upon the scoring function given in Efficient [20]. The scoring function $score(T, Q)$ in Equation 4 has 3 components.

$$\boxed{score(T, Q) = score_a(T, Q) * score_b(T, Q) * score_c(T, Q)} \tag{4}$$

Here T is an tuple tree $\in CN$ and Q is the keyword query.

$$score_a(T, Q) = \sum_{w \in T \cap Q} \frac{1 + log(1 + log(tf_w(T)))}{(1-s) + s * \frac{dl_T}{avdl_{CN(T)}}} * log(idf_w)$$

$$tf_w(T) = \sum_{t \in T} tf_w(t).$$

Here $tf_w(t)$ indicates the frequency of the term w in the tuple t which $\in$ tree T, s is a tunable constant.

$$idf_w = \frac{1}{1 - \prod_j (1 - p_w(R_j))}.$$

s_1 and s_2 are tunable parameters.

$$score_b(T, Q) = 1 - (\frac{\sum_{1 \leq i \leq m}(1 - T.i)^p}{m})^{\frac{1}{p}}$$

$$T.i = (\frac{tf_{w_i}(T)}{max_{1 \leq j \leq m} tf_{w_j}(T)})(\frac{idf_{w_i}}{max_{1 \leq j \leq m} idf_{w_j}})$$

$$score_c(T, Q) = (1 + s_1 - s_1 * size(CN))(1 + s_2 - s_2 * size(CN^{nf}))$$

Here R_j is a relation which $\in CN$, $p_w(R_j)$ is the percentage of tuples in the relation R_j that contains w, dl_T is the combined length of the text attributes in T, $avdl_{CN(T)}$ is the average combined length of the text attributes that $\in T$ wrt their base relations, m is the number of keywords in the query Q, p is a tuning parameter, $size(CN)$ is the number of relations in CN, CN^{nf} is the number of relations to which keywords are not mapped in CN.

The scoring function of Spark also lacks the tuple monotonicity property.

1.1.4 Labrador [1]

Labrador is again a Candidate Network model system.

For the CN $\mathsf{Proceeding.title^{Data}} \times \mathsf{InProceeding.title^{Mining}}$, the scoring function for ranking the result set tuple trees of the CN is given in Equation 5.

$$score(T) = n * \frac{\cos(p.title, \overrightarrow{p.title}) + \cos(i.title, \overrightarrow{i.title})}{2} \tag{5}$$

$$\cos(p.title, \overrightarrow{p.title}) = \frac{w_{data}}{\sqrt{\sum w_k^2}}$$

$$w_k = log(1 + \frac{total\ tuples\ retrieved}{f_k})$$

$$w_{data} = log(1 + \frac{total\ tuples\ retrieved}{f_{data}})$$

$$\cos(i.title, \overrightarrow{i.title}) = \frac{w_{mining}}{\sqrt{\sum w_r^2}}$$

$$w_{mining} = log(1 + \frac{total\ tuples\ retrieved}{f_{mining}})$$

$$w_r = log(1 + \frac{total\ tuples\ retrieved}{f_r})$$

$score(T)$ is the score for the tuple tree $T \in CN$,
n is a constant,

w_k is the weight of the term $k \in$ p.title of the result set,
w_r is the weight of the term $r \in$ i.title of the result set,
w_{data} is the weight of the term $data \in$ p.title of the result set,
w_{mining} is the weight of the term $mining \in$ i.title of the result set,
f_k is the frequency of the term $k \in$ p.title of the result set,
f_r is the frequency of the term $r \in$ i.title of the result set,
f_{data} is the frequency of the term $data \in$ p.title of the result set,
f_{mining} is the frequency of the term $mining \in$ i.title of the result set.

For a k term keyword query, the scoring function assumes the form given in Equation 6. Here $A_1, A_2.....A_k$ are the attributes on which the keyword query terms are mapped.

$$\boxed{score(T) = n * \frac{\sum_{i=1}^{k} \cos(A_i, \overrightarrow{A_i})}{size(T)}} \tag{6}$$

Number of relations involved in the tuple tree T is given by $size(T)$.

The Labrador scoring function has tuple monotonicity property. It also has a unique challenge of result set dependency. This is because in order to calculate the weight of a term, frequency of the term in the result set of the Candidate Network is required. This forces the result set to be materialized for obtaining term frequency statistics.

The number of Candidate Networks that can be generated grows exponentially with the number of keywords [20]. This makes evaluating these CN extremely expensive. Most of the current systems use a threshold parameter to restrict the number of relations in the CN. The CN system execution procedure involves four steps in providing top-K answers for the keyword query:

1. Generate all the CN by using the threshold parameter through the Candidate Network generator module.
2. Execute each CN to obtain the result set through SQL queries.
3. Apply the scoring function on tuple trees of the result set of each CN.
4. Rank the tuple trees and provide the top-K answers by merging and sorting the result set of all CN.

The ordering of the top-K results may alter with different values of threshold parameter. Ideally the threshold parameter should be set such that large nunber of Candidate Networks are not generated and also the most important Candidate Networks are not left out from the result set. This is again customized for a specific database through empirical validation.

The CN systems have basically 2 components. The first component is CN generation module and the second is result ranking. The first step mentioned in the CN system execution procedure is done by the first component. The remaining three steps are performed by the second component. In the literature [3, 20, 23] performance analysis has been done by including both the components. In the empirical analysis performed in this work by using the DBLP dataset. The CN generation module in most of the cases incurs around 10% of the total cost of providing the answer to the keyword query. Thus it is the result generating component which is significantly costlier and needs particular attention wrt improving performance.

1.2 Data Graph Systems

The Data Graph model systems construct a graph index to store the tuple inter-connectivity of the database. For a given keyword query the graph index is searched to find the potential tuple trees which are also known as Steiner trees [9] that can provide the answer. The task of finding minimal steiner trees is known to be NP-hard [9] and hence approximate solutions

have been proposed [9, 21]. Specially designed scoring functions which consider the link properties in the graph index are applied to rank the steiner trees. Data graph systems, unlike the CN systems, have a single component which is the result ranking component. Many different graph traversal algorithms [7, 9, 21] have been proposed to improve the performance of result ranking component.

1.2.1 Banks [9]

Banks assigns weight to each node of the steiner tree T, $N(v)$ indicates the weight of the node $v \in T$ which is the in degree of the node, $s(R_1, R_2)$ indicates the similarity between the relations R_1 and R_2. The function value depends on the type of link between the two relations. Consider two nodes such as u and v. If (u, v) exists then $s(R(u), R(v))$ is assigned as the edge weight or edge score. Here $R(u)$ and $R(v)$ are the relations to which u and v belong. If (v, u) exists then $I_v(u)s(R(v), R(u))$ is assigned as the edge weight where $I_v(u)$ is the in degree of u due to the tuples of relation $R(v)$. If both the edges (u, v) and (v, u) exists then $min(I_v(u)s(R(v), R(u)), s(R(u), R(v))$ will be the edge weight. Average node weight of the nodes in the steiner tree is given by $Nscore$, $Escore$ is the normalized edge score, λ is a tunable constant. The scoring function used in Banks [9] is given in Equation 7.

$$Escore = \frac{1}{\sum_{e \in T} edgescore(e)}$$

$$\boxed{Score(T) = (1 - \lambda)Escore + (\lambda)Nscore} \tag{7}$$

Bidirectional [21] also uses the same scoring function but it has a different graph traversal algorithm.

1.2.2 Blinks [10]

The Blinks scoring function employs a dual approach wherein both the links and the content are used as parameters to the scoring function of a data graph model system. Similar to Banks [9], it maintains a graph index of the inter connectivity between the tuples. For a given tuple tree T there exists a node r from which a path exists to all the keyword nodes in the tuple tree. They define tuple tree T as $T =< r, (n_1, n_2....., n_k) >$ where $n_1, n_2...n_k$ are keyword nodes in the tree for the given query $(w_1, w_2.....w_k)$.

$$\boxed{Score(T) = f(S_r(r) + \sum_{i=1}^{k} S_n(n_i, w_i) + S_p(r, n_i))} \tag{8}$$

The function f takes in 3 parameters, $S_r(r)$ assigns a score to the root node, $S_n(n_i, w_i)$ measures the similarity between the keywords of the query and the keyword nodes of the tuple tree. This is again calculated by using the base relation statistic. This score is obtained by scanning the contents of the individual nodes in the tuple tree. The function $S_p(r, n_i)$ calculates the minimal path distance between the root node to the keyword node in the graph index.

1.3 Schema Integration of Result Ranking

RDBMS provides a rich set of features which have largely remained unused wrt keyword search domain. Discover [3], Efficient [20] and other CN model systems created an inverted

```
select name, __search_id ,(1+log(1+log(tf)))/(0.2*dl/avdl+0.8)*log(N+1/
   df)as score from(
select name, __search_id ,term_frequency(name,'denzel')as tf ,(select
   avg(char_length(name)) from name) as avdl,length(name) as dl,(
   select frequency from document_frequency where term='denzel' and
   relation = 'name' and attribute = 'name')as df,(select count(*)
   from name)as N  from name where name ilike '%denzel%')as query1
   order by score DESC;
```

Figure 1: Schema Integration of CN result scoring(Efficient)

index as a relation to store the term frequencies. This index was accessed through SQL queries and ultimately the scoring was performed through imperative programming. For the first time, schema integration was performed to score the data graph model system [7]. The function parameters and link information were stored in the relations and top-K steiner trees were provided through a complex SQL query. The entire functionality of top-K result production shifted from imperative space to declarative space. By using imperative scoring many SQL queries have to be executed which leads to frequent shifts from programming space to RDBMS space which can reduce the performance efficiency of the system. If one monolithic SQL query performs the scoring then there would be limited shifts from RDBMS space to programming space and back. This was also proved empirically for the data graph model systems [7] where the declarative approach gave much better performance benefits over the imperative approach.

Schema integration for CN scoring functions(excluding Labrador [1]) is straight forward. The SQL query in Figure 1 performs scoring on a single keyword CN, $\mathsf{Name}^{\mathsf{Denzel}}$. It uses the Efficient [20] scoring function.

The relation documentfrequency(term,relation,attribute,frequency) stores the term frequencies wrt base relation. Building of this relation is a one time cost, unless the base relations are modified. The User Defined Function *termfrequency(string1,string2)* returns the count of *string*2 occurrence in *string*1.

Schema integration of *Labrador scoring function* (eq 6) can be achieved in two phases (refer section 3 for details):

1. In the first phase the result set is scanned to build the inverted index to store the term frequencies wrt result set. The inverted index is stored in a relation.

2. In the second phase result set is again scanned to apply the scoring function. This is again achieved by a complex SQL query which uses the inverted index built in the first phase.

The only implementation technique that is available for *Labrador scoring function* (eq 6) is the imperative version [1]. In our work, a schema integrated approach similar to the scheme described in Figure 1 called SQL-Wrapper is implemented and evaluated.

1.4 Operator Integration of Result Ranking

In the schema integrated approach for *Labrador scoring function*(eq 6), disk-resident inverted index has to be built on the fly during the ranking process for every CN. This disk-resident implementation can pull down the performance which was seen in many real world data sets such as IMDB, Mondial, Wikipedia and DBLP. This problem motivates to build first class SQL operators to perform scoring and ranking the result set of CN.

SQL operators provide the flexibility to build suitable indexes that can boost performance whereas in schema approach only disk resident indexes have to be used.

Two new ranking operators called Root Rank and Join Rank will be introduced in this work:

1. Root Rank operator performs scoring and ranking at the root of the plan tree. Given any plan tree the ranking operator will be introduced at the root.

2. For the Join Rank operator, scoring and ranking will be pushed inside the top join node of the plan tree.

Both these operators work in two phases:

1. In the first phase, disk resident inverted index of schema approach is replaced with memory-disk resident inverted index(refer section 4) to store the term frequencies. This index provides the same advantage as a disk resident index wrt scalability. It also provides an added advantage of performance improvement due to in-memory operations that are involved.

2. The second phase performs the scoring and ranking by applying the scoring function using memory-disk resident index built in the first phase.

These two ranking operators achieve high degree of tight coupling between RDBMS and IR systems.

2 Quality Analysis

A benchmark technique has been proposed [26] which evaluates the keyword search system for user relevant results. In the benchmark technique, datasets, queries and user relevant results are generated first and later a given keyword search system is evaluated for result quality. The systems that were used in the study were Discover [3], Efficient [20], Banks [9], Bidirectional [21], Blinks [10], Effective [28], DPBF [24], CD [25] and Spark [23]. Entire keyword search system involves many components such as CN generator module, scoring functions and CN optimization module. In this scenario different keyword search systems might generate different Candidate Networks and different result set.

The Labrador scoring function(eq 6) needs to be assessed for result quality to justify its usage in keyword search systems. So, we perform empirical study only on the scoring function quality effectiveness by using the benchmark technique [26]. This involves using a single set of Candidate Networks for each keyword query. The different scoring functions are applied on the merged result set of these CN to perform the quality analysis. This study also involves modified Banks [9] scoring function called CNBanks. The CNBanks scoring function uses the Banks [9] scoring function on the CN result set. This experiment is conducted to analyze the effectiveness of data graph model scoring functions on CN systems.

Two metrics are used for result quality analysis. *Reciprocal rank* is the reciprocal of the highest ranked relevant result for a given query. We consider the *mean reciprocal rank* over a set of queries as the mean reciprocal rank (MRR) metric. *Average precision* for a query is the average of the precision values calculated after each relevant result is retrieved for a given query. *Mean average precision*(MAP) metric is given by the mean of average precision values for a set of queries. The experimental procedure used datasets such as IMDB, Mondial and Wikipedia. Each system was subjected to around 50 user queries through which, MAP and reciprocal rank metric values are obtained. Figure 2 and 3 respectively indicate the MAP and MRR analysis of keyword search systems. Labrador scoring function (eq 6) comes with an added advantage of high user relevance as seen in Figure 2 and 3.

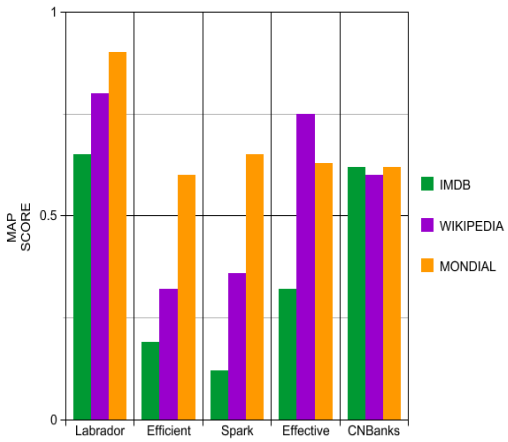

Figure 2: MAP ranking

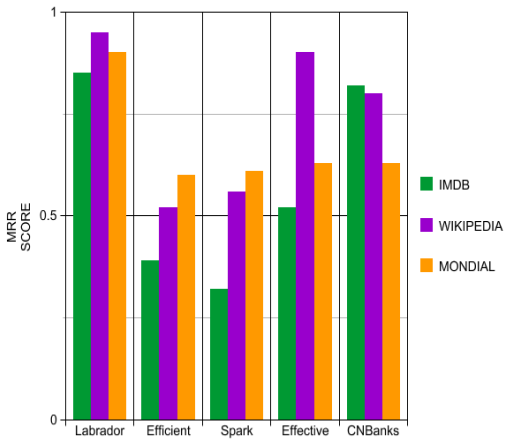

Figure 3: MRR ranking

3 SQL-Wrapper System

SQL-Wrapper performs Candidate Network result ranking of Labrador scoring function (eq 6) by schema integration. In pursuit of this goal the index that is created on the fly is stored as a relation and all the ranking is performed through the aid of such relations.

The relation labterms(rowid integer, term text, attribute text, frequency integer) is used when the required attribute of the result set is split and the terms are stored in this relation, *frequency* attribute has the default value as 1, *rowid* is the identifier for the result row, *term* is the word that is extracted from splitting the tuple and *attribute* is the attribute to which the term belongs. The relation labvalue(term text, attribute text, frequency integer) stores the frequency of each term in the attribute of the result set. In fact this is the main index which is required for ranking. The relation numterms(rowid integer, attribute text, numeratorweight float) is used for storing the numerator part of the cosine value given in Equation(6). The relation denomterms(rowid integer, attribute text, denominatorweight float) has got the same purpose as above but for denominator part of the cosine value. The relation finalscore(rowid integer, score float) stores the final score of a tuple in the result set.

```
select *
from inproceeding
where title ilike '%system%';
```

Figure 4: SQL query generated for a Candidate Network

We will consider a simple case wherein there is a single term keyword query *system*. For this keyword query lets assume that the Candidate Network Inproceeding.title^{system} is generated. The SQL query(base query) in Figure 4 is generated for the Candidate Network.

The first SQL query in Figure 5 splits the string in the specified attribute of the tuple and stores the terms inside a relation called labterms. This action is accomplished through the user defined Pl/PGSQL function *labsplitstringintoterms*(). The second SQL query is calculating the frequency of each term in the result set and inserting the result into the relation labvalue. The third SQL query applies the scoring function on the result set but calculates only the numerator part of the cosine value of the scoring function and inserts the result into numterms. The fourth SQL query calculates the denominator part of the

cosine value of the scoring function on each tuple and inserts into denomterms. The fifth SQL query calculates the final score of the query tuples and inserts into finalscore. The last query presents the base query answers along with their scores in an ordered fashion. Performance of SQL-Wrapper is accelerated by building indexes on all the relations which will be used.

SQL-Wrapper exploits the inbuilt features of RDBMS to perform CN result scoring and ranking. It suffers performance issues due to frequent disk access performed for each term in scoring process(refer to section 7).

4 Root Rank Operator

The Root Rank operator is a first class operator to perform Candidate Network result ranking by using the Labrador scoring function (eq 6). Since, this scoring function has the property of result set dependency, the complete information of the result set is essential. The Root Rank operator is introduced at the root of the plan tree because at this position the complete result set can be materialized.

This operator uses a memory-disk resident index to store the term frequencies which means that the index provides both disk and memory storage facility. Since, Labrador scoring function(eq 6) uses the string equality operation to calculate term frequencies, hashing technique is used to build the memory-disk resident index. Simple Hashing technique is used for building the index for evaluation purpose. In future advanced techniques can be used to build the index depending on the performance requirements. This index is used for both updating and accessing the term frequencies.

The structure of the index is shown in Figure 6. The index in Figure 6 has a two level architecture. The first level are the In-Memory hash buckets. The second level are the hash files which are chained to these hash buckets.

The terms and their frequencies are stored in the corresponding In-Memory hash buckets. If a collision occurs then the term along with its other parameters are updated into the hash file chained to the hash bucket. Since hashing is invoked even for disk storage, the duration of I/Os are also limited. Facility has been added to the Root Rank operator where in the main memory usage can be controlled by assigning usage value to a system variable. This decides the number of In-Memory hash buckets used in the Root Rank operator.

Algorithm 1 Root Rank Operator Execution Algorithm

Let PL be the root of the plan tree returned by the optimizer before Root Rank operator is added.
Let *rnode* be the new empty plan node created for Root Rank operator.
$rnode.cardinality = PL.cardinality$
rnode.cost be assigned by using Equation 19.
$rnode.leftchild = PL$
$rnode.rightchild = NULL$
Invoke Algorithm 2 on *rnode* during the executor phase of the database engine to build the index.
Invoke Algorithm 3 on *rnode* during the executor phase to rank the tuples.

Figure 7 indicates the operator presence in the plan tree. Algorithm 1 explains the details of the algorithm used by the Root Rank operator for execution. There are 2 scans performed on the result set. The first pass is used for updating the frequency of each term.

```
select labsplitstringintoterms(inproceedingid,title,'inproceeding',
'title','1') from inproceeding where title ilike '%system%';
```

Figure 5(a): First SQL Query of SQL-Wrapper

```
insert into labvalue select term, attribute,sum(frequency) from
labterms group by term,attribute;
```

Figure 5(b): Second SQL Query of SQL-Wrapper

```
insert into numterms select id,attr,sum(log(1+(select count(*)
from inproceeding where title ilike system)/freq)) as weight
from (select t.rowid as id,v.attribute as attr,t.term as term,
v.frequency as freq from labterms as t,labvalue as v where
t.term =system and v.term=t.term and v.attribute=t.attribute)as
 a group by id;
```

Figure 5(c): Third SQL Query of SQL-Wrapper

```
insert into denomterms select rowid , attribute,
sqrt(sum(weight*weight)) as denom from (select id as rowid, attr
as attribute,log(1+(select count(*) from inproceeding where title
ilike system)/freq) as weight from (select t.rowid as id,t.attribute
as attr,t.term as term,v.frequency as freq from labvalue as v,
labterms as t where v.term=t.term and v.attribute = t.attribute)
as d)as a group by rowid;
```

Figure 5(d): Fourth SQL Query of SQL-Wrapper

```
insert into finalscore select ids,(select weight from numterms
where id=ids)/d from(select d.rowid as ids, d.weight as d from
numterms as n, denomterms as d where n.id=d.id) as a;
```

Figure 5(e): Fifth SQL Query of SQL-Wrapper

```
select * from Inproceeding as i,finalscore as f
where i.inproceedingid=f.id and i.title ilike '%system%'
order by score DESC;
```

Figure 5(f): Sixth SQL Query of SQL-Wrapper

Figure 5: SQL-Wrapper

Algorithm 2 Index Creation Algorithm

while There are tuples present in the left child of *rnode* **do**
Extract a tuple T from the left child of *rnode*.
Let $(A_1, A_2, ...A_n)$ be the attributes of T where the keyword search is performed for a n keyword query.
for $i = 1 \rightarrow n$ **do**.
split(A_i).
The *split*() function performs the tokenization of terms in the attribute A_i.
Let there be m In-Memory hash buckets allocated for the algorithm run.
for each term $t_{ij} \in A_i$ **do**.
$bucket_{ij} = hash(t_{ij})$
The function $hash()$ maps the term t_{ij} to a memory slot $bucket_{ij}$
if $bucket_{ij}.fill = 0$ **then**
$bucket_{ij}.term = t_{ij}$
$bucket_{ij}.frequency = 1$
$bucket_{ij}.attribute = A_i$
$bucket_{ij}.fill = 1$
else if $bucket_{ij}.fill = 1$ AND $bucket_{ij}.term = t_{ij}$ AND $bucket_{ij}.attribute = A_i$ **then**
$bucket_{ij}.frequency++$
else
$open(file_{ij})$
The *open*() function invokes the hash file attached to the corresponding bucket.
$update(file_{ij}, t_{ij}, A_i)$
The *update*() function updates the term frequency inside the hash file.
end if
end for
end for
store the tuple T inside disk.
end while

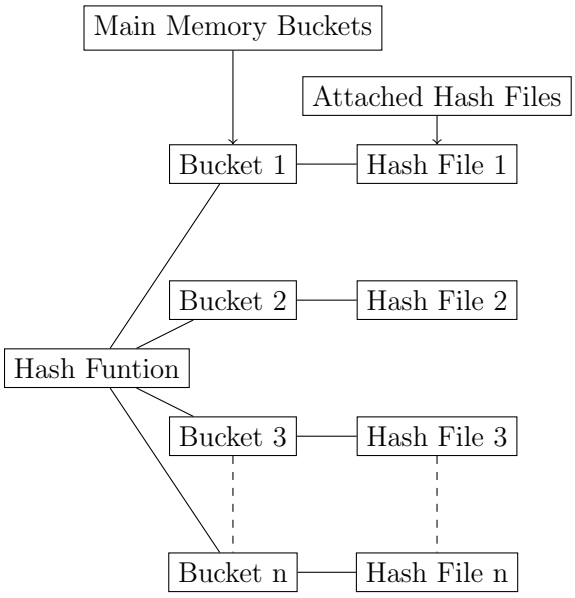

Figure 6: Architecture of memory-disk resident index.

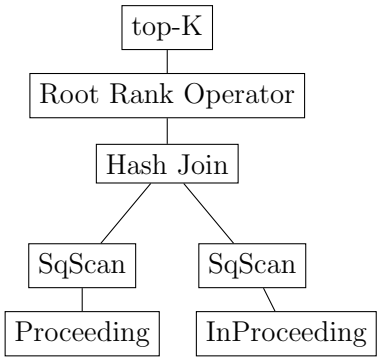

Figure 7: presence of Root Rank operator in plan tree.

Algorithm 3 Tuple Ranking Algorithm

while Result tuples are present in the disk **do**
  Extract a tuple T from the disk.
  for $i = 1 \rightarrow n$ **do**.
    split(A_i).
    for each term $t_{ij} \in A_i$ **do**.
      $bucket_{ij} = hash(t_{ij})$
      if $bucket_{ij}.term = t_{ij}$ AND $bucket_{ij}.attribute = A_i$ **then** $t_{ij}.frequency = bucket_{ij}.frequency$
      else
        $open(file_{ij})$
        $t_{ij}.frequency = seek(file_{ij}, t_{ij}, A_i)$
        The $seek()$ function extracts the frequency of the term t_{ij} from the hash file.
      end if
    end for
  end for
  apply the scoring function listed in (eq 6).
end while
sort the tuples and provide the top-K answers.

Each tuple is scanned in the required attributes and terms are obtained by splitting the string. Each term is directed towards their hash bucket and if the term is already present inside the bucket, the frequency is updated or in the case where collision occurs then it is updated into the hash file. In the next pass, the scoring function (eq 6) is applied which requires term frequencies. The splitting operation is performed on the same attributes and the term frequency is obtained by the hash bucket or by scanning the hash file. After obtaining the individual scores of the tuples, top-K results are provided by sorting the scores.

4.1 I/O Cost Modeling

In this section mathematical model for the I/O performed during the execution of Root Rank operator is established. This I/O model is used for building the optimizer cost model for Root Rank operator.

Consider the index building stage. Here, for a given hash file i the total number of terms that are mapped to it is indicated by $total_i$ and $distinct_i$ is the number of distinct terms present among the terms that are mapped to the hash file i. Let tav_i indicate the average size of terms mapped to a hash file i. In both the update and access case for the index, page miss scenario is assumed. This scenario implies that when a term has to be updated or accessed in a hash file then, the required page has to be loaded from the disk. The page size of the hash files is given by p_{size}. Consider a scenario for the index building phase where the distinct terms from the terms that are mapped to the hash file i are first mapped and the remaining terms in any order are later mapped to the hash file i. This scenario actually gives the largest I/O cost incurred for a hash file which is indicated below.

$(total_i - distinct_i) * \lceil tav_i * distinct_i/p_{size} \rceil + distinct_i$

If there are k hash files allocated for the run of operator. By using the approximation, $\lceil tav_i * distinct_i/p_{size} \rceil \approx tav_i * distinct_i/p_{size}$

$$X_u = \sum_{i=1}^{k} (total_i - distinct_i) * distinct_i * tav_i/p_{size} + distinct_i \tag{9}$$

The number of I/O performed during the index building phase(X_u) is given by equation 9.

$$X_a = \sum_{i=1}^{k} total_i * distinct_i * tav_i/p_{size} \tag{10}$$

In the scoring phase where term frequency is accessed from the hash files, the number of I/O performed during this stage (X_a) is given by equation 10.

$$\boxed{X_{root} = X_u + X_a + 2 * card(r_s)} \tag{11}$$

The total I/O cost(X_{root}) of the Root Rank operator is given by combining the cost of two stages which is given by equation 11. Here $card(r_s)$ indicates the cardinality of result set. The result set is written and read once during the tuple ranking process which is added to the final I/O cost.

4.2 Optimizer Cost Modeling

The optimizer cost model for the Root Rank operator is derived from the I/O cost model developed in the previous section. Fixed number of tuples are obtained as a sample from

the base relations which contain attributes on which keyword search is performed. Let A_c be the attribute for which optimizer cost model statistics are built. Let $sampleset(A_c)$ be the set of tuples obtained as a sample from the base relation which contains attribute A_c. The equations below are derived to build the cost model statistics for attribute A_c. The tuples that are used for this process are obtained from $sampleset(A_c)$.

Let k be the number of memory buckets allotted for the Root Rank operator execution, X_i is the random variable to denote the number of terms that will be mapped to the hash file of memory bucket B_i during index creation phase from $sampleset(A_c)$, $|B_i|$ be the number of distinct terms that can be mapped to the bucket B_i from $sampleset(A_c)$, t_{ij} denotes the distinct term j that is mapped to B_i, $f(t_{ij})$ indicates the frequency of the term t_{ij} in $sampleset(A_c)$.

Let X be the random variable to denote the number of terms that will be mapped to all the hash files. By using expectation conditioning.

$$E[X_i] = \sum_{j=1}^{j=|B_i|} E[X_i|B_i = t_{ij}]P(B_i = t_{ij}) \tag{12}$$

The expression $B_i = t_{ij}$ denotes the condition wherein term t_{ij} is stored in bucket B_i.

$$E[X_i] = \sum_{j=1}^{j=|B_i|} ((\frac{f(t_{ij})}{\sum_r f(t_{ir})})(\sum_{r=1}^{r=|B_i|} (f(t_{ir}) - 1) : r \neq j)) + |B_i| - 1 \tag{13}$$

Since,

$$E[X] = \sum_{i=1}^{i=k} E[X_i] \tag{14}$$

$$E[X] = \sum_{i=1}^{i=k} \sum_{j=1}^{j=|B_i|} ((\frac{f(t_{ij})}{\sum_r f(t_{ir})})(\sum_{r=1}^{r=|B_i|} (f(t_{ir}) - 1) : r \neq j)) + |B_i| - 1 \tag{15}$$

Let I_u be the random variable which denotes the I/O's performed during the index creation phase. The expected value is obtained by using the I/O cost model in eq 9

$$E[I_u] = \sum_{i=1}^{i=k} \sum_{j=1}^{j=|B_i|} ((\frac{f(t_{ij})}{\sum_r f(t_{ir})})(\sum_{r=1}^{r=|B_i|} (f(t_{ir}) - 1) : r \neq j)(\frac{tav_i * (|B_i| - 1)}{p_{size}})) + |B_i| - 1 \tag{16}$$

I_a is the random variable which denotes I/O's performed during index access phase. The expected value can be derived in a similar fashion by using the I/O cost model in eq 10.

$$E[I_a] = \sum_{i=1}^{i=k} \sum_{j=1}^{j=|B_i|} ((\frac{f(t_{ij})}{\sum_r f(t_{ir})})(\sum_{r=1}^{r=|B_i|} f(t_{ir}) : r \neq j)(\frac{tav_i * (|B_i| - 1)}{p_{size}})) \tag{17}$$

$$rootcost(A_c) = \frac{(E[I_u] + E[I_a]) * card(A_c)}{|sampleset(A_c)|} \tag{18}$$

Here, $card(A_c)$ is the cardinality of base relation which contains attribute A_c, $|sampleset(A_c)|$ is the cardinality of $sampleset(A_c)$.

The $rootcost(A_c)$ is the cost model statistic which is built for the Root Rank operator WRT attribute A_c. It is stored in the meta data table to be used during the cost optimization phase.

For a n term keyword query Q which is mapped to n attributes $A_1, A_2.....A_n$.

$$rootplancost = n * \sum_{i=1}^{i=n} \frac{rootcost(A_i) * |RS|}{card(A_i)} + 2 * |RS| \tag{19}$$

$|RS|$ is the estimated output cardinality of the node just below the Root Rank operator in the plan tree. Optimizer plan cost for the Root Rank operator is given by *rootplancost* in Equation 19.

5 Join Rank Operator

Root Rank operator can be seen as a first step for tightly coupling keyword search result ranking with RDBMS. But, in certain queries the top most join node in the plan tree might be time expensive compared to the Root Rank operator. In this situation the bulk of the query execution time is consumed in executing this costly join node(refer section 7) rather than ranking the result set. This situation can be optimized for better performance by applying the Join Rank operator. Join Rank helps in cutting down on the cost of expensive join node and also provides the required top-K answers by directly pulling the ranking process inside the top most join node. Join Rank operator is a modified version of Rank Join operator [2] which performed result ranking inside a join node.

Rank Join operator [2] had the following constraints:

1. The scoring function should be monotonic as illustrated in Equation 1.
2. It should also be result set independent.

The main goal of the operator was to prevent the complete search of the join space and provide the top-K answers by using subset of the search space. In order to perform this, the relations on which the Rank Join is performed should be sorted on those attributes which participate in the scoring function. These sorted inputs can be easily obtained through the interesting order mechanism of the optimizers. Finally, the top-K answers are provided by applying Ripple Join [31] procedure over these sorted inputs.

```
select *
from a,b,c
where a.1 = b.1 and b.2 = c.2
order by (0.6*a.1+0.9*b.2)
stop after 10;
```

Figure 8: Top-K SQL query

The top-K SQL query in the Figure 8 has a scoring function which satisfies both the constraints mentioned above. The query plan in Figure 9 shows the plan without the Rank Join operator. The ranking is performed at the root of the tree similar to the Root Rank operator. The alternate query plan in Figure 10 exhibits the presence of Rank Join operator. As mentioned above it receives inputs ordered on those attributes which participate in scoring function.

The Rank Join operator [2] was not designed for result set dependent scoring functions such as Labrador (eq 6). The Join Rank operator proposed in Algorithm 4 is specifically designed for result set dependent scoring functions. The query plan tree in Figure 11 represents the query plan for the CN $Proceeding^{data} \times Inproceeding^{mining} \times Publisher^{springer}$. The Join Rank operator seen in Figure 11 is applied at the top most join node which is

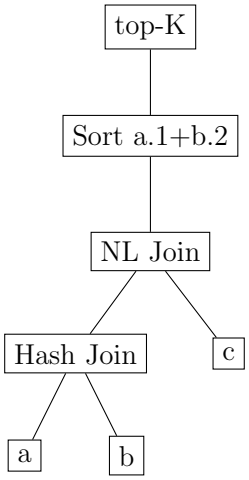

Figure 9: ranking at the top of plan tree.

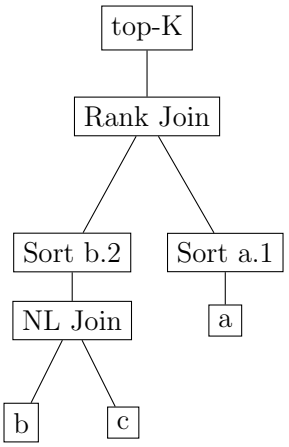

Figure 10: Rank Join operator in plan tree.

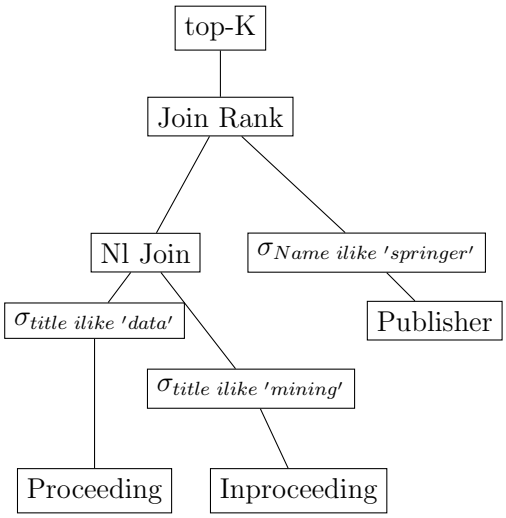

Figure 11: Join Rank operator for keyword search.

a prerequisite. This is due to the requirement of final join column values for its effective implementation.

The Join Rank operator builds two memory-disk resident indexes. The first for storing the join column value frequencies of the left and right input of the Join Rank node. The second is for storing the term frequencies of the keyword search attributes of both left and right input tuples. In the first step index is built for the join column value frequencies. The second step is an optimization step where the tuples that do not participate in the join are dropped. For this procedure the index for the join column value will be used.

For each tuple in the left input of the Join Rank node, the number of tuples it joins with the right input is calculated using the join column value index. If there is no join the tuple is dropped otherwise the terms of the keyword search attributes of the left input tuple are updated inside the second index. This update process uses the join cardinality of the left input tuple as a parameter. So, if a term $t \in$ left input tuple and if its join cardinality is n_t then the frequency of t is considered as n_t. Identical procedure is employed for the right input tuples.

In the last phase each tuple in the left and right inputs are scored and sorted. Finally, Ripple Join [31] algorithm is applied to provide top-K answers.

Algorithm 4 Join Rank node executor algorithm

Build Join Column Index().
Build Term Index for Join Rank().
Perform Join Rank()

Algorithm 5 Build Join Column Index()

while There are tuples in left child of Join Rank node **do**
  Extract a tuple T_l from the left child of Join Rank node.
  let j_l be the join attribute of T_l and t_{jl} be its value.
  I_j is the index built for the join attributes.
  $insert(I_j, j_l, t_{jl}, 1)$
  The $insert()$ function inserts the term t_{jl} into the index I_j.
  Store tuple T_l in the disk.
end while
while There are tuples in right child of Join Rank node **do**
  Extract a tuple T_r from the right child of Join Rank node.
  let j_r be the join attribute of T_r and t_{jr} be its value.
  $insert(I_j, j_r, t_{jr}, 1)$
  Store tuple T_r in the disk.
end while

5.1 I/O Cost Modeling

The I/O cost model for the Join Rank operator is developed by using the I/O cost model of the Root Rank operator. In the first stage I/O cost model for building and accessing the index for the join columns is developed.

$$X_{u_j} = \sum_{i=1}^{k_j} (total_{i_j} - distinct_{i_j}) * distinct_{i_j} * tav_{i_j}/p_{size} + distinct_{i_j} \tag{20}$$

Algorithm 6 Build Term Index for Join Rank()

while There are tuples T_l present in the disk **do**
Extract a tuple T_l from the disk.
Let $(A_1, A_2, ...A_n)$ be the attributes of T_l where the keyword search is performed for a $n+m$ keyword query.
if $(!access(I_j, j_r, t_{jl}))$ **then**
$access()$ extracts the frequency of term t_{ij} from the index I_j WRT attribute j_r.
$joinfrequency = access(I_j, j_r, t_{jl})$
for $i = 1 \rightarrow n$ **do**.
split(A_i).
for each term $t_{ij} \in A_i$ **do**.
$insert(I_t, A_i, t_{ij}, joinfrequency)$
I_t is the index built to store the terms of the keyword search attributes.
end for
end for
end if
Mark and store the tuple T_l inside disk.
end while
while There are tuples T_r present in the disk **do**
Extract a tuple T_r from the disk.
Let $(B_1, B_2, ...B_m)$ be the attributes of T_r where the keyword search is performed for a $n+m$ keyword query.
if $(!access(I_j, j_l, t_{jr}))$ **then**
$joinfrequency = access(I_j, j_l, t_{jr})$
for $i = 1 \rightarrow m$ **do**.
split(B_i).
for each term $t_{ij} \in B_i$ **do**.
$insert(I_t, B_i, t_{ij}, joinfrequency)$
end for
end for
end if
Mark and store the tuple T_r inside disk.
end while

Algorithm 7 Perform Join Rank()

while There are marked tuples T_l present in the disk **do**
  Extract a marked tuple T_l from the disk.
  for $i = 1 \rightarrow n$ **do**.
    split(A_i).
    for each term $t_{ij} \in A_i$ **do**.
      $t_{ij}.frequency = access(I_t, A_i, t_{ij})$
    end for
  end for
  apply the scoring function (eq 6) on T_l and store in disk.
end while
while There are marked tuples T_r present in the disk **do**
  Extract a marked tuple T_r from the disk.
  for $i = 1 \rightarrow m$ **do**.
    split(B_i).
    for each term $t_{ij} \in B_i$ **do**.
      $t_{ij}.frequency = access(I_t, B_i, t_{ij})$
    end for
  end for
  apply the scoring function (eq 6) on T_r and store in disk.
end while
$Ripplejoin(list(T_l), list(T_r))$
  $Ripplejoin()$ procedure performs the Ripple Join [31] to provide top-K tuples. The $list(T_l)$ and $list(T_r)$ contains the sorted tuples which will be used in the *RippleJoin* procedure.

Algorithm 8 $insert(index, attribute, term, frequency)$

$index.bucket_{term} = jhash(term, index)$
The function $jhash()$ maps the $term$ to a slot $bucket_{term}$ for the given $index$.
if $index.bucket_{term}.fill = 0$ **then**
  $index.bucket_{term}.term = term$
  $index.bucket_{term}.frequency = frequency$
  $index.bucket_{term}.attribute = attribute$
  $index.bucket_{term}.fill = 1$
else if $index.bucket_{term}.fill = 1$ AND $index.bucket_{term}.term = term$ AND $index.bucket_{term}.attribute = attribute$ **then**
  $index.bucket_{term}.frequency = index.bucket_{term}.frequency + frequency$
else
  $jopen(file_{term}, index)$
  The $jopen()$ function invokes the hash file attached to the corresponding bucket for the given $index$.
  $jupdate(file_{term}, term, attribute, frequency, index)$
  The $jupdate()$ function updates the term frequency inside the hash file for the given $index$.
end if

Algorithm 9 $access(index, attribute, term)$

$index.bucket_{term} = jhash(term, index)$
if $index.bucket_{term}.term = term$ AND $index.bucket_{term}.attribute = attribute$ **then** return $index.bucket_{term}.frequency$
else
$jopen(file_{term}, index)$
return $jseek(file_{term}, term, attribute, index)$
The $jseek()$ function extracts the frequency of the $term$ from the hash file for the given $index$.
end if

$$total_{i_j} = total_{i_l} + total_{i_r} \tag{21}$$

$$distinct_{i_j} = distinct_{i_l} + distinct_{i_r} \tag{22}$$

$$X_{a_j} = \sum_{i=1}^{k_j} (lookup_{i_l} + lookup_{i_r}) * distinct_{i_j} * tav_{i_j}/p_{size} \tag{23}$$

The number of I/O performed during index building for the join columns is given by X_{u_j}. The terms $total_{i_l}$ and $total_{i_r}$ are the no of terms that are mapped to the hash file i by the left and right inputs, respectively. Similarly, $distinct_{i_l}$ and $distinct_{i_r}$ are the no of distinct terms that are stored in the hash file i by the left and right input. The average term size inside hash file i is given by tav_{i_j}. The no of buckets allotted for building the index for join columns is given by k_j. The no of terms that are mapped to the hash file i from left input to perform the frequency lookup inside right input is given by $lookup_{i_l}$. Similarly, $lookup_{i_r}$ indicates the same concept but for the right input, X_{a_j} is the number of I/O performed during the index access for the join columns.

$$X_{u_t} = \sum_{i=1}^{k_t} (total_{i_t} - distinct_{i_t}) * distinct_{i_t} * tav_{i_t}/p_{size} + distinct_{i_t} \tag{24}$$

$$X_{a_t} = \sum_{i=1}^{k_t} total_{i_t} * distinct_{i_t} * tav_{i_t}/p_{size} \tag{25}$$

$$X_{rankjoin} = X_{u_j} + X_{a_j} + X_{u_t} + X_{a_t} + 2*(card(rs_l) + card(rs_r)) + 2*(\alpha * card(rs_l) + \beta * card(rs_r)) \tag{26}$$

The I/O performed during the index build and index access stage for the keyword search attributes is given by X_{u_t} and X_{a_t}. The total no of terms and distinct terms that are mapped to hash file i is given by $total_{i_t}$ and $distinct_{i_t}$. The no of buckets that are allotted to build the index is indicated by k_t. The average term size inside hash file i is given by tav_{i_t} The cardinality of left and right input is given by $card(rs_l)$ and $card(rs_r)$. The scaling factors α and β are used to obtain the actual tuples in left and right input that participate in join(marked tuples). The total no of I/O performed during Join Rank is given by $X_{rankjoin}$. The left and right input is written and read once. Similarly, the marked tuples are written and read once. These costs are added to the final I/O cost in (eq 26).

5.2 Optimizer Cost Model

The optimizer cost model for the Join Rank operator is derived from the I/O cost model developed in the previous section, A and R are the relations which can be joined through the join columns A_n and R_m respectively. Let $sampleset(A_n)$ and $sampleset(R_m)$ be the set of tuples obtained as a sample from the base relations. The equations below are derived to build the cost model statistics for attribute pair (A_n, R_m). The tuples that are used for this process are obtained from $sampleset(A_n)$ and $sampleset(R_m)$.

Let k_j be the number of memory buckets allotted for the Join Rank operator execution WRT join columns, X_i is the random variable to denote the number of terms that will be mapped to the hash file of memory bucket B_i during index creation phase from $sampleset(A_n) \cup sampleset(R_m)$, $|B_i|$ be the number of distinct terms that can be mapped to the bucket B_i. Similarly, $|B_{i_a}|$ and $|B_{i_r}|$ be the number of distinct terms that can be mapped to the bucket B_i from $sampleset(A_n)$ and $sampleset(R_m)$ respectively, tav_i is the average term size of those terms that can be mapped to B_i. Let, g_{ij} be the distinct term j that is mapped to B_i from $sampleset(A_n) \cup sampleset(R_m)$, $f(g_{ij})$ is the frequency of the term g_{ij}, a_{ij} is the distinct term j that is mapped to B_i from $sampleset(A_n)$, r_{ij} is the distinct term j that is mapped to B_i from $sampleset(R_m)$, X is is the random variable to denote the number of terms that will be mapped to all the hash files.

$$E[X_i] = \sum_{j=1}^{j=|B_i|} E[X_i|B_i = g_{ij}]P(B_i = g_{ij}) \tag{27}$$

Equation 27 is obtained through expectation conditioning. The expression $B_i = g_{ij}$ denotes that term g_{ij} is stored in memory bucket B_i.

$$E[X_i] = \sum_{j=1}^{j=|B_i|} ((\frac{f(g_{ij})}{\sum_r f(g_{ir})})(\sum_{r=1}^{r=|B_i|} (f(g_{ir}) - 1) : r \neq j)) + |B_i| - 1 \tag{28}$$

Since,

$$E[X] = \sum_{i=1}^{i=k_j} E[X_i] \tag{29}$$

$$E[X] = \sum_{i=1}^{i=k_j} \sum_{j=1}^{j=|B_i|} ((\frac{f(g_{ij})}{\sum_r f(g_{ir})})(\sum_{r=1}^{r=|B_i|} (f(g_{ir}) - 1) : r \neq j)) + |B_i| - 1 \tag{30}$$

Let I_u be the random variable which denotes the I/O's performed during the index creation phase. The expected value is obtained by using the I/O cost model in eq 20

$$E[I_u] = \sum_{i=1}^{i=k_j} \sum_{j=1}^{j=|B_i|} ((\frac{f(g_{ij})}{\sum_r f(g_{ir})})(\sum_{r=1}^{r=|B_i|} (f(g_{ir}) - 1) : r \neq j)(\frac{tav_i * (|B_i| - 1)}{p_{size}})) + |B_i| - 1 \tag{31}$$

I_a is the total I/O's performed during index access phase, I_{a_l} and I_{a_r} are the I/O's performed during index access phase by $sampleset(A_n)$ and $sampleset(R_m)$.

$$I_{a_l} = \sum_{i=1}^{i=k} \sum_{j=1}^{j=|B_{i_a}|} lookup(a_{ij})(\frac{tav_i * (|B_i| - 1)}{p_{size}}) \tag{32}$$

$$lookup(a_{ij}) = \begin{cases} 0, & \text{if } B_i \cong a_{ij}. \\ f(a_{ij}), & \text{otherwise.} \end{cases} \tag{33}$$

$$I_{a_r} = \sum_{i=1}^{i=k} \sum_{j=1}^{j=|B_{i_r}|} lookup(r_{ij})(\frac{tav_i * (|B_i| - 1)}{p_{size}}) \tag{34}$$

$$lookup(r_{ij}) = \begin{cases} 0, & \text{if } B_i \cong r_{ij}. \\ f(r_{ij}), & \text{otherwise.} \end{cases} \tag{35}$$

The expression $B_i \cong a_{ij}$ denotes match for the term a_{ij} in memory bucket B_i during index access stage.

$$I_a = I_{a_l} + I_{a_r} \tag{36}$$

$$rankjoincost(A_n, R_m) = \frac{(E[I_u] + I_a) * (card(A_n) + card(R_m))}{|sampleset(A_n)| + |sampleset(R_m)|} \tag{37}$$

$$rankjoinplancost(A, R) = \sum_{i=1}^{i=n_1} \frac{rootcost(A_i) * |RS|}{card(A_i)} + \sum_{i=1}^{i=n_2} \frac{rootcost(R_i) * |RS|}{card(R_i)} + \frac{rankjoincost(A_n, R_m) * (|A_{RS}| + |R_{RS}|)}{card(A_n) + card(R_m)} + 4 * (|A_{RS}| + |R_{RS}|) \tag{38}$$

The statistic $rankjoincost(A_n, R_m)$ is stored in the meta data tables which will be used during query optimization phase. This statistic is built for all join column pairs of relations which will participate in keyword search. Facility has been provided to manually update both $rootcost(A_c)$ for Root Rank and $rankjoincost(A_n, R_m)$ for Join Rank by using $ANALYZE$ command of Postgres.

Consider a n_1+n_2 term keyword query Q which is mapped to attributes $A_1, A_2.....A_{n_1}$ of A and $R_1, R_2.....R_{n_2}$ of R with A_n and B_m as the join attributes. The optimizer estimated cost for the rankjoin node is given by $rankjoinplancost(A, R)$ in Equation 38, $|RS|$ is the estimated result set of the join , $|A_{RS}|$ and $|R_{RS}|$ are the estimated cardinality of left and right input of the Join Rank node.

6 Implementation Details

Both the Root Rank and Join Rank operator for simplicity are introduced without any addition to the SQL language inside PostgreSQL(9.1.2). The presence of an *ilike* operator inside the SQL query signals the invocation of both the operators. The SQL query(Q) in Figure 12 is a typical example of a keyword search query. By using, SET $enable\ Rootoperator = false$ and SET $enable\ JoinRank = false$ both these operators can be disabled which will result in normal execution of the queries without applying keyword search ranking functionality. By using, SET $ScoreMemory$ the memory required to execute both the operators can be assigned. For the Join Rank operator assigned memory is split into $3:1$ ratio. This means that 3/4 of memory is allotted for building keyword search attribute index and the remaining is used for building join value index. The ratio is used because keyword search index

```
select *
from Proceeding as p,Inproceeding as i,Publisher as l
where p.title ilike '%data%' and i.title ilike '%mining%'
and l.name ilike '%springer%'
and p.proceedingid=i.proceedingid
and l.publisherid=i.publisherid
limit 10;
```

Figure 12: Top-K SQL query (Q) for keyword search

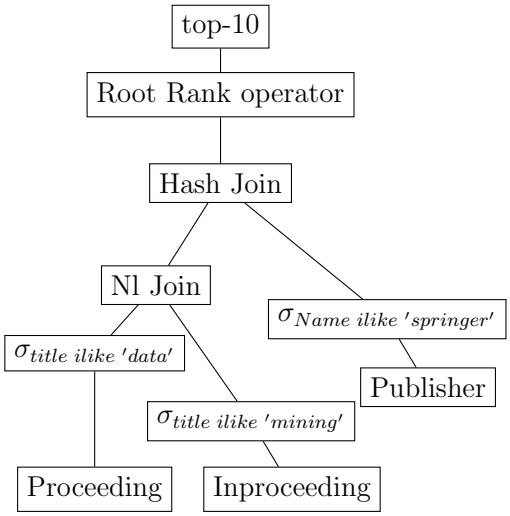

Figure 13: Query plan for (Q) having the Root Rank operator

usually requires bigger storage than join index. In case of Root Rank operator the entire memory is allotted for the keyword search attribute index.

6.1 Parser

The first stage of query execution involves the parser which verifies the syntax and builds the parse tree. The parse tree is stored in the structure *ParseState*. Since, no new SQL language addition is made for the new operators the parser stage is used without any modification.

6.2 Analyzer

The second stage involves the analyzer phase where a parse tree is converted into query tree. The analyzer phase performs the semantic analysis of the SQL query by using the parse tree. In this stage the presence of an *ilike* operator is detected. If successful, then new structures are created to store the attribute, relation and *ilike* term information which will be used in the later stages. The query tree is stored in the structure *Query*.

6.3 Optimizer

The optimization process is performed by using the structure *Query* obtained during the analyzer phase. The Join Rank operator is implemented by using equivalence class strategy

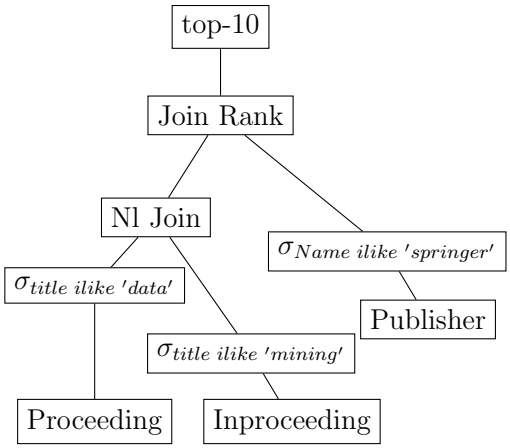

Figure 14: Alternate query plan for (Q) having the Join Rank operator

which most of the current optimizers use. In Figure 14 the DP-Lattice algorithm creates the Join Rank node as an equivalence class which produces two alternate plans(refer to Figure 13). The final optimal plan is obtained by evaluating the complete cost of both the plan trees. The final query plan tree is stored in the structure *PlanState*. An extra field is created in this structure to store the new structures that were created during analyzer phase.

6.4 Executor

The state information of the executor gives the execution road map which is stored in the *PlanState* structure. Executor module uses the algorithms illustrated in Algorithm 1 and Algorithm 4 to execute the Root Rank and Join Rank operator.

The keyword search and join index are implemented as an array of structures called $Hashscoreindex[size1]$ and $Hashjoinindex[size2]$. The former is for the keyword search attributes and the latter for join columns. Each structure element is tailed with a file which acts as a hash file. The API *MakeSingleTupleTableSlot* is used for copying of tuples. The data inside the tuples are stored in *Datum* format which is basically a pointer to the actual data. The actual value has to be extracted by the API *DatumGetCString*. The API *ExecCopySlotMinimalTuple* writes tuples to files by compressing them. Tuple compression provides an opportunity to perform many instances of storage and sorting operations inside memory. Otherwise, in many cases these operations had to be performed through disk. Thus, providing much needed performance benefits. Postgres provides a dynamic memory allocation through *palloc*. The memory allotted through this mechanism need not be freed individually. Once the executor completes its execution all the memory allotted through *palloc* is freed automatically. This mechanism provides a shield against accidental memory leakage.

The steps mentioned below illustrates the Root Rank operator implementation inside executor routine:

1. $Hashscoreindex[size1]$ is initialized by allotting the required memory through *palloc*. Similarly, the corresponding hash files are also initialized.

2. Since Root Rank operator has only left child. Each tuple is extracted from left child through the API *ExecProcNode*. Every tuple that is extracted is deep copied through the API *MakeSingleTupleTableSlot*. If the tuples have to be stored on the disk then the API *ExecCopySlotMinimalTuple* is invoked.

3. The tuples are probed for the relevant attributes through the structure *tupleDescriptor*.

Number	Query	Candidiate Network
1	mining	$InProceeding^{mining}$
2	operating	$InProceeding^{operating}$
3	algorithm	$InProceeding^{algorithm}$
4	network	$InProceeding^{network}$
5	data	$InProceeding^{data}$
6	system	$InProceeding^{system}$
7	data	$Proceeding^{data}$
8	system	$Proceeding^{system}$

Table 1: Single Keyword Queries

The required values are extracted through the API $DatumGetCString$. The extracted values are used to build the required index.

4. Once the tuple scoring phase is completed. All the sorted tuples are indexed by their rank and are returned to the destination port. For the Join Rank operator, two indexes $Hashscoreindex[size1]$ and $Hashjoinindex[size2]$ are initialized. The operator has both left and right child so the tuples are extracted by using the API's $ExecProcNode(outernode)$ and $ExecProcNode(innernode)$. All the other executor functionalities remain similar to Root Rank operator.

7 Experiments

All the three systems SQL-Wrapper, Root Rank operator and Join Rank operator were subjected to performance experiments. The DBLP database which has become a de facto benchmark in keyword search system evaluation [9, 10, 21, 23] has been used in these experiments. The dataset has around 600 MB of data, 9 relations, 25 text attributes and 7000k of tuples. Since benchmark keyword queries do not exist for this dataset, we have used queries from three sources,

1. Queries which were used in expert finding technique [30]. This procedure involved finding the experts in different fields by using the DBLP dataset. The field names on which expert finding algorithm was executed were used as keyword query in this experiment.

2. Queries which were used in other keyword search systems [10].

3. Queries that were self constructed.

The SQL queries were divided according to number of joins(m) that were involved in them. Table 1 enumerates the single term keyword queries and their corresponding CN. Figure 15 exhibits the performance of SQL-Wrapper and Root Rank operator for the keyword queries listed in Table 1. The number of results retrieved was kept static(k=100). As expected, Root Rank operator has a considerable performance benefits over SQL-Wrapper. This is largely due to the in-memory index which provides superior performance benefits over purely disk based index used in SQL-Wrapper.

The optimizer cost model developed for both Root Rank and Join Rank operators models the I/O cost. In Figure 16 the correlation between the actual I/O performed during runtime and optimizer cost is plotted for Root Rank operator wrt Table 1 queries. The plan trees utilized the perfect selectivity estimation scenario. In this case the queries were executed once and the correct selectivity at all nodes of the plan tree was calculated and used in cost estimation. In this scenario the deficiency in the cost model can be analyzed because if the cost model performs poorly it is evident that the selectivity estimation is not

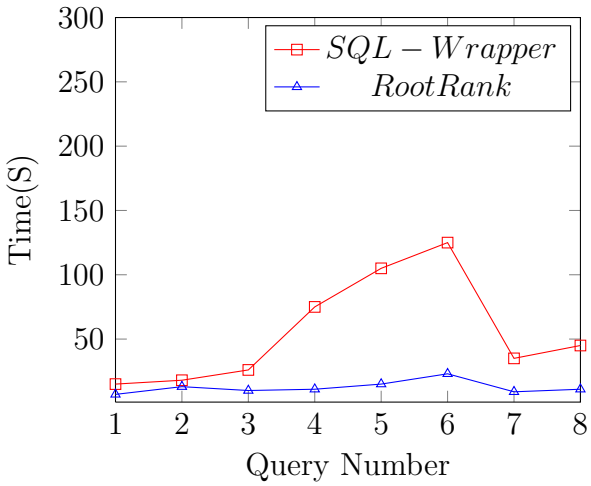

Figure 15: SQL-Wrapper vs Root (m=0, k=100).

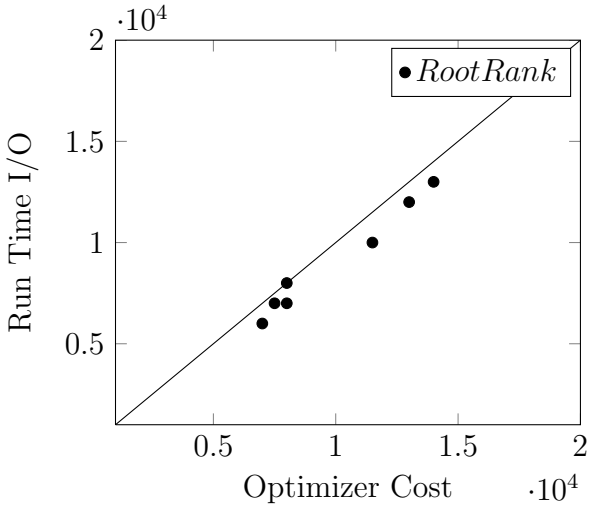

Figure 16: Optimizer cost vs Run Time I/O(Table 1)

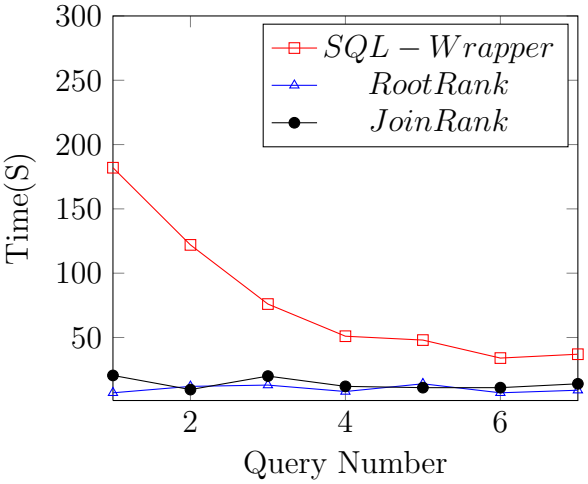

Figure 17: SQL-Wrapper vs Root Rank vs Join Rank(m=1, k=100).

Number	Query	Candidiate Network
1	system data	$InProceeding^{system} \times Proceeding^{data}$
2	system data	$InProceeding^{data} \times Proceeding^{system}$
3	computer science	$InProceeding^{computer} \times Proceeding^{science}$
4	computer science	$InProceeding^{science} \times Proceeding^{computer}$
5	software system	$InProceeding^{software} \times Proceeding^{systems}$
6	elsevier performance	$Publisher^{elsevier} \times Proceeding^{performance}$
7	springer algorithm	$Publisher^{springer} \times Proceeding^{algorithm}$

Table 2: Two Term Keyword Queries

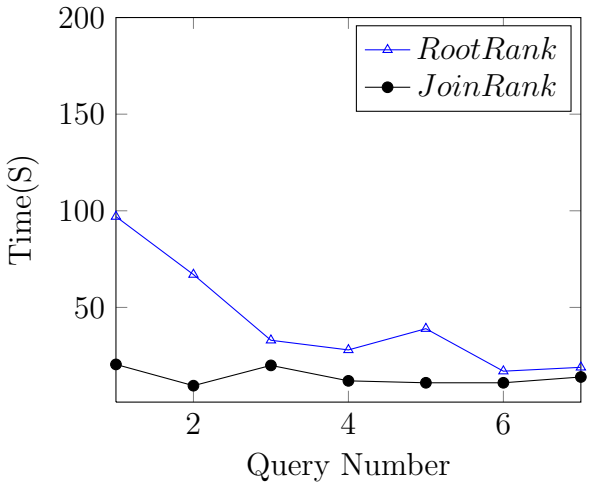

Figure 18: Root Rank vs Join Rank (skew version).

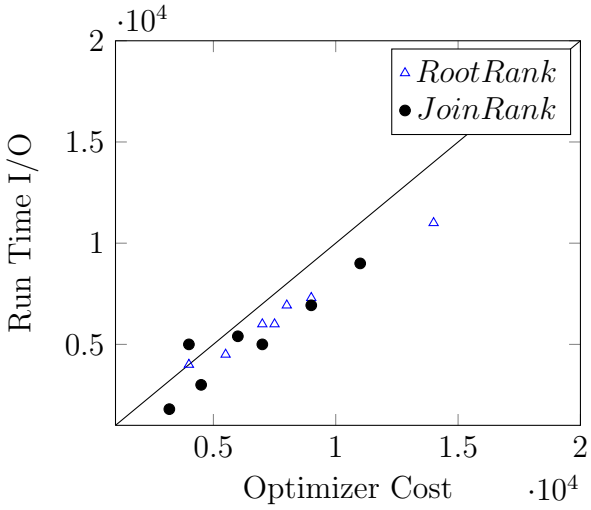

Figure 19: Optimizer cost vs Run Time I/O(Table 2)

Number	Query	Candidiate Network
1	computer science	$Series^{computer} \times Proceeding \times InProceeding^{Science}$
2	computer science network	$Series^{computer} \times Proceeding^{network} \times InProceeding^{Science}$
3	springer data	$Publisher^{springer} \times Proceeding \times InProceeding^{data}$
4	michael database	$Person^{michael} \times RelationPersonInProceeding \times InProceeding \times Proceeding^{database}$
5	kevin statistical	$Person^{kevin} \times RelationPersonInProceeding \times InProceeding \times Proceeding^{statistical}$

Table 3: Keyword Queries for $m > 1$

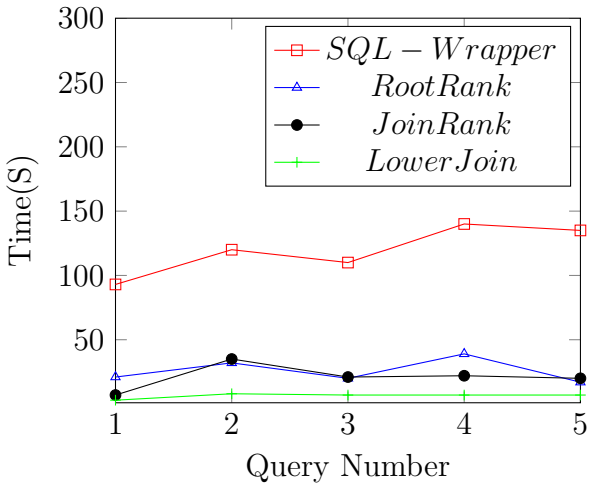

Figure 20: SQL-Wrapper vs Root Rank vs Join Rank ($m > 1, k = 100$).

the problem but the cost model fits poorly to the operator. Perfect straight line fit cannot be observed in Figure 16 because the operator is applied on text attributes and meta data tables only store partial information of the text data.

Table 2 enumerates the two term keyword queries along with their CN. In Figure 17 performance of keyword queries listed in Table 2 are plotted. SQL-Wrapper again suffers from performance bottleneck WRT both Root Rank and Join Rank operator. Both the operators do not exhibit large variations in their performance. The main reason being that Join Rank requires two indexes to be built and this can be a suboptimal choice in certain queries. Root Rank operator is beneficial when the final join node has a minimal cost. Similarly, Join Rank is beneficial choice when final join node has a huge cost due to skewed distribution of join column values which can hamper the normal join algorithms performance (In this scenario hash join becomes disk oriented rather than finishing the task inside memory which can degrade performance).

Since, the normal DBLP has a negligible skew. Artificial skew is introduced in the join columns for Table 2 queries in order to exhibit the superior performance of Join Rank. Figure 18 exhibits performance benefits of such a situation where the degree of skewness was 20 percent. Consider a situation for a query where both the join columns had a single common value and a high join input cardinality. In this situation Hash Join algorithm exhausts memory. But, Join Rank only updates the join column frequency inside the memory-disk resident index. Similarly, the term index updates only the corresponding term frequency. Thus, Join Rank saves itself from burning out of memory. In fact, when the degree of skewness reached around 30 percent many queries exhausted their memory using the Root Rank operator.

Figure 19 plots the correlation between runtime I/O against the optimizer cost for Table 2 queries. Again, perfect selectivity estimation scenario was utilized. Both the operators exhibited small variations in their relative predicted cost for the same set of queries. This also substantiates the empirical result which does not exhibit large variations in the relative performance of these operators.

Table 3 enumerates the keyword queries which produce CN with more than one join. Figure 20 plots the relative performance of all the three systems for Table 3 queries. In these SQL queries the total execution cost of plan tree below the Join Rank node is also plotted as *LowerJoin*. As seen in Figure 20 most of the cost has been due to the ranking functionality. If skew is introduced in the join columns this can alter situation where in some instances the joins in the lower level of the plan tree would incur higher cost.

As explained above skewness in a dataset can be eliminated by the help of Join Rank operator. But, this is largely effective on a single join queries. For those queries with multiple joins, ranking functionality can become lighter than the lower level joins. In the current work four practical datasets have been examined namely DBLP, IMDB, Mondial and Wikipedia. In all these datasets, degree of skewness is not significant.

8 Conclusions

In this work three different techniques have been introduced to perform keyword search result ranking on RDBMS. Out of which two new operators have been introduced to achieve the long desired goal of tightly coupling IR systems and RDBMS. The new operators have justified their introduction by providing excellent performance benefits over SQL-Wrapper technique. The Labrador scoring function which has been used in this task has justified its usage by providing high user relevant answers.

There are two major future extensions for this work:

1. The Join Rank operator has been introduced only at the top most join node. The Join Rank operator [2] can be introduced at join nodes which are located at all levels of plan tree. The task of introducing Join Rank operator at all levels is a hard problem. At lower levels of plan tree it is difficult to know the complete information about the final join column values which is essential to apply the Join Rank algorithm.

2. There is a need of new operators not just in the traditional RDBMS but also in recent systems such as column database and probabilistic database systems. These database systems currently do not even have result scoring functions. By developing scoring functions and operators, complete integration of IR and all kinds of Database systems can be achieved.

References

[1] Da Silva, Altigran S. and Mesquita, Filipe and de Moura, Edleno S. and Calado, Pável and Laender, Alberto H. F. *LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces*. Inf. Process. Manage., July, 2007, 983–1004.

[2] Aref, Walid G. and Ilyas, Ihab F. and Elmagarmid, Ahmed K., *Supporting top-K join queries in relational databases*, Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '03, Berlin, Germany, 754–765.

[3] Hristidis, Vagelis and Papakonstantinou, Yannis, *Discover: keyword search in relational databases*, Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, Hong Kong, China, 670–681.

[4] Agrawal, Sanjay and Chaudhuri, Surajit and Das, Gautam, *DBXplorer: enabling keyword search over relational databases*, Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin, 627–627.

[5] Finger, Jonathan and Polyzotis, Neoklis, *Robust and efficient algorithms for rank join evaluation*, Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, Providence, Rhode Island, USA, 415–428.

[6] Schnaitter, Karl and Polyzotis, Neoklis, *Evaluating rank joins with optimal cost*, Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '08, Vancouver, Canada, 43–52.

[7] Li, Guoliang and Feng, Jianhua and Zhou, Xiaofang and Wang, Jianyong, *Providing built-in keyword search capabilities in RDBMS*, The VLDB Journal, February 2011, 1–19.

[8] Schnaitter, Karl and Polyzotis, Neoklis, *Optimal algorithms for evaluating rank joins in database systems*, ACM Trans. Database Syst., February 2010, 6:1–6:47.

[9] Aditya, B. ,Bhalotia, Gaurav , Chakrabarti, Soumen , Hulgeri, Arvind , Nakhe, Charuta , Parag , Sudarshan, S., *BANKS: browsing and keyword searching in relational databases*, Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, Hong Kong, China, 1083–1086.

[10] He, Hao and Wang, Haixun and Yang, Jun and Yu, Philip S., *BLINKS: ranked keyword searches on graphs*, Proceedings of the 2007 ACM SIGMOD international conference on Management of data, Beijing, China, 305–316.

[11] Carey, Michael J. and Kossmann, Donald, *On saying Enough already in SQL*, Proceedings of the 1997 ACM SIGMOD international conference on Management of data, Tucson, Arizona, United States, 219–230.

[12] *h*ttp://www.cse.yorku.ca/ oz/hash.html

[13] Martinenghi, Davide and Tagliasacchi, Marco, *Proximity rank join*, Proc. VLDB Endow., September 2010, 352–363.

[14] Chang, Lijun and Yu, Jeffrey Xu and Qin, Lu and Lin, Xuemin, *Probabilistic ranking over relations*, Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, 2010, Lausanne, Switzerland, 477–488.

[15] Schneider, Carl and Polyzotis, Neoklis, *Best Newcomer Award: Evaluating rank joins with optimal costs*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada.

[16] Ilyas, Ihab F. and Shah, Rahul and Aref, Walid G. and Vitter, Jeffrey Scott and Elmagarmid, Ahmed K., *Rank-aware query optimization*, Proceedings of the 2004 ACM SIGMOD international conference on Management of data, Paris, France, 203–214.

[17] Ladwig, Günter and Tran, Thanh, *Index structures and top-k join algorithms for native keyword search databases*, Proceedings of the 20th ACM international conference on Information and knowledge management, CIKM '11, Glasgow, Scotland, UK, 1505–1514.

[18] Ilyas, Ihab F. and Aref, Walid G. and Elmagarmid, Ahmed K., *Joining ranked inputs in practice*, Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, Hong Kong, China, 950–961.

[19] *http://www.cs.ust.hk/ dlee/Papers/ir/ieee-sw-rank.pdf.*

[20] Hristidis, Vagelis and Gravano, Luis and Papakonstantinou, Yannis, *Efficient IR-style keyword search over relational databases*, Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '03, Berlin, Germany, 850–861.

[21] Kacholia, Varun and Pandit, Shashank and Chakrabarti, Soumen and Sudarshan, S. and Desai, Rushi and Karambelkar, Hrishikesh, *Bidirectional expansion for keyword search on graph databases*, Proceedings of the 31st international conference on Very large data bases, VLDB '05, Trondheim, Norway, 505–516.

[22] Li, Guoliang and Feng, Jianhua and Wang, Jianyong and Zhou, Lizhu, *Effective keyword search for valuable lcas over xml documents*, Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, CIKM '07, Lisbon, Portugal, 31–40.

[23] Luo, Yi and Lin, Xuemin and Wang, Wei and Zhou, Xiaofang, *Spark: top-k keyword query in relational databases*, Proceedings of the 2007 ACM SIGMOD international conference on Management of data, Beijing, China, 115–126.

[24] Bolin Ding and Yu, J.X. and Shan Wang and Qin, Lu and Xiao Zhang and Xuemin Lin, IEEE 23rd International Conference on Data Engineering, *Finding Top-k Min-Cost Connected Trees in Databases*, 2007, 836-845.

[25] Coffman, Joel and Weaver, Alfred C., *Structured data retrieval using cover density ranking*, Proceedings of the 2nd International Workshop on Keyword Search on Structured Data, KEYS '10, Indianapolis, Indiana, 1:1–1:6.

[26] Coffman, Joel and Weaver, Alfred C., *A framework for evaluating database keyword search strategies*, Proceedings of the 19th ACM international conference on Information and knowledge management, CIKM '10, Toronto, ON, Canada, 729–738.

[27] Ross, Sheldon M., *Introduction to Probability Models, Ninth Edition*, 2006, Academic Press, Inc., Orlando, FL, USA.

[28] Liu, Fang and Yu, Clement and Meng, Weiyi and Chowdhury, Abdur, *Effective Keyword Search in Relational Databases*, Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, 563–574.

[29] Martinenghi, Davide and Tagliasacchi, Marco, *Proximity measures for rank join*, ACM Trans. Database Syst., mar, 2012, 2:1–2:46.

[30] Moreira, Catarina and Calado, Pvel and Martins, Bruno, *Learning to Rank for Expert Search in Digital Libraries of Academic Publications*, Springer Berlin Heidelberg, Progress in Artificial Intelligence, 7026, Lecture Notes in Computer Science, 2011, 431-445.

[31] Haas, Peter J. and Hellerstein, Joseph M., *Ripple Joins for Online Aggregation*, Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA, 287–298.

[32] Pagh, Anna and Pagh, Rasmus, *Uniform Hashing in Constant Time and Optimal Space*, SIAM J. Comput., March 2008, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[33] Ramakrishnan, Raghu and Gehrke, Johannes, *Database Management Systems*, 2003, McGraw-Hill, Inc., New York, NY, USA.