Root Rank: A Relational Operator for KWS Result Ranking

Vinay M S
Database Systems Lab, Indian Institute of Science
Bangalore, India
vinayms84@gmail.com

Jayant R. Haritsa
Database Systems Lab, Indian Institute of Science
Bangalore, India
haritsa@iisc.ac.in

ABSTRACT

A popular approach to hosting Keyword Search Systems (KWS) on relational DBMS platforms is to employ the Candidate Network framework. The quality of a Candidate Network-based search is critically dependent on the scoring function used to rank the relevant answers. In this paper, we first demonstrate, through a detailed empirical study, that the Labrador scoring function provides the best user relevance among contemporary Candidate Network scoring functions.

Efficiently incorporating the Labrador function, however, is rendered difficult due to its *Result Set Dependent* (RSD) characteristic, wherein the distribution of keywords in the *query results* influences the ranking. In this paper, we investigate addressing the RSD challenge through inclusion of *custom operators* within the database engine. Specifically, we propose and evaluate an operator called Root Rank, which performs result ranking in the root of the query execution plan.

The Root Rank operator has been implemented on a PostgreSQL codebase, and its performance profiled over real-world data sets, including DBLP and Wikipedia. Our experimental observations indicate that the Root Rank operator is highly successful in delivering processing times that are comparable to, or better than, those of non-RSD implementations. We expect these results to aid in the organic hosting of KWS functionality on database systems.

ACM Reference Format:
Vinay M S and Jayant R. Haritsa. 2019. Root Rank: A Relational Operator for KWS Result Ranking. In *6th ACM IKDD CoDS and 24th COMAD (CoDS-COMAD '19), January 3–5, 2019, Kolkata, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297001.3297014

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[1].

```
SELECT *
FROM Proceeding AS p, Inproceeding AS i
WHERE p.title ilike '%data%' AND
 i.title ilike '%mining%'
AND p.proceedingid=i.proceedingid
```

Figure 1: CNEQ for $CN1$

Many KWS [2–10] are built by using the popular framework of Candidate Networks (CNs). A CN refers to a joined network of relations, whose result set provides an idiosyncratic set of answers to the user's keyword query. Each row of the CN result set is formed by connecting various tuples of different relations, and hence constitutes a *tuple tree.*

To make the concept of CN more concrete, consider the keyword query *Data Mining* applied on the DBLP dataset [5], resulting in the following CN, which is denoted as $CN1$:

$$\mathsf{Proceeding.title}^{\mathsf{Data}} \times \mathsf{InProceeding.title}^{\mathsf{Mining}}$$

Here, *Data* keyword is mapped to the attribute **Proceeding.title(p.title)** and *Mining* keyword is mapped to the attribute **InProceeding.title(i.title)**.

The *straightforward implementation technique* to provide top-K tuple trees for the user keyword query involves: (1) Executing each qualifying CN through SQL query denoted as *CN Execution Query (CNEQ)*, which is represented in Figure 1. (2) Each tuple tree of the CN result set is scored through specified *scoring function.* (3) Merging and ordering the tuple trees of all CNs based on their individual scores.

1.1 Result Relevance

In the above process, key aspects of the scoring function are the (a) quality with regard to result relevance, and (b) efficiency with regard to implementation. With regard to both these aspects, we address a scoring function presented in the literature, which is denoted as Labrador Scoring Function (LSF) [2]. We show in this paper that, the LSF is empirically superior in its result quality as compared to the several scoring functions proposed earlier in the literature (e.g. [4–6, 8]). Specifically, we demonstrate this desirable behavior on the popular Coffman-Weaver benchmark [11].

The enhanced quality of the LSF can be attributed to its unique *result set dependent* (RSD) feature, wherein the distribution of keywords in the *query results* is a primary factor in the ranking. To make this RSD feature concrete, consider $CN1$:

The LSF for ranking the result set tuple trees of $CN1$, is the Cosine Similarity-based formulation given in Equation 1

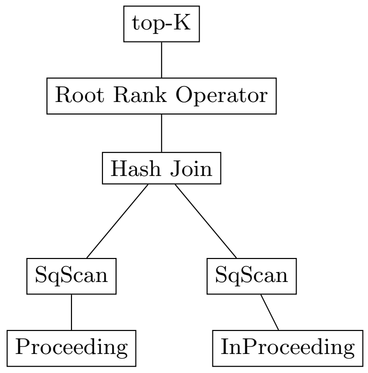

Figure 2: Root Rank in CNEQ Plan Tree ($CN1$)

– a complete description of LSF is given in Appendix A. Here, $score(T)$ assigns LSF score to the tuple tree T, $\overrightarrow{p.title}$ and $\overrightarrow{i.title}$ denote the vectors which contain only those terms that $\in$ keyword query, and n is a suitable constant.

$$\boxed{score(T) = n * \frac{\cos(p.title, \overrightarrow{p.title}) + \cos(i.title, \overrightarrow{i.title})}{2}} \tag{1}$$

The Cosine Similarity function requires calculation of weights for all the terms present in the value of the attribute mapped to answer the keyword query. The weight calculation is shown in Equation 2 – here, w_r is the weight of the term r which belongs to the given attribute value, f_r is its frequency in the given attribute of the *result set*, and $|result\_set|$ is the cardinality of the CN result set.

$$w_r = log(1 + \frac{|result\_set|}{f_r}) \tag{2}$$

1.2 Computational Efficiency Challenge

The enhanced quality of LSF, however, comes with a catch – the challenge of providing an efficient implementation. LSF requires the creation of an *inverted index* on-the-fly for storing and accessing term frequencies during the result ranking process of each CN, whereas the implementation techniques for the non-RSD scoring functions [4, 5] do not have this constraint. The original Labrador implementation assumed a wholly in-memory inverted index platform; however, this fails to scale to the Big Data environments that are becoming commonplace.

To address this RSD efficiency challenge, we have designed, implemented and evaluated an approach; wherein, the database engine is augmented with a *custom operator* that performs result ranking in the query execution plan of CNEQ. Specifically, a new first-class query operator called **Root Rank** is introduced to implement the LSF. This operator performs ranking of the CN result set at the *root* of the CNEQ plan tree, because at this position, complete result can be generated for implementing LSF. Figure 2, illustrates the CNEQ plan tree which utilizes the Root Rank operator for implementing LSF. Here, the CNEQ plan tree for $CN1$ is considered for illustration. The Root Rank operator builds a simple hybrid memory-cum-disk resident hash index, wherein each memory bucket of the hash index is tailed with a hash file on disk. The terms and their frequencies are stored in these hash buckets or hash files. The Root Rank operator is expected to provided better performance than contemporary techniques [4, 5], due to speeding up access to the inverted index by utilizing the available memory.

The Root Rank operator has been implemented on the PostgreSQL platform, and a detailed performance evaluation has been carried out on the DBLP and Wikipedia datasets. The results are compared using the implementation techniques recommended for the standard non-RSD scoring functions [4, 5] as the efficiency yardstick. Our results indicate that the Root Rank operator is highly efficient, delivering processing times that are comparable to, or better than, those of non-RSD implementations. We expect these results to aid in the efficient hosting of KWS functionality on database systems.

1.3 Organization

The remainder of the paper is organized as follows: Section 2 describes the related work in the KWS domain. The analysis of scoring function quality is carried out in Section 3. The Root Rank operator is described in Section 4, followed by a detailed empirical evaluation in Section 5, using conventional scoring functions as the yardstick. Finally, we summarize our conclusions in Section 6, and outline future research avenues.

2 RELATED WORK

The first Information Retrieval (IR) oriented scoring function designed for CN result ranking was Efficient [4]. Result pipeline-based implementation techniques, called Single Pipeline and Global Pipe-line, which facilitated the evaluation of the Efficient scoring function were proposed for single CN and multiple CNs, respectively.

An improved extension of Efficient, called Spark, was proposed in [5], and Block Pipeline technique was presented for its implementation, which achieves the dual goal of providing top-K result set for both individual and group CNs.

All the above three techniques: Single Pipeline, Global Pipeline and Block Pipeline, are implemented through imperative programing, and utilize a disk-based inverted index to store and access the scoring function parameter values. Since the RSD property is not found in the scoring functions of either Efficient or Spark, they do not require on-the-fly index creation on the result set.

The Effective scoring function, which is also non-RSD, was proposed in [6], and focuses solely on the result quality – it exhibited better performance than Efficient with regard to providing user relevant results. The Effective scoring function was implemented through straightforward implementation technique.

The number of qualifying CNs for a keyword query can be extremely large, especially when the query contains more number of query terms or the underlying schema of the dataset is complex. In such scenarios, two different schemes are used in the literature to limit the number of generated CNs. In the first scheme [7, 9], CNs themselves are ranked based on their potential to produce relevant answers, and top-ranked CNs are selected for subsequent result production. Whereas, in the second scheme [10], the number of joins involved in the qualifying CNs is controlled through a threshold. However, in this work, the issues related to CN result ranking are addressed, and therefore the mechanisms presented in [7], [9] and [10] are not evaluated in this work.

In [8], an improvement over Block Pipeline, called the LP technique, was presented, whose goal was to increase the execution efficiency of result production. The major performance issue in the: Single, Global and Block pipeline techniques was that, given a set of tuples that can potentially form a tuple tree, their join condition is checked through launching parametrized queries, which can become large in some queries and result in performance bottlenecks [8]. Hence, the LP technique utilized memory caching to check for join conditions. However, if the sizes of the CN result sets are large, it might not resolve the above outlined performance issue [8].

Apart from CNs, another popular framework to implement KWS is the Data Graph model [12–18]. In Data Graph systems, the tuple connectivity information is stored as a graph in the memory. To answer a keyword query, graph traversal algorithms are designed to obtain the relevant results, and these results are sorted by using data graph scoring functions. However, a major drawback of Data Graph framework is that the graph has to be materialized and stored in the memory, which can face serious scalability issues against large datasets [8].

3 SCORING FUNCTION ANALYSIS

In this section, we focus on the scoring function quality, and demonstrate that the LSF delivers the best overall performance.

A unique component of LSF, intended to improve its user relevance, is its RSD property. To understand the significance of RSD property, it is important to analyze the structure of IR scoring functions. Considering the generic IR systems, the IR scoring functions are used for ranking the retrieved results from the document repository WRT user query Q. Let D indicate this document depository, and $R = (d_1, d_2, ...d_n)$ denote the set of retrieved documents that form the result set for Q. The IR scoring functions assign weights to query terms and, in some cases, other terms located in each result set document $d_i (1 \leq i \leq n)$. Let, $w(t_j)$ be the term weight of $t_j \in d_i$. The weight assigned to the term is decided on a special metric called *specificity*, and this metric is represented in Equation 3 – here, $spec(t_j, D)$ denotes the specificity of t_j in D, and $idf(D, t_j)$ is the number of documents in D in-which t_j occurs. Larger term weights in a document, will improve the score of the document.

$$w(t_j) \propto spec(t_j, D) \propto \frac{1}{idf(D, t_j)} \tag{3}$$

Consider a scenario, suppose that t_j has high specificity in D. Since $R \subseteq D$, it can be inferred that t_j will have similar specificity, when specificity is calculated by considering only R. Similar reasoning can be used to analyze the case where t_j has low specificity. Thus, the specificity of a term in D is generally similar in R.

The KWS on RDBMS have a different interpretation of D and R compared to the generic IR systems. This different interpretation is that, each tuple tree of the CN result set is considered as a result set document which $\in R$. Similarly, each tuple in the base relation is considered as a separate document which $\in D$. Thus, all the tuples in base relations collectively form D. However, it must be noted that tuple trees are created by joining tuples from different relations involved in the CN. So, each tuple of the base relation might join multiple times with other tuples of different base relations in forming the result set. Thus, clearly, $R \nsubseteq D$.

The non-RSD scoring functions [4–6, 8] calculate the specificity of each required term $\in R$, by using the base relation statistics – meaning statistics corresponding to D. Since, $R \nsubseteq D$, it might create scenarios where a term t_j having a particular specificity (low or high) in the base relation might demonstrate the opposite specificity (high or low) in the result set R. Hence, the non-RSD scoring functions fail to capture such scenarios. However, the LSF due to its RSD property can easily account for such scenarios, and deliver results with high user relevance.

A benchmark framework for evaluating the result effectiveness WRT user relevance for KWS was proposed in [11]. Here, the datasets, queries and user relevant results are initially generated, and a given KWS is subsequently evaluated for its result quality. The systems that were used in the study included both CN systems and Data Graph systems, including the CN systems presented in [4–6]. The study used three datasets: IMDB, Mondial and Wikipedia, and each KWS was subjected to around 50 user queries on the individual datasets. Two metrics were used for result quality analysis, *Mean Reciprocal Rank* (MRR) and *Mean Average Precision* (MAP), which are described below.

Consider a keyword query Q. The Reciprocal Rank for Q is represented in Equation 4. Here, $rank_Q$ is the rank of the first relevant tuple tree of Q in the retrieved result set, and $reciprocal(Q)$ is the reciprocal rank for Q. The MRR for a set of queries, denoted by $Q\_set = (Q_1, Q_2, ...Q_k)$, is represented in Equation 5 – here, $MRR(Q\_set)$ is the MRR value for Q_set.

$$reciprocal(Q) = \frac{1}{rank_Q} \tag{4}$$

$$MRR(Q\_set) = \frac{\sum_{i=1}^{k} reciprocal(Q_i)}{k} \tag{5}$$

The Precision metric for query Q, denoted by $prec(Q)$, is represented in Equation 6. Here, R is the result set obtained for Q, and $relevant(R)$ indicates the number of tuple trees present in R which are relevant to Q.

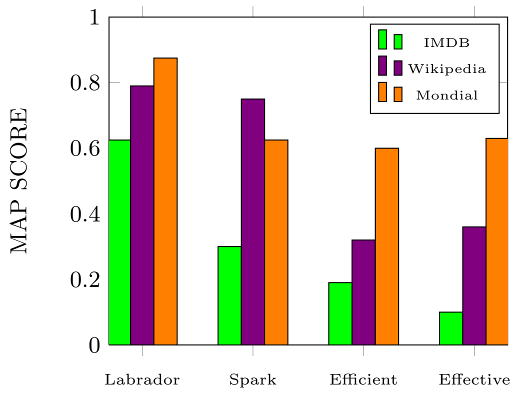

Figure 3: MAP ranking

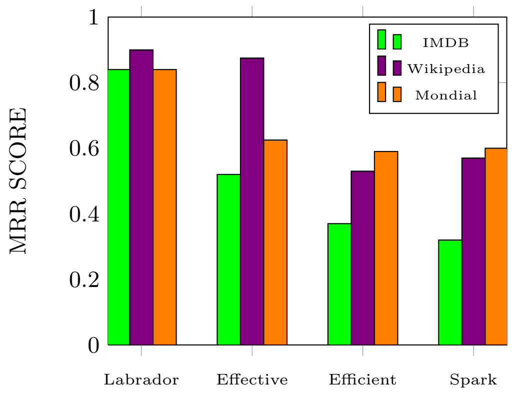

Figure 4: MRR ranking

$$prec(Q) = \frac{relevant(R)}{|R|} \tag{6}$$

The Average Precision metric for query Q, denoted by $AP(Q)$, is represented in Equation 7. Here, $P_c(b)$ is the Precision calculated until cut-off $b \leq |R|$ in the result set list, and $rel(b) = 1$ if the tuple tree at rank b in the result set is relevant; otherwise, $rel(b) = 0$.

$$AP(Q) = \frac{\sum_{b=1}^{|R|} rel(b) \times P_c(b)}{relevant(R)} \tag{7}$$

The MAP for Q_set, denoted by $MAP(Q\_set)$, is represented in Equation 8.

$$MAP(Q\_set) = \frac{\sum_{i=1}^{k} AP(Q_i)}{k} \tag{8}$$

The benchmark framework evaluates the quality of entire KWS, which contain multiple components, and the CNs generated by various KWS to answer a keyword query might differ[10]. Since, our goal is to evaluate only the quality of the various CN scoring functions, instead of the entire KWS, we performed an empirical study using the benchmark technique on a single set of CNs for each keyword query. Also, the LP technique [8] is not separately analyzed here, because it uses the Efficient scoring function [4]; hence, its result will be exactly the same as that obtained for Efficient [4]. The different scoring functions were applied on the merged result set of these CNs to perform the quality analysis. For this setup, the MAP and MRR performance behavior of the functions are presented in Figures 3 and 4, respectively. As evident in these figures, LSF clearly performs better than the scoring functions proposed earlier in the literature: Spark, Efficient and Effective, with regard to *both metrics*.

4 ROOT RANK OPERATOR

We move on in this section to considering internal changes to the database engine in order to facilitate the computation of RSD-based scoring functions. We start with the design of the *Root Rank* operator. Firstly, due to the RSD characteristic, complete information about the result set is required prior to computing the LSF. Therefore, the Root Rank operator is introduced at the root of the CNEQ plan tree. Secondly, the relational inverted index of SQL-Wrapper is replaced by an internal data structure that has a memory-based component for storing term frequencies, backed up by a disk-based component which stores the frequencies of those terms that could not be accommodated within the memory component. This inverted index is used for both updating and accessing the term frequencies. Since, the LSF uses string equality operations to calculate the term frequencies, a hashing technique is used to build the index. The index construction is deliberately kept very simple to ensure that, it is a light-weight operation within the engine and for evaluation purposes. In future, more advanced indexes can be considered depending on the performance requirements.

The logical structure of the index is shown in Figure 5. The In-Memory hash buckets form the memory component of the inverted index, while the hash files which are chained to these In-Memory buckets form the disk component. Each tuple in the inverted index is composed of 3 fields: term, attribute and frequency. The field term indicates a particular term present in the result set, attribute indicates the attribute of the result set to which the term belongs and frequency indicates the frequency of the term in the result set. Each memory bucket stores a single inverted index tuple. If a collision occurs, the new tuple will be stored in the hash file attached to the memory bucket. Since each inverted index tuple requires very little memory, a large number of tuples can be accommodated in the memory buckets. Also, hashing is invoked for disk storage, which results in limited number of disk I/Os to update or access term frequencies.

The execution details of the Root Rank operator are outlined in Algorithm 1. Let, Q and C indicate the submitted keyword query, and one of the CN qualified for answering Q, respectively. Let, Q contain n keyword query terms. Initially, the plan tree $plan\_tree(C)$, for executing the CNEQ corresponding to C is generated by the query optimizer through: $call\_optimizer(C)$, and PL indicates the root node of the

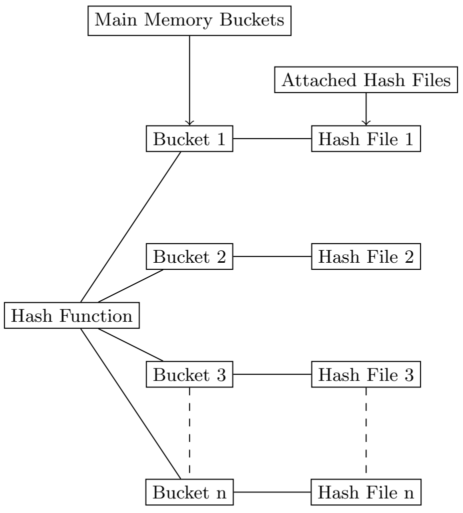

Figure 5: Memory and Disk-resident Inverted Index

plan_tree. Empty *plan_tree* node *rnode* is created to accommodate the Root Rank operator through *create_plan_node*(). The PL can be made as either the left child or right child of *rnode*; however, just for convenience PL is made the left child of *rnode* and the right child of *rnode* is set to $NULL$.

Specifically, there are two passes involved in Algorithm 1. The first pass is involved in inverted index construction; whereas, in the second pass, the actual scoring and ranking of tuple trees is performed. In the first pass, each tuple tree denoted by T, is extracted from *rnode* through *tuple_extractor*(*rnode.leftchild*). Let $(A_1, A_2, ...A_n)$ indicate the attributes of T to which Q is mapped. Each $A_i (1 \leq i \leq n)$ is tokenized through *split*(A_i) to produce the token or term set $\{t_{ij}\}$. Each t_{ij} is hash-mapped to a unique in-memory hash bucket indicated by $bucket_{ij}$ of the proposed inverted index through $hash(t_{ij})$. There are basically three mutually exclusive cases after mapping of t_{ij} to $bucket_{ij}$. In the first case, $bucket_{ij}$ does not contain any stored term, which is indicated through $bucket_{ij}.fill = 0$. Hence, t_{ij} is stored in $bucket_{ij}$ using the variable $bucket_{ij}.term$. Correspondingly, the attribute to which t_{ij} belongs, and its initial frequency is stored in $bucket_{ij}.attribute$ and $bucket_{ij}.frequency$, respectively. To indicate that $bucket_{ij}$ already contains a term, the variable setting $bucket_{ij}.fill = 1$ is used. In the second case, the $bucket_{ij}$ already stores the same term t_{ij}, which is identified through the conditions: $bucket_{ij}.fill = 1$, $bucket_{ij}.attribute = A_{ij}$ and $bucket_{ij}.term = t_{ij}$. Hence, the frequency of t_{ij} is updated through $bucket_{ij}.frequency++$. In the final case, the $bucket_{ij}$ contains another term than t_{ij}. In this case, t_{ij} is stored in the hash file appended to $bucket_{ij}$ indicated by $file_{ij}$ by opening the required hash file through $open(file_{ij})$, and updating the term frequency of t_{ij} through $update(file_{ij}, t_{ij}, A_i)$. Clearly, these three handling cases of t_{ij} will ensure that the term frequencies for each t_{ij} will be updated correctly in the inverted index. Finally, T is stored in the disk using $store(disk, T)$ for retrieving and using it in the second pass.

In the second pass, the disk-stored tuple trees during the first pass are retrieved one-by-one through the function: *extract_tuple_tree*(*disk*). Each extracted tuple tree indicated by T is again subjected to tokenization on the same attributes used in first pass, which results in the production of token/term set. Each t_{ij} is subjected to hash bucket mapping inside the proposed inverted index as outlined in the first pass. For each t_{ij}, there are two mutually exclusive cases to identify its term frequencies. In the first case, t_{ij} is found inside $bucket_{ij}$ identified through conditions: $bucket_{ij}.attribute = A_{ij}$ and $bucket_{ij}.term = t_{ij}$. Here, the frequency of t_{ij} indicated by $t_{ij}.frequency$ is assigned with the value $bucket_{ij}.frequency$. In the second case, t_{ij} is located inside $file_{ij}$. Here, $t_{ij}.frequency$ is calculated through the disk access function $seek(file_{ij}, t_{ij}, A_i)$. It is clear that, these two handling cases of t_{ij} will ensure that, the term frequencies for each t_{ij} is retrieved correctly from the inverted index. Finally, after calculating all the term frequencies for the terms $\in T$, the LSF is applied on T through $LSF\_score(T)$, and final top-K tuples trees are computed by sorting the tuple trees based on their LSF scores through $LSF\_sort(\{T\})$.

The correctness of Algorithm 1 is proved in Theorem 4.1.

THEOREM 4.1. *(Algorithm Correctness)*
For a given CN C, the Root Rank operator produces the exact top-K tuple trees corresponding to LSF scored result set of C.

PROOF. Suppose the exact top-K tuples tree list for C be indicated as $L_C = (T_1, T_2, ...T_k)$. The Algorithm 1 cannot produce L_C under exactly two scenarios. In the first scenario, some of the tuple trees in L_C are not considered for scoring. However, this scenario is not plausible because *tuple_extractor*(*rnode.leftchild*) ensures that, every tuple tree for C is considered for the scoring process. In the second scenario, some of the term frequencies might not be correctly calculated. However, again this scenario is not plausible because the three handling cases for t_{ij} in the first pass of Algorithm 1 ensures that, every term belonging to result set of C is correctly updated along with its frequency in the inverted index. Similarly, the two handling cases for t_{ij} in second pass ensure that, the exact term frequency is retrieved. Thus immediately proving the Theorem.

□

5 EXPERIMENTAL EVALUATION

For simplicity, the Root Rank operator is introduced by overloading the ilike operator, without making any addition to the SQL language, inside PostgreSQL (9.1.2). The presence of an *ilike* operator inside the SQL query, signals the optimizer to utilize the Root Rank operator to create the plan tree. The *ilike* operator overloading is optional, which can be disabled through configuration parameters.

Algorithm 1 Root Rank Operator Execution Algorithm

```
/*Plan Tree Modification/*
$plan\_tree(C) = call\_optimizer(C)$
$rnode = create\_plan\_node()$
$rnode.leftchild = PL$
$rnode.rightchild = NULL$
/*First Pass/*
while $tuple\_extractor(rnode.leftchild)!$ do
    $T = tuple\_extractor(rnode.leftchild)$
    for $i = 1 \to n$ do.
        $split(A_i)$.
        for each $t_{ij} \in A_i$ do.
            $bucket_{ij} = hash(t_{ij})$
            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
                $open(file_{ij})$
                $update(file_{ij}, t_{ij}, A_i)$
            end if
        end for
    end for
    $store(disk, T)$
end while
/*Second Pass/*
while $extract\_tuple\_tree(disk)!$ do
    $T = extract\_tuple\_tree(disk)$
    for $i = 1 \to n$ do.
        $split(A_i)$.
        for each $t_{ij} \in A_i$ 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
                $open(file_{ij})$
                $t_{ij}.frequency = seek(file_{ij}, t_{ij}, A_i)$
            end if
        end for
    end for
    $LSF\_score(T)$
end while
$LSF\_sort(\{T\})$.
```

Figure 6, shows an example of a SQL query which utilizes the Root Rank operator to obtain the top-K results of the CN:
Proceeding.title$^{\text{data}}$ × Inproceeding.title$^{\text{mining}}$× Publisher.name$^{\text{springer}}$(DBLP).

```
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 6: Top-K SQL query for Keyword Search

All the four techniques discussed in our study – that is, the Single Pipeline [4], Global Pipeline [4], Block Pipeline [5] techniques from the literature, and the Root Rank operator proposed here – were implemented and subjected to a detailed performance evaluation. Specifically, the Effective [6] scoring function was implemented through straightforward implementation technique. The Single Pipeline, Global Pipeline and Block Pipeline techniques were also implemented through the imperative programing construction described in [4, 5].

Due to our focus on addressing execution efficiency, the DBLP [5] and Wikipedia [11] datasets were used in our experiments; specifically, they produce CNs with large result sets, when compared to the other datasets used in this work – IMDB and Mondial. Due to space limitations, the details regarding DBLP and Wikipedia datasets are not outlined, and the complete details of these datasets are outlined in [5] and [11] respectively. Since benchmark keyword queries for performance evaluation do not readily exist for these datasets, we constructed queries from three sources:

(1) Queries used for identifying experts in [19]. This procedure involves finding the experts in different fields by using the DBLP dataset. The field names on which the expert finding algorithm was executed were used as keyword queries in this experiment.
(2) Queries which were used in other KWS [14].
(3) Queries which were used for result quality evaluation in the benchmark of [11].

5.1 Single CN Scenario

Our experimental analysis begins with a simple case in which keyword queries with only *2 terms* are analyzed. Subsequently, complex cases involving a large number of terms are considered. For every keyword query, only a single CN which has a large result cardinality is selected. These CNs involve a single join of two relations. The execution performance of the various implementation techniques for different CNs is illustrated in Figures 7 and 8, where the different CNs are indicated through numbers. The top-100 CN tuple trees were retrieved from each technique.

In the performance figures, it is clear that the Root Rank operator performs the best among all techniques. The Single Pipeline and Block Pipeline techniques also exhibit good performance; however, they do not scale to the performance level of Root Rank, because, as described in Section 2, both

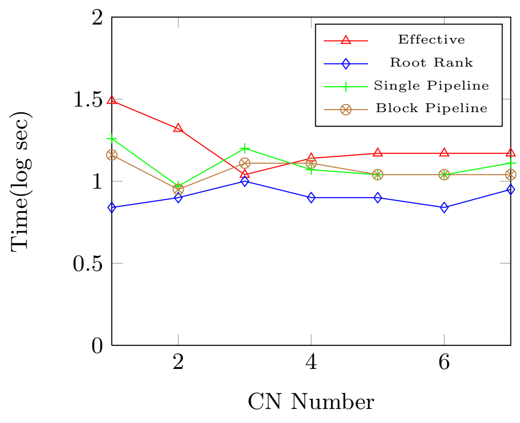

Figure 7: CN Execution(DBLP)

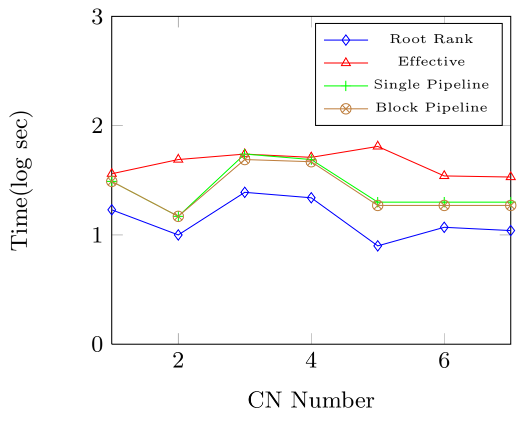

Figure 8: CN Execution(Wiki)

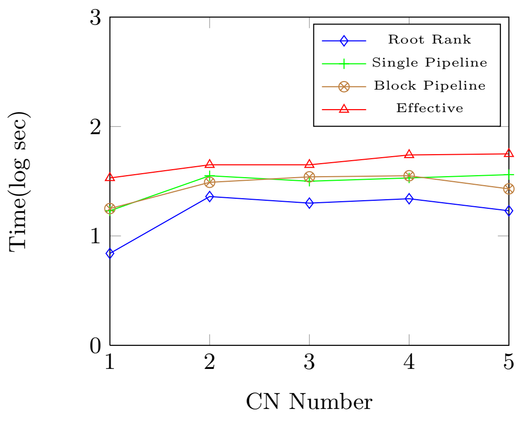

Figure 9: CN Execution(DBLP)

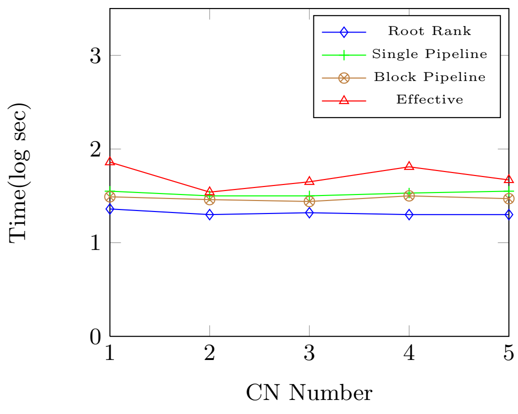

Figure 10: CN Execution(Wikipedia)

these techniques require large number of parametric queries to probe and produce the top-K results [5]. Requirement of executing such large number of parametric queries can result in performance degradation, whereas Root Rank completely avoids executing parametric queries. Also, when top-100 results are increased to top-200 or more, the performance penalty due to the execution overhead of parametric queries in Single Pipeline and Block Pipeline techniques, becomes more severe. Finally, the Effective technique exhibits mediocre performance, due to the lack of a specialized implementation mechanism.

5.2 Multi-Term Keyword Query Scenario

The performance of different techniques when executed on CNs generated for multi-term keyword queries, featuring between 3 and 5 terms, is illustrated in Figures 9 and 10 for the DBLP and Wikipedia datasets, respectively. Each CN had more than two joins, and the top-100 tuple trees were retrieved from each technique.

We again observe that the Root Rank operator performs clearly better than the other techniques, for the same reasons as those explained for the single CN scenario in Section 5.1.

5.3 Complete Execution Scenario for Keyword Query

Until now, only the execution performance of different techniques WRT individual CNs was discussed. However, answering a keyword query also involves the scoring of the result set of *all* qualifying CNs. In order to perform integration of Root Rank operator in this framework, we construct a simple Operator-K technique, which provides the top-K answers by executing every qualified CN with the Root Rank operator, and merging the result sets of all the CNs. Similarly, the Effective-K technique obtains the top-K results of each CN through the application of Effective scoring function[6], and merges the results to obtain the final top-K result set.

The performance of different techniques in the above scenario are shown in Figures 11 and 12, for the DBLP and Wikipedia datasets, respectively (the queries are identified by numbers in the figures). The top-100 tuple trees were

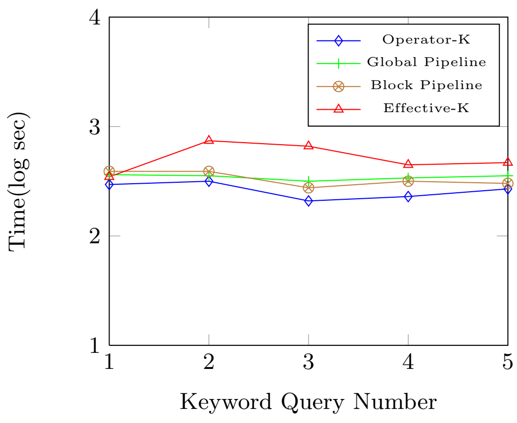

Figure 11: KWS Query Execution(DBLP)

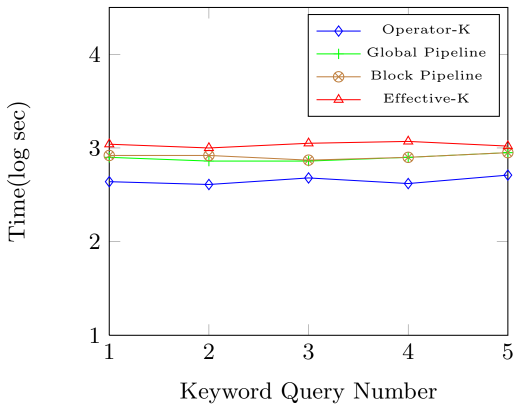

Figure 12: KWS Query Execution(Wiki)

retrieved from each technique. We observe in the figures that the Operator-K technique is the clear winner WRT keyword query answering performance. The Global-Pipeline technique, like Single Pipeline and Block Pipeline techniques, suffers due to execution overhead of large number of parametric queries. Wherease, the Effective-K technique exhibits mediocre performance, again due to lack of a specialized implementation mechanism.

6 CONCLUSION

In this work, the superior user relevance quality of LSF has been demonstrated. A new relational operator called Root Rank has been introduced to perform KWS result ranking using LSF. The Root Rank operator has justified its introduction by providing excellent execution efficiency benefits over other contemporary techniques.

Looking to the future, there is a need of new operators not just in the traditional RDBMS, but also in nascent systems such as columnar and probabilistic databases. These advanced database systems currently do not even have result scoring functions. By developing scoring functions and operators, complete integration of IR result ranking techniques with all kinds of database systems can be achieved.

Appendix A LSF [2]

For the DBLP dataset [5], an example Candiadte Network: $\mathsf{Proceeding.title}^{\mathsf{Data}} \times \mathsf{InProceeding.title}^{\mathsf{Mining}}$ is generated, and the LSF for ranking the result set is given in Equation 9.

$$\boxed{score(T) = n * \frac{\cos(p.title, \overrightarrow{p.title}) + \cos(i.title, \overrightarrow{i.title})}{2}} \tag{9}$$

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

$$w_k = log(1 + \frac{|result\_set|}{f_k})$$

$$w_{data} = log(1 + \frac{|result\_set|}{f_{data}})$$

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

$$w_{mining} = log(1 + \frac{|result\_set|}{f_{mining}})$$

$$w_r = log(1 + \frac{|result\_set|}{f_r})$$

Here, T indicates a tuple tree, and $score(T)$ is its LSF score, n is a positive constant, $|result\_set|$ is the cardinality of the CN result set, w_k is the weight of the term $k \in$ p.title(Proceeding.title) of the result set, w_r is the weight of the term $r \in$ i.title(Inproceeding.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 in the result set, f_r is the frequency of the term $r \in$ i.title in the result set, f_{data} is the frequency of the term $data \in$ p.title in the result set, and f_{mining} is the frequency of the term $mining \in$ i.title in the result set.

For a m term keyword query, the LSF assumes the form given in Equation 10. Here, $A_1, A_2.....A_m$ are the attributes on which the keyword query terms are mapped, and number of relations involved in the tuple tree T is given by $size(T)$.

$$\boxed{score(T) = n * \frac{\sum_{i=1}^{m} \cos(A_i, \overrightarrow{A_i})}{size(T)}} \tag{10}$$

REFERENCES

[1] P. Jaehui, and L. Sang-goo. Keyword Search in Relational Databases. *Knowledge and Information Systems*, Volume 26, Issue 2, pp 175–193, doi: 10.1007/s10115-010-0284-1, 2010.
[2] C. Pável, Da Silva A. S., M. Filipe, De-Moura E. S., and Laender A. H. F. LABRADOR: Efficiently Publishing Relational Databases on the Web by using Keyword-based Query Interfaces. *Information Processing and Management*, Volume 43, pp 983–1004, doi: 10.1016/j.ipm.2006.09.018, 2007.

[3] H. Vagelis, and P. Yannis. Discover: Keyword Search in Relational Databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, pp 670–681, 2002.

[4] G. Luis, H. Vagelis, and P. Yannis. Efficient IR-style Keyword Search over Relational Databases. In *Proceedings of the 29th International Conference on Very large Databases*, Berlin, Germany, pp 850–861, 2003

[5] L. Yi, L. Xuemin, W. Wei, and Z. Xiaofang. Spark: Top-k Keyword Query in Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, pp 115–126, doi: 10.1145/1247480.1247495, 2007.

[6] C. Abdur, L. Fang, Y. Clement, and M. Weiyi. Effective Keyword Search in Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, pp 563–574, doi: 10.1145/1142473.1142536, 2006.

[7] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, and X. Yu. Meaningful Keyword Search in Relational Databases with Large and Complex Schema. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, pp 411-422, doi:10.1109/ICDE.2015.7113302, 2015.

[8] Y. Xu. Scalable Top-k Keyword Search in Relational Databases. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, doi: 10.1007/s10586-017-1232-6, 2017.

[9] P. D. Oliveira, A. D. Silva, and E. D. Moura. Ranking Candidate Networks of Relations to Improve Keyword Search over Relational Databases. In *Proceedings of IEEE 31st International Conference on Data Engineering*, Seoul, pp 399–410, doi: 10.1109/ICDE.2015.7113301, 2015.

[10] B. Akanksha, R. Ian, L. Jiexing, D. AnHai, and N. Jeffrey. Towards Scalable Keyword Search over Relational Data. In *Proceedings of VLDB Endowment*, Volume 3, pp 140–149, doi: 10.14778/1920841.1920863, 2010.

[11] C. Joel, and Weaver A. C. A Framework for Evaluating Database Keyword Search Strategies. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, Toronto, Canada, pp 729–738, doi: 10.1145/1871437.1871531, 2010.

[12] Aditya B., B. Gaurav, C. Soumen, H. Arvind, N. Charuta, Parag, and Sudarshan S. BANKS: Browsing and Keyword Searching in Relational Databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, pp 1083–1086, 2002.

[13] C. Soumen, K. Varun, P. Shashank, Sudarshan S., D. Rushi, and K. Hrishikesh. Bidirectional Expansion for Keyword Search on Graph Databases. In *Proceedings of the 31st International Conference on Very Large Databases*, Trondheim, Norway, pp 505–516, 2005.

[14] H. Hao, W. Haixun, Y. Jun, and Yu P. S. BLINKS: Ranked Keyword Searches on Graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, pp 305–316, doi: 10.1145/1247480.1247516, 2007.

[15] B. Ding, Yu J.X., S. Wang, Q. Lu, X. Zhang, and X. Lin. Finding Top-k Min-Cost Connected Trees in Databases. In *IEEE 23rd International Conference on Data Engineering*, pp 836-845, doi: 10.1109/ICDE.2007.367929, 2007.

[16] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k Nearest Keyword Search on Large Graphs. In *Proceedings of VLDB Endowment*, Volume 6, pp 901-912, doi: 10.14778/2536206.2536217, 2013.

[17] Y. Yuan, X. Lian, L. Chen, Jeffery X. Y., G. Wang, and Y. Sun. Keyword Search over Distributed Graphs with Compressed Signature. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pp 1212-1225, doi: 10.1109/TKDE.2017.2656079, 2017.

[18] X. Yu, and H. Shi. CI-Rank: Ranking Keyword Search Results based on Collective Importance. In *Proceedings of IEEE 28th International Conference on Data Engineering*, pp 78-89, doi: 10.1109/ICDE.2012.69, 2012.

[19] C. Pavel, M. Catarina, and M. Bruno. Learning to Rank for Expert Search in Digital Libraries of Academic Publications. In *Progress in Artificial Intelligence: 15th Portuguese Conference on Artificial Intelligence*, Proceedings, Lisbon, Portugal, pp 431–445, doi: 10.1007/978-3-642-24769-9-32, 2011.